Introduction

When we declare a function in JavaScript, we often expect it to be invoked with arguments of a certain type. JavaScript does not share our expectations: its type flexibility often allows functions to be invoked with unexpected argument types. Even when this doesn’t result in thrown errors, there can be negative consequences:

function printLengthOfText(text) {

console.log(text.length);

}

printLengthOfText(3); // Prints: undefined

JavaScript developers have found error-handling solutions to avoid such undesirable effects, but these techniques can be cumbersome:

function printLengthOfText(text) {

if (typeof text !== 'string') {

throw new Error('Argument is not a string!');

}

console.log(text.length);

}

printLengthOfText(3); // Error: Argument is not a string!

In the code above, the function first checks the argument type. If it doesn’t match the expected type, an error is thrown; otherwise, it continues with its intended operation.

Before we explore how TypeScript handles this issue, let’s practice fixing some JavaScript type-related bugs. These are the kinds of problems that TypeScript helps us diagnose early on, before they become hard to spot.

Instructions

1.This code looks pretty good! Let’s go ahead and run it using node index.js in the terminal. What could possibly go wrong?

Checkpoint 2 Passed

2. AHHH! Error messages! They’re telling us that printOperations() should take in two numbers. It looks like it’s receiving two strings now. Change the function call (printOperations('6', '6')) to pass the right type of data. Then click “Run” to save your changes.

Checkpoint 3 Passed

Stuck? Get a hint

3. Try node index.js in the terminal one more time. No more error messages! But the function exclaim(name, count) still isn’t working properly. It is meant to print name a total of count times.

Do you spot the bug where it’s being called?

Fix the bug in exclaim()‘s function call so that the code prints 'Muriel!' six times.

function printOperations(a, b) {

  if (typeof a !== 'number' || typeof b !== 'number') {

    throw new Error('Both arguments must be numbers!');

  }

  console.log(a + b, a / b);

}

// The function call below should print: 12 1

printOperations(6, 6);

function exclaim(name, count) {

  for (let i = 0; i < count; i += 1) {

    console.log(`${name}!`);

  }

}

// Exclaim 'Muriel!' six times

exclaim('Muriel',6);

**Parameter Type Annotations**

In TypeScript, function parameters may be given type annotations with the same syntax as variable declarations: a colon next to the name. The type annotations ensure that the parameters are of the correct type:

function greet(name: string) {  
  console.log(`Hello, ${name}!`);  
}  
   
greet('Katz'); // Prints: Hello, Katz    
   
greet(1337); // Error: argument '1337' is not assignable to parameter of type 'string'

By placing : string after the name parameter, we’re declaring that name is of type string. Any invocation of greet() should pass a value of type string as the first argument, or else an error will be thrown.

Parameters that we do not provide type annotations for are assumed to be of type any—the same way variables are.

function printKeyValue(key: string, value) {  
  console.log(`${key}: ${value}`);  
}  
   
printKeyValue('Courage', 1337); // Prints: Courage: 1337  
printKeyValue('Mood', 'scared'); // Prints: Mood: scared

Here, the parameter value is an any variable: it’s compatible with any type.

**Instructions**

**1.**The triple() function is meant to multiply a value by three. Add a type annotation to the value parameter to indicate that it’s meant to be a number.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**greetTripled() is meant to take in a string and a number. Add type annotations to its type parameters as well.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**Everything seems to look good. Let’s quickly compile our code and move on to the next exercise. To do this, simply run tsc in the terminal.

Checkpoint 4 Passed

**4.** Woah! TypeScript found a bug, thanks to our type annotations. It looks like greetTripled() is being called with the wrong types of arguments. Fix the code to print Hiya, 15!.

function triple(value:number) {

  return value \* 3;

}

function greetTripled(greeting:string, value:number) {

  console.log(`${greeting}, ${triple(value)}!`);

}

greetTripled( 'Hiya',5);

**Optional Parameters**

TypeScript normally gives an error if we don’t provide a value for all arguments in a function. This isn’t always desirable; sometimes we’d like to skip providing values.

function greet(name: string) {  
  console.log(`Hello, ${name || 'Anonymous'}!`);  
}  
   
greet('Anders'); // Prints: Hello, Anders!  
greet(); // TypeScript Error: Expected 1 arguments, but got 0.

When the code snippet above is compiled to JavaScript, the greet() function will correctly print 'Hello, Anonymous!'. That’s because when no arguments are passed in, name has the falsy value undefined, which means that name || 'Anonymous' evaluates to 'Anonymous'. Since the final code works as intended, we want to prevent TypeScript from throwing errors:

To indicate that a parameter is intentionally optional, we add a ? after its name. This tells TypeScript that the parameter is allowed to be undefined and doesn’t always have to be provided.

function greet(name?: string) {  
  console.log(`Hello, ${name|| 'Anonymous'}!`);  
}  
   
greet(); // Prints: Hello, Anonymous!

**Instructions**

**1.**

Our proclaim() method is meant to default to printing 'I'm not ready...' when called without arguments. Mark its status parameter as optional.

Afterwards, feel free to run tsc in the terminal to verify that there are no errors.

function proclaim(status?: string) {

  console.log(`I'm ${status || 'not ready...'}`);

}

proclaim();

proclaim('ready?');

proclaim('ready!');

**Default Parameters**

If a parameter is assigned a default value, TypeScript will infer the variable type to be the same as the default value’s type. (This is similar to how TypeScript infers the type of an initialized variable to be the same as the type of its initial value.)

The following code snippet logs a string to greet a user’s name, and defaults to the name 'Anonymous' if no name is provided.

function greet(name = 'Anonymous') {  
  console.log(`Hello, ${name}!`);  
}

The function greet() can receive a string or undefined as its name parameter—if any other type is provided as an argument, TypeScript will consider that a type error.

This is a cleaner way of getting the same functionality we had in the previous exercise. There, we used ? to mark the name parameter as optional. But parameters with default values don’t need a ? after their name, since assigning a default value already implies that they’re optional parameters.

**Instructions**

**1.**

The function proclaim() would be a little easier to read if it used default parameters, don’t you think? Give status and repeat default parameter values of 'not ready...' and 1, respectively. You should also remove the unnecessary parameter type annotations.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Now that proclaim() uses default parameters, remove the unnecessary || statements from its code.

function proclaim(status="not ready...", repeat=1) {

  for (let i = 0; i < repeat; i += 1) {

    console.log(`I'm ${status}`);

  }

}

proclaim();

proclaim('ready?');

proclaim('ready!', 3);

**Inferring Return Types**

TypeScript can also infer the types of values returned by functions. It does this by looking at the types of the values after a function’s return statements.

function createGreeting(name: string) {  
  return `Hello, ${name}!`;  
}  
   
const myGreeting = createGreeting('Aisle Nevertell');

Here’s how TypeScript can infer myGreeting above to be of type string:

1. createGreeting()’s return statement is followed by a string variable, so createGreeting() is inferred to return string.
2. createGreeting('Aisle Nevertell') therefore must result in a value of type string.
3. myGreeting is initialized to createGreeting('Aisle Nevertell'), which is a value with the type string.

Cool! Let’s see how this can help us fix bugs:

function ouncesToCups(ounces: number) {  
  return `${ounces / 16} cups`;  
}  
   
const liquidAmount: number = ouncesToCups(3);  
// Type 'string' is not assignable to type 'number'.

Here, TypeScript was able to infer that liquidAmount was being assigned a value of type string, despite it being declared as a variable of type number. This correctly results in an error being surfaced.

**Instructions**

**1.**

Challenge! Using what you’ve learned above, create a variable myVar with the type number. To make this more interesting:

* You must not use the : character.
* You must not type any numbers into your code.
* You may not use functions other than the one provided in **index.ts**.

function getRandomNumber(){

  return Math.random();

}

let myVar = getRandomNumber();

**Explicit Return Types**

If we’d like to be explicit about what type a function returns, we can add an explicit type annotation after its closing parenthesis. Here, we use the same syntax as other type annotations, a colon followed by the type. TypeScript will produce an error for any return statement in that function that doesn’t return the right type of value.

function createGreeting(name?: string): string {  
  if (name) {  
    return `Hello, ${name}!`;  
  }  
   
  return undefined;  
  //Typescript Error: Type 'undefined' is not assignable to type 'string'.  
};

We can also explicitly state return types for arrow functions (which were defined in the ES6/ES2015 version of JavaScript). We’ll see the same kinds of error messages for both function types.

const createArrowGreeting = (name?: string): string => {  
    if (name) {  
        return `Hello, ${name}!`;  
    }  
   
    return undefined;  
    // Typescript Error: Type 'undefined' is not assignable to type 'string'.  
};

Now let’s take a look at an example of how explicit return type annotation can be useful when we’re working with other people’s code.

**Instructions**

**1.**

You’re designing an online multiplayer game where fruit battles against fruit for culinary supremacy.

The function getUserChoice() gets a user’s fruit choice entered as a number from 1 to 12. Then, returnFruit() runs a fruit-generating function (f1() through f12()) to return a filename '<fruitname>.jpg'.

First, let’s verify that this works. Add console.log(returnFruit()) to the code.

Click “Run” to save your changes, compile using tsc, then execute the code using node.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

You may or may not have noticed a problem here. In fact, only 11 out of 12 of the functions f1() through f12() correctly return a fruit .jpg filename. Run the code a bunch more times until you see the problem. This might take a few tries, since getUserChoice() is always different.

Checkpoint 3 Passed

**3.**

2545434685676. What? That’s not a fruit. Clearly, one of the functions f1() through f12() are not working correctly. But which one? It would be a nightmare to have to look through all that source code! (That’s why we’ve disabled that option for you.)

You could just run them all one by one. But doing something 12 times is too much work. Instead, let TypeScript do the work for you. Think about which kind of type annotation can help you here. Then, implement it!

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Finally, we can figure out which of the functions f1() through f12() have the problem. Once you know which one, type your answer as a code comment, in the following format:

//The problematic function is f2()

import {getUserChoice,f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12} from './resources'

function returnFruit():string{

  let num = getUserChoice();

  switch(num){

    case 1 : return f1();

    case 2 : return f2();

    case 3 : return f3();

    case 4 : return f4();

    case 5 : return f5();

    case 6 : return f6();

    case 7 : return f7();

    case 8 : return f8();

    case 9 : return f9();

    case 10 : return f10();

    case 11 : return f11();

    case 12 : return f12();

  }

  return 'durian.jpg'

}

console.log(returnFruit())

**Void Return Type**

By now, we’ve made a pretty convincing case that type annotations are very useful and should always be used unless there’s a very good reason not to. They make everything neat and aid in understanding our code.

For these reasons, it is often preferred to use type annotations for functions, even when those functions don’t return anything. Example:

function logGreeting(name: string){  
  console.log(`Hello, ${name}!`)  
}

The function logGreeting() simply logs a greeting to the console. There is no returned value, so we must treat the return type as void. A proper type annotation for this function would look like this:

function logGreeting(name:string): void{  
  console.log(`Hello, ${name}!`)  
}

**Instructions**

**1.**

Use TypeScript to check this fruit salad code for type errors.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

It looks like the type annotation for the return type of makeFruitSalad() isn’t quite right. Please correct this!

function makeFruitSalad(fruit1:string, fruit2:string):void{

  let salad=fruit1+fruit2+fruit2+fruit1+fruit2+fruit1+fruit1;

  console.log(salad);

}

makeFruitSalad('banana','pineapple');

**Documenting Functions**

TypeScript recognizes JavaScript comment syntax:

// This is a single line comment   
   
/\*  
This is a   
multiline  
comment  
\*/

But it’s common in TypeScript to see a third comment style: documentation comments. A documentation comment is denoted with the first line /\*\* and a final line \*/. It’s common for each line within the comment to start with an asterisk (\*):

/\*\*  
\* This is a documentation comment  
\*/

Documentation comments are especially useful for documenting functions. We place a function’s documentation comment in the code directly above the function declaration. We can use special tags within the comment to highlight certain aspects of the function. We can use @param to describe each of the function’s parameters, and we can use @returns to describe what the function returns:

  /\*\*  
   \* Returns the sum of two numbers.  
   \*  
   \* @param x - The first input number  
   \* @param y - The second input number  
   \* @returns The sum of `x` and `y`  
   \*  
   \*/  
  function getSum(x: number, y: number): number {  
    return x + y;  
  }  
}

Many text editors will helpfully display documentation comments, for example, when hovering over a function name.

**Instructions**

**1.**

We grabbed a few functions from earlier exercises. They are sorely missing some documentation comments…

Add a documentation comment makeFruitSalad() function. Your comment should include:

* a brief description of what the function does
* a first @param tag for the first parameter
* a second @param tag for the second parameter
* a @returns tag explaining what the function returns (in this case, nothing)

Checkpoint 2 Passed

**2.**

Add a documentation comment proclaim() function. Your comment should include:

* a brief description of what the function does
* a first @param tag for the first parameter
* a second @param tag for the second parameter
* an @returns tag explaining what the function returns (again, in this case, nothing)
* /\*\*
* \* Logs a concatenated string of Salads
* \* @param fruit1 - the first input string
* \* @param fruit2 - the second inoput string
* \* @returns nothing just logs to console
* \*/
* function makeFruitSalad(fruit1: string, fruit2: string): void {
* let salad = fruit1 + fruit2 + fruit2 + fruit1 + fruit2+ fruit1 + fruit1;
* console.log(salad);
* }
* /\*\*
* \* Loops and logs status if i am not ready
* \* @param status - the first  with a default value of a string with text 'not ready ...'
* \* @param repeat - the second number based input with default set to 1
* \* @returns nothing just logs to console
* \*/
* function proclaim(status = 'not ready...', repeat = 1) {
* for (let i = 0; i < repeat; i += 1) {
* console.log(`I'm ${status}`);
* }
* }

**Review**

We learned all about how to use TypeScript to specify the types of function parameters and return types. Now we know how to:

* Give type annotations to function parameters.
* Deal with type annotations for optional parameters, which may have default values.
* Understand how TypeScript determines the return type of a function.
* Explicitly specify return types for functions, including for functions that don’t return anything.
* Use the above skills to diagnose and fix bugs in our code.

Great job!