Chapter 3 – Complex Types

1. Introduction
   1. TypeScript makes it very easy to keep track of element types in arrays
   2. Manual type-checking is needlessly difficult and adds complications

|  |
| --- |
| let customersArray = ['Custy Stomer', 'C. Oostomar', 'C.U.S. Tomer', 3432434, 'Custo Mer', 'Custopher Ustomer', 3432435, 'Kasti Yastimeur'];  //Write Your Code here:  function checkCustomersArray() {  for (el of customersArray) {  if (typeof el != 'string') {  console.log(`Type error: ${el} should be a string!`);  }  }  }  function stringPush(val) {  if (typeof val != 'string') {  return;  }  customersArray.push(val);  } |

1. Array Type Annotations
   1. Type annotation for array types is fairly straightforward: [] after the element type
   2. Throw errors when elements of the wrong type are added

|  |
| --- |
| let names: string[] = ['Danny', 'Samantha'];  let names: string[] = ['Damien'];  names.push(666) // Type Error! |

* 1. Alternate - use the Array<T> syntax, where T stands for the type.

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

Exercise

|  |
| --- |
| // 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]; |

1. Multi-dimensional Arrays
2. Done using [][]

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

Exercise

|  |
| --- |
| // 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: string[][] = [bestLunches, bestBreakfasts, ['baked potato', 'mashed potato']];  let bestBooleansTwice: boolean[][] = [bestBooleans, bestBooleans];  let numbersMulti: number[][][] = [ [[1],[2,3]], [[7],bestNumbers] ]; |

1. Tuples
   1. Tuple acts like arrays
      1. Has .length properties
      2. Can access element using [index]
      3. But can’t assign an array to a tuple variable

|  |
| --- |
| 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. |

* 1. 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. |

Exericse

|  |
| --- |
| let favoriteCoordinates:[  number, number, string,  number, number, string, number];      favoriteCoordinates = [17, 45, 'N', 142, 30, 'E', -100];  favoriteCoordinates[6] = -6.825; |

1. Array Type Interface
   1. Typescript infers to type[] before tuple
   2. TypeScript can infer variable types from initial values and return statements

|  |
| --- |
| let examAnswers= [true, false, false];  examAnswers[3] = true; // No type error. |

* 1. We also get the same kind of type inference when we use the .concat() method
     1. **concatResult** as an array of numbers, not a tuple.

|  |
| --- |
| 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]. |

Exercise

|  |
| --- |
| // Don't change this part:  let dogTup: [string, string, string, string] = ['dog', 'brown fur', 'curly tail', 'sad eyes'];  // Your code goes here:  let myArr = dogTup.concat("");  myArr[50] = "not a dog"; |

1. Rest Parameters
   1. 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
      1. Here the rest parameters are, ‘h’, ‘h’, ‘H’, ‘H’, ‘H’, ‘H’, ‘!’

|  |
| --- |
| function smush(firstString, ...otherStrings){  let output = firstString;  for(let i = 0; i < otherStrings.length; i++){  output = output.concat(otherStrings[i]);  }  return output;  }  smush('a','h','h','H','H','H','!','!'); // Returns: 'ahhHHH!!'. |

* 1. The above is not safe.
     1. Allows other types in **otherStrings**
  2. Is made safe using typescript

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

Exercise

|  |
| --- |
| function addPower(p: number, ...numsToAdd: number[]): number {  let answer = 0;  for(let i = 0; i < numsToAdd.length; i++){  answer += numsToAdd[i] \*\* p;  }  return answer;  }  addPower('a string', 4, 5, 6); |

1. Spread Syntax
   1. 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
      1. Here the rest parameters are, ‘h’, ‘h’, ‘H’, ‘H’, ‘H’, ‘H’, ‘!’
   2. If not used, is awkward to read

|  |
| --- |
| gpsNavigate(40, 43.2, 'N', 73, 59.8, 'W', 25, 0, 'N', 71, 0, 'W') |

* 1. We can use tuple variables that represent the starting and ending coordinates to improve readability of code

|  |
| --- |
| let codecademyCoordinates: [number, number, string, number, number, string] = [40, 43.2, 'N', 73, 59.8, 'W'];  let bermudaTCoordinates: [number, number, string, number, number, string] = [25, 0 , 'N' , 71, 0, 'W'];  gpsNavigate(...codecademyCoordinates, ...bermudaTCoordinates);  // And by the way, this makes the return trip really convenient to compute too:  gpsNavigate(...bermudaTCoordinates, ...codecademyCoordinates);  // If there is a return trip . . . |

Exercise

|  |
| --- |
| function performDanceMove(moveName:string, moveReps:number, hasFlair:boolean):void{  console.log(`I do the ${moveName} ${moveReps} times !`);  if(hasFlair){  console.log('I do it with flair!');  }  }  let danceMoves: [string, number, boolean][] = [  ['chicken beak', 4, false],  ['wing flap', 4, false],  ['tail feather shake', 4, false],  ['clap', 4, false],  ['chicken beak', 4, true],  ['wing flap', 4, true],  ['tail feather shake', 4, true],  ['clap', 4, true],  ];  for (let danceMove of danceMoves) {  performDanceMove(...danceMove);  } |

Quiz

1. The code snippet prints the content of praises, a two-dimensional array of random strings, on the console. Fill in the code to initialize the variable, praises, with the correct value so that it only has five random messages.

Text

Description automatically generated

* 1. []

1. Which of these variable declarations is a tuple with correctly-assigned values?

Graphical user interface

Description automatically generated

* 1. B: const heading: [string, string, string, string] = ['Product', 'Price', 'Sale', 'Quantity']

1. Fill in the code to define a generic type for an array in TypeScript.

A picture containing text, screen, screenshot, hand

Description automatically generated

* 1. Array, <, >, [
  2. ]

1. Fill in the code with the correct return type for the function mySort().

Graphical user interface, text

Description automatically generated

* 1. Any[]

1. Which of the following is a valid function call for policy()?

Graphical user interface

Description automatically generated

* 1. A: Const adult: [string, number, Boolean] = [“Don”, 21, false];

1. Fill in the code to match the array types with the array values.

Graphical user interface, diagram, text

Description automatically generated with medium confidence

* 1. String[][], [‘Cereal’, ‘Toast’]
  2. String[][][], [lunch]

1. Which of these statements does NOT describe a tuple correctly?

Graphical user interface, website

Description automatically generated

* 1. D: An array can be assigned to a tuple when their elements are of the same type

1. Which of the following code snippets will generate a TypeScript type error?

Graphical user interface

Description automatically generated

* 1. A: let areacodes: number[] = [617, 413, 857, null];

1. Consider the following code snippet. What would be the correct type for the variable, updated?

Graphical user interface, application

Description automatically generated with medium confidence

* 1. D: Any[]

1. Which of the following function calls will produce a type error for the function, range(), which takes a rest parameter as an argument?

A picture containing graphical user interface

Description automatically generated

* 1. B: range([3,7,5])

Chapter 4 – Custom Types

1. Introduction

A screenshot of a video game

Description automatically generated with medium confidence

* 1. TypeScript can also be used to create *custom types*, rather than being limited to pre-defined types
  2. Custom types are what make TypeScript really fun and useful
     1. They enable type checking that’s tailored to your exact purposes

Example 1

|  |
| --- |
| let myVar: compType; |

Example 2

|  |
| --- |
| function testFn(param: compType): returnedCompType {  /\*Function body\*/  } |

Example 3

|  |
| --- |
| let inferredTypeVariable = testFn(myVar);  // The variable inferredTypeVariable will have the type returnedCompType. |

1. Enums
   1. Is a complex type that is also one of the most useful
   2. **enum**erates all the possible values that a variable could have

Example

|  |
| --- |
| enum Direction {  North,  South,  East,  West  } |

Example 2

|  |
| --- |
| let whichWayToArcticOcean: Direction;  whichWayToArcticOcean = Direction.North; // No type error.  whichWayToArcticOcean = Direction.Southeast; // Type error: Southeast is not a valid value for the Direction enum.  whichWayToArcticOcean = West; // Wrong syntax, we must use Direction.West instead. |

* 1. We can also assign it with value
     1. By default, Enum values are assigned a numerical value according to their listed order

|  |
| --- |
| enum Direction {  North = 8,  South = 2,  East = 6,  West = 4  } |

Exercise

|  |
| --- |
| enum Pet {  Hamster,  Rat,  Chinchilla,  Tarantula  };  let petOnSale = 'chinchilla';  let ordersArray = [  ['rat', 2],  ['chinchilla', 1],  ['hamster', 2],  ['chinchilla', 50]  ];  // Write your code below:  let petOnSaleTS: Pet = Pet.Chinchilla;  let ordersArrayTS: [Pet, number][] = [  [Pet.Rat, 2],  [Pet.Chinchilla, 1],  [Pet.Hamster, 2],  [Pet.Chinchilla, 50]  ];  ordersArrayTS.push([Pet.Jerboa, 3]); |

1. String Enums vs. Numeric Enums
   1. Typescript also allows enums to be based on strings
   2. Numeric enums, the numbers could be assigned automatically
   3. String enums we must write the string explicitly
   4. **We recommend to always use string enums because numeric enums allow for some behaviors that can let bugs sneak into our code**
      1. String enums are much more strict

Example

|  |
| --- |
| enum DirectionNumber { North, South, East, West }  enum DirectionString { North = 'NORTH', South = 'SOUTH', East = 'EAST', West = 'WEST' } |

Example 2

|  |
| --- |
| let whichWayToAntarctica: DirectionString;  whichWayToAntarctica = '\ (•◡•) / Arbitrary String \ (•◡•) /'; // Type error!  whichWayToAntarctica = 'SOUTH'; // STILL a type error!  whichWayToAntarctica = DirectionString.South; // The only allowable way to do this. |

Exercise

|  |
| --- |
| let petOnSale = 'chinchilla';  enum Pet {  Hamster = "HAMSTER",  Rat = "RAT",  Chinchilla = "CHINCHILLA",  Tarantula = "TARANTULA"  };  let petOnSaleTS: Pet = Pet.Chinchilla;  let ordersArrayTS: [Pet, number][] = [  [Pet.Rat, 2],  [Pet.Chinchilla, 1],  [Pet.Hamster, 2],  [Pet.Chinchilla, 50]  ];  // Write your code below:  ordersArrayTS.push(['HAMSTER', 1]); |

1. Object Types
   1. TypeScript’s *object types* are extremely useful, as they allow us extremely fine-level control over variable types in our programs

Example

|  |
| --- |
| let aPerson: {name: string, age: number}; |

Example 2

|  |
| --- |
| aPerson = {name: 'Aisle Nevertell', age: "wouldn't you like to know"}; // Type error: age property has the wrong type.  aPerson = {name: 'Kushim', yearsOld: 5000}; // Type error: no age property.  aPerson = {name: 'User McCodecad', age: 22}; // Valid code. |

* 1. TypeScript places no restrictions on the types of an object’s properties
     1. They can be enums, arrays, and even other object types!

Example 3

|  |
| --- |
| let aCompany: {  companyName: string,  boss: {name: string, age: number},  employees: {name: string, age: number}[],  employeeOfTheMonth: {name: string, age: number},  moneyEarned: number  }; |

Exercise

|  |
| --- |
| function sayHappyBirthdayWithObject(personObject: {  name: string,  age: number,  giftWish: string,  success: boolean  }){  let output ='';  output += 'Happy Birthday '  + personObject.name + '! ';  output += 'You are now '  + personObject.age + ' years old! ';  output += 'Your birthday wish was to receive '  + personObject.giftWish  + '. And guess what? You will ';  if (!personObject.success){  output += 'not ';  }  output += 'receive it! \n';  console.log(output);  }  let birthdayBabies: {  name: string,  age: number,  giftWish: string,  success: boolean  }[] = [  {name: 'Liam', age: 0, giftWish: 'karate skills', success: false},  {name: 'Olivia', age: 0, giftWish: 'a bright future', success:true},  {name: 'Ava', age: 0, giftWish: '$0.25', success:true}  ];  birthdayBabies.forEach(sayHappyBirthdayWithObject); |

1. Type Aliases
   1. Is one great way to customize the types in our programs
   2. Type aliases are truly useful for referring to complicated types that need to be repeated

Example

|  |
| --- |
| type MyString = string;  let myVar: MyString = 'Hi'; // Valid code. |

Example 2

|  |
| --- |
| type Person = {name:string, age:number};  let aCompany: {companyName: string, boss: Person, employees:Person[], employeeOfTheMonth: Person, moneyEarned: number}; |

Exercise

|  |
| --- |
| // Add your type alias below:  type Coord = [number, number, string, number, number, string];  let codecademyCoordinates: Coord = [40, 43.2, 'N', 73, 59.8, 'W'];  let bermudaTCoordinates: Coord = [25, 0 , 'N' , 71, 0, 'W']; |

1. Function Types
2. **Function types are most useful when applied to**[**callback functions**](https://developer.mozilla.org/en-US/docs/Glossary/Callback_function)**.**
3. TypeScript is that we can precisely control the kinds of functions assignable to a variable. We do this using *function types*

Example Syntax

|  |
| --- |
| type StringsToNumberFunction = (arg0: string, arg1: string) => number; |

Example 2

|  |
| --- |
| let myFunc: StringsToNumberFunction;  myFunc = function(firstName: string, lastName: string) {  return firstName.length + lastName.length;  };    myFunc = function(whatever: string, blah: string) {  return whatever.length - blah.length;  };  // Neither of these assignments results in a type error. |

1. IMPORTANT
   1. Never omit the parameter names
   2. Never omit the parenthesis around the parameters

Example

|  |
| --- |
| type StringToNumberFunction = (string)=>number; // NO  type StringToNumberFunction = arg: string=>number; // NO NO NO NO |

Exercise

|  |
| --- |
| // Math Operations  function add(a, b){return a+b }  function subtract(a, b){return a-b }  function multiply(a, b){return a\*b}  function divide(a, b){return a/b}  function wrongAdd(a, b){return (a+b)+''}  // Add your function type below:  type OperatorFunction = (arg0: number, arg1: number) => number;  // Math Tutor Function That Accepts a Callback  function mathTutor(operationCallback: OperatorFunction) {  console.log("Let's learn how to", operationCallback.name,'!');  let value25 = operationCallback(2,5);  console.log('When we', operationCallback.name, '2 and 5, we get', value25, '.');  console.log('When we', operationCallback.name, value25, 'and 7, we get', operationCallback(value25,7), '.');  console.log('Now fill out this worksheet.');  }  // Call your functions below:  mathTutor(multiply);  mathTutor(wrongAdd); |

1. Generic Types
   1. Generics give us the power to define our own collections of object types
      1. In example, we must substitute **T** with some type of our choosing, for example **string**
      2. Then, Family<string> is exactly the same as the object type given by setting T to string: {parents:[string,string], mate:string, children: string[]}.

Example

|  |
| --- |
| type Family<T> = {  parents: [T, T], mate: T, children: T[]  }; |

Example 2

|  |
| --- |
| let aStringFamily: Family<string> = {  parents: ['stern string', 'nice string'],  mate: 'string next door',  children: ['stringy', 'stringo', 'stringina', 'stringolio']  }; |

Exercise

|  |
| --- |
| type Human = {name: string, job: string};  type Dog = {name: string, tailWagSpeed: number};  type Family<T> = {  parents: [T, T], mate: T, children: T[]  };  //Do not change the code above this line.  //Provide type annotations for the variables below:  let theFamily: Family<number> = {  parents: [3, 4], mate: 9, children: [5, 30, 121]  };  let someFamily: Family<boolean> = {  parents: [true, true], mate: false,  children: [false, false, true, true]  };  let aFamily: Family<Human> = {  parents: [  {name: 'Mom', job: 'software engineer'},  {name: 'Dad', job: 'coding engineer'}  ],  mate: {name: 'Matesky', job: 'engineering coder'},  children: [{name: 'Babesky', job: 'none'}]  };  let anotherFamily: Family<Dog> = {  parents: [  {name: 'Momo', tailWagSpeed: 3},  {name: 'Dado', tailWagSpeed: 100}  ],  mate: {name: 'Cheems', tailWagSpeed: 7},  children: [  {name: 'Puppin', tailWagSpeed: 0.001},  {name: 'Puppenaut', tailWagSpeed: 0.0001},  {name: 'Puppenator', tailWagSpeed: 0.01}  ]  }; |

1. Generic Functions
   1. Are easiest to understand via example
      1. **In the following, we run into a problem when we try to specify the function’s return type**
      2. **Generic functions to rescue!!**

Example (Javascript)

|  |
| --- |
| function getFilledArray(value, n) {  return Array(n).fill(value);  } |

Example (Typescript, Generic functions)

|  |
| --- |
| function getFilledArray<T>(value: T, n: number): T[] {  return Array(n).fill(value);  } |

* 1. The above code tells TypeScript to make sure that value and the returned array have the same type T
  2. using getFilledArray<string>('cheese', 3), which sets T equal to string.

Exercise

|  |
| --- |
| function getFilledArray<T>(value: T, n: number): T[] {  return Array(n).fill(value);  }  let stringArray: string[];  let numberArray: number[];  let personArray: {name: string, age: number}[];  let coordinateArray: [number, number][];  // Write your code below:  stringArray = getFilledArray<string>('hi', 6);  numberArray = getFilledArray<number>(9, 6);  personArray = getFilledArray<{name: string, age: number}>({name: 'J. Dean', age: 24}, 6);  coordinateArray = getFilledArray<[number, number]>([3,4], 6); |