Chatper 2 – Functions

1. Introduction
   1. type flexibility often allows functions to be invoked with unexpected argument types
   2. 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 |

Exercise

|  |
| --- |
| 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); |

1. Parameter Type Annotations
   1. parameters may be given type annotations with the same syntax as variable declarations: a colon next to the name

|  |
| --- |
| 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' |

* 1. 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 |

Exercise

|  |
| --- |
| function triple(value: number) {  return value \* 3;  }  function greetTripled(greeting: string, value: number) {  console.log(`${greeting}, ${triple(value)}`!);  }  greetTripled('Hiya', 5); |

1. Optional Parameters
   1. TypeScript normally gives an error if we don’t provide a value for all arguments in a function – Not desirable
   2. Made optional by adding ‘?’ after it’s name
      1. 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! |

Exercise

|  |
| --- |
| function proclaim(status?: string) {  console.log(`I'm ${status || 'not ready...'}`);  }  proclaim();  proclaim('ready?');  proclaim('ready!'); |

1. Default Parameters
   1. If a parameter is assigned a default value, TypeScript will infer the variable type to be the same as the default value’s type.
   2. 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}!`);  } |

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

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

Exercise

|  |
| --- |
| function getRandomNumber(){  return Math.random();  }  let myVar = getRandomNumber(); |

1. Explicit Return Types
2. Achieved by using the same syntax as other type annotations, a colon followed by the type

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

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

Exercise

|  |
| --- |
| //The problematic function is f9()  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()); |

1. Void Return Type
2. it is often preferred to use type annotations for functions, even when those functions don’t return anything

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

Exercise

|  |
| --- |
| function makeFruitSalad(fruit1:string, fruit2:string):void{  let salad=fruit1+fruit2+fruit2+fruit1+fruit2+fruit1+fruit1;  console.log(salad);  }  makeFruitSalad('banana','pineapple'); |

1. Documenting Functions
2. TypeScript recognizes JavaScript comment syntax

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

1. We can use @param to describe each of the function’s parameters, and we can use @return 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;  }  } |

Exercise

|  |
| --- |
| /\*\*  \* Returns ingredients used in fruit salad  \* @param fruit1 - the first fruit  \* @param fruit2 - the second fruit  \* @returns none  \*/  function makeFruitSalad(fruit1: string, fruit2: string): void {  let salad = fruit1 + fruit2 + fruit2 + fruit1 + fruit2+ fruit1 + fruit1;  console.log(salad);  }    /\*\*  \* repeats status repeat many times  \* @param status - status of the user  \* @param repeat - number of times the status to be repeated  \* @returns none  \*/  function proclaim(status = 'not ready...', repeat = 1) {  for (let i = 0; i < repeat; i += 1) {  console.log(`I'm ${status}`);  }  } |

Quiz

1. Fill in the code to declare the correct return type for the arrow function, whatsMyAge().
   1. :, number