**Introduction**

Our TypeScript journey has now led us to a new destination: Array-bia. Typing arrays is a little bit different than working with primitive types. This is because arrays usually contain many pieces of data. Keeping track of the array’s type means keeping track of every element’s type. For example:

let firstArray = [1, 2, 3, 4];  
let secondArray =  [5, '6', [7]];

We can see that firstArray has elements that are all number types. On the other hand, secondArray has elements of varying types: number, string, and Array. Both are examples of valid JavaScript arrays. As we’ll soon explore, TypeScript makes it very easy to keep track of element types in arrays, such as those above.

For now, though, let’s make life difficult by pretending that TypeScipt didn’t have special ways of typing arrays. How hard would it be to enforce type consistency?

### Instructions

**1.**

Imagine you are a software engineer for a company that keeps track of customer names in the array customersArray. The company’s codebase loops through this array to generate an annual report. The code was written by an untrained intern over 10 years ago, and it will crash the entire system if customersArray is found to contain any data of the number type!

Because you’re not using TypeScript, you need to write the function checkCustomersArray() to check that every element of customersArray is a string. For any element el that is not a string, this function should log an error message using the statement console.log(`Type error: ${el} should be a string!`).

Call the checkCustomersArray() function after you’ve defined it to see your messages print out.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

OK, you found a couple of errors. (Wouldn’t it be nice if this error checking was automatic?) You’ll fix them later. Right now, there is a more pressing concern. The array customersArray is constantly being altered by existing code. The last thing we want is for somebody to .push() a non-string value into this array.

Write an alternative pushing function called stringPush() that takes one argument: val, the value meant to be pushed into the array. The function should check if the val is a string and .push() it into the customersArray array only if it is a string.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Even with all this, people could ruin your precious string array by writing code like customersArray[4] = 4. You don’t have to waste time finding a way around this right now.

Now you can see how needlessly difficult manual type-checking is. And imagine, you would have to write even more code if you needed arrays of numbers! It’s all too much.

Click the “Run” button to move on to the next exercise, where you will learn how TypeScript makes it easy to maintain a typed array!

let customersArray = ['Custy Stomer', 'C. Oostomar', 'C.U.S. Tomer', 3432434, 'Custo Mer', 'Custopher Ustomer', 3432435, 'Kasti Yastimeur'];

//Write Your Code here:

function checkCustomersArray(arr){

  arr.forEach(element=>{

    if(typeof element!=="string"){

      console.log(`Type error: ${element} should be a string!`)

    }

  })

}

checkCustomersArray(customersArray)

function stringPush(val){

  if (typeof val === "string"){

    customersArray.push(val)

  }

}

**Array Type Annotations**

The TypeScript type annotation for array types is fairly straightforward: we put [] after the element type. In this code, names is an Array that can only contain strings:

let names: string[] = ['Danny', 'Samantha'];

An alternative method is to use the Array<T> syntax, where T stands for the type.

let names: Array<string> = ['Danny', 'Samantha'];

In the code above, the type, T, is string. We won’t use Array<T> syntax in this lesson, but it’s nice to be familiar with it.

As we may expect, we get a type error if we try to assign an array of numbers to a string[] variable:

let names: string[] = [1,2,3]; // Type Error!

That was just like an assignment error with primitive types. TypeScript arrays, however, can also throw errors when elements of the wrong type are added:

let names: string[] = ['Damien'];  
names.push(666) // Type Error!

Let’s practice some array type annotations!

### Instructions

**1.**

The code editor contains several array assignments. Add the appropriate type annotations to the array variables in the // Arrays: section.

// Arrays:

let bestNumbers = [7,77,4];

let bestLunches = ['chicken soup', 'non-chicken soup'];

let bestBreakfasts= ['scrambled eggs', 'oatmeal', 'tamago kake gohan', 'any kind of soup'];

let bestBooleans = [true, false];

Solution

// Arrays:

let bestNumbers:number[]= [7,77,4];

let bestLunches:string[] = ['chicken soup', 'non-chicken soup'];

let bestBreakfasts:string[]= ['scrambled eggs', 'oatmeal', 'tamago kake gohan', 'any kind of soup'];

let bestBooleans:boolean[] = [true, false];

// I added below myself just to check any array type annotations

let anyTypeOfElements:any[]=[1,2,"ABC",true,[1,"CDE"]]

console.log(anyTypeOfElements)

**Multi-dimensional Arrays**

So far we’ve looked at string[] arrays, but we could also have arrays that only contain numbers (number[]) or booleans (boolean[]). In fact, we can make arrays of any type whatsoever. We can also declare multidimensional arrays: arrays of arrays (of some type).

let arr: string[][] = [['str1', 'str2'], ['more', 'strings']];

Think of the string[][] above as short for (string[])[], that is, an array where every element has type string[]. We’ll explore more complex array types in later lessons.

The empty array ([]) is compatible with any array type:

let names: string[] = []; // No type errors.  
let numbers: number[] = []; // No type errors.  
names.push('Isabella');    
numbers.push(30);

Practice time!

### Instructions

1. Please provide type annotations for all the arrays provided in the // Multidimensional arrays: section.

**Problem**

// Arrays:

let bestNumbers: number[] = [7,77,4];

let bestLunches: string[] = ['chicken soup', 'non-chicken soup'];

let bestBreakfasts: string[]= ['fasting', 'oatmeal', 'tamago kake gohan', 'any kind of soup'];

let bestBooleans: boolean[] = [true, false];

// Multidimensional Arrays:

let bestMealPlan = [bestLunches, bestBreakfasts, ['baked potato', 'mashed potato']];

let bestBooleansTwice = [bestBooleans, bestBooleans];

let numbersMulti = [ [[1],[2,3]], [[7],bestNumbers] ];

**Tuples**

So far we’ve looked at arrays with all elements of the same type. But, as we know, JavaScript arrays are flexible and can have elements of different types. With TypeScript, we can also define arrays with a fixed sequence of types:

let ourTuple: [string, number, string, boolean] = ['Is', 7 , 'our favorite number?' , false];

In TypeScript, when an array is typed with elements of specific types, it’s called a *tuple*. The tuple above (ourTuple) contains the elements: 'Is', 7 , 'our favorite number?' , false and the tuple has a type of [string, number, string, boolean]. Tuple types specify both the lengths and the orders of compatible tuples, and will cause an error if either of these conditions are not met:

let numbersTuple: [number, number, number] = [1,2,3,4]; // Type Error! numbersTuple should only have three elements.  
let mixedTuple: [number, string, boolean] = ['hi', 3, true] // Type Error! The first elements should be a number, the second a string, and the third a boolean.

As far as JavaScript is concerned, tuples act just like arrays. They both have .length properties. We can access (or change) the elements of both using [index]. But despite their similarities, tuples and arrays do not have compatible types within TypeScript. Specifically, we can’t assign an array to a tuple variable, even when the elements are of the correct type:

let tup: [string, string] = ['hi', 'bye'];  
let arr: string[] = ['there','there'];  
tup = ['there', 'there']; // No Errors.  
tup = arr; // Type Error! An array cannot be assigned to a tuple.

Now let’s practice using tuples and tuple types!

**Tuples**

So far we’ve looked at arrays with all elements of the same type. But, as we know, JavaScript arrays are flexible and can have elements of different types. With TypeScript, we can also define arrays with a fixed sequence of types:

let ourTuple: [string, number, string, boolean] = ['Is', 7 , 'our favorite number?' , false];

In TypeScript, when an array is typed with elements of specific types, it’s called a tuple. The tuple above (ourTuple) contains the elements: 'Is', 7 , 'our favorite number?' , false and the tuple has a type of [string, number, string, boolean]. Tuple types specify both the lengths and the orders of compatible tuples, and will cause an error if either of these conditions are not met:

let numbersTuple: [number, number, number] = [1,2,3,4]; // Type Error! numbersTuple should only have three elements.  
let mixedTuple: [number, string, boolean] = ['hi', 3, true] // Type Error! The first elements should be a number, the second a string, and the third a boolean.

As far as JavaScript is concerned, tuples act just like arrays. They both have .length properties. We can access (or change) the elements of both using [index]. But despite their similarities, tuples and arrays do not have compatible types within TypeScript. Specifically, we can’t assign an array to a tuple variable, even when the elements are of the correct type:

let tup: [string, string] = ['hi', 'bye'];  
let arr: string[] = ['there','there'];  
tup = ['there', 'there']; // No Errors.  
tup = arr; // Type Error! An array cannot be assigned to a tuple.

Now let’s practice using tuples and tuple types!

### Instructions

**Array Type Inference**

TypeScript can infer variable types from initial values and return statements. Even still, we may not know exactly what type inference to expect when dealing with arrays. For example:

let examAnswers= [true, false, false];

What is the type of examAnswers? It seems it could equally well be boolean[] or [boolean, boolean, boolean]. In reality, it is always the first of these, since this is the less restrictive type. This enables us to expand the array:

examAnswers[3] = true; // No type error.

Since tuples have fixed lengths, we wouldn’t be able to add additional boolean elements to a tuple:

let tupleOfExamAnswers: [boolean, boolean, boolean] = [true, false, false];  
tupleOfExamAnswers[3] = true; // Type error! The tuple only has 3 elements.

We also get the same kind of type inference when we use the .concat() method:

let tup: [number, number, number] = [1,2,3];  
let concatResult = tup.concat([4,5,6]); // concatResult has the value [1,2,3,4,5,6].

In the code above, TypeScript infers the variable concatResult as an array of numbers, not a tuple.

The takeaway here is that type inference returns arrays. When we want tuples, we need to use explicit type annotations.

### Instructions

**1.**

Let’s do a tricky example. We have defined the tuple dogTup in the code editor. Your challenge is to define a variable myArr of the type string[].

Not so fast! The hard part is that you are not allowed to use the word Array, or the characters [ and ]. Use the .concat() method and your wits to solve the problem.

Stuck? Get a hint

**2.**

You figured it out! Now demonstrate that myArr is an array rather than a tuple by adding the code myArr[50] = 'not a dog' and running tsc. If myArr were a tuple, accessing element 50 would be impossible!

**Rest Parameters**

Assigning types to [rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) is similar to assigning types to arrays. Here’s a rest parameter example without types:

function smush(firstString, ...otherStrings){  
  let output = firstString;  
  for(let i = 0; i < otherStrings.length; i++){  
    output = output.concat(otherStrings[i]);  
  }  
  return output;  
}

This function concatenates all of its arguments. For example, calling: smush('hi ', 'there') returns the value 'hi there'.” The rest parameter otherStrings lets the function work with any number of parameters greater than zero:

smush('a','h','h','H','H','H','!','!'); // Returns: 'ahhHHH!!'.

The function works well, but it is not type safe. We don’t want a misguided coder to make a mistake like smush(1,2,3), when that would cause an error. TypeScript to the rescue! Type annotations for a rest parameter are identical to type annotations for arrays. The function with a correctly typed rest parameter is then:

function smush(firstString, ...otherStrings: string[]){  
  /\*rest of function\*/  
}

With this change, TypeScript will treat otherStrings as an array of strings. This means that smush(1,2,3) will result in a type error because [2,3] is not an array of strings.

Now, it’s your turn to write a rest parameter type!

### Instructions

**1.**

The code editor includes another function using rest parameters. The function, addPower(p, ...numsToAdd), adds up the pth powers of all the subsequent arguments. So the call addPower(2, 3, 4) returns 25, since 3 squared plus 4 squared is 25. (Here, we are using the exponentiation operator \*\*.)

Please add the appropriate type annotations for this function. Include the annotations for all arguments and the return type.

Stuck? Get a hint

**2.**

There is no such thing as a number to the power of 'a string'. Verify that the function results in a type error when you add the code addPower('a string', 4, 5, 6) and run tsc.