TS

Functions

Introduction

- Functions are the fundamental building block of any application in TypeScript
- They’re how you build up layers of abstraction, mimicking classes, information hiding, and modules
- Technically, a function is a block of code which can be reused in your application
- Types
  - Named function
  - Anonymous function

Named Function

- A function declared in TypeScript which has a name
- A function needs to be declared with keyword function
- A function can accept one or more parameters

```
function myFunction() {
  console.log('inside myFunction')
}
```

Anonymous Function

- A function without name is known as anonymous function

```
const myFunction = function () {
  console.log('inside myFunction')
}
```

Lambda Function

- Lambda refers to anonymous functions in programming
- Lambda functions are a concise mechanism to represent anonymous functions. These functions are also called as **Arrow functions**
- There are 3 parts to a Lambda function
  - **Parameters** – A function may optionally have parameters
  - **The fat arrow notation/lambda notation (=>)** – It is also called as the goes to operator
  - **Statements** – represent the function’s instruction set

```
const myFunction = () => {
  console.log('inside myFunction')
}
```

Lambda Function

- Optional parentheses for a single parameter
- Optional braces for a single statement
- Empty parentheses for no parameter

Function Constructor

- TypeScript also supports defining a function with the built-in JavaScript constructor called Function()

```
const add = new Function('p1', 'p2', `return p1 + p2`)
const addition = add(10, 20)
console.log(`addition = ${addition}`)
```

Parameters

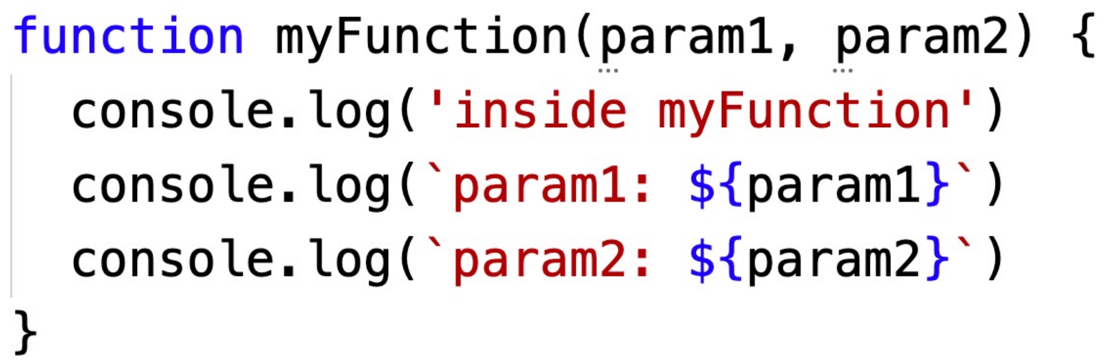
Parameterless Function

- A function without parameter is known as parameter less function

```
function myFunction() {
  console.log('inside myFunction')
}
```

Parameterized function

- A function accepting one or more parameters is known as parameterized function



```
function myFunction(param1, param2) {
  console.log('inside myFunction')
  console.log(`param1: ${param1}`)
  console.log(`param2: ${param2}`)
}
```

Default Parameters

- A function can set default value for parameters
- Such parameters having default value can be optionally passed while making function call
- If value for parameter is passed then, parameter will use that value
- If value for parameter is not passed then, parameter will use the default value

```
function myFunction(param1, param2 = 10) {
  console.log(`param1: ${param1}`)
  console.log(`param2: ${param2}`)
}

// param1: 10, param2: 20
myFunction(10, 20)

// param1: 10, param2: 10
myFunction(10)
```

Rest Parameters

- Rest parameters don’t restrict the number of values that you can pass to a function
- However, the values passed must all be of the same type
- In other words, rest parameters act as placeholders for multiple arguments of the same type
- To declare a rest parameter, the parameter name is prefixed with three periods
- Any nonrest parameter should come before the rest parameter

```
function myFunction(...values) {
  let sum = 0
  for (const value of values) {
    sum += value
  }
  console.log(`sum of all values = ${sum}`)
}

myFunction(10, 20)
myFunction(10, 20, 30, 40, 50)
```

Optional Parameters

- Optional parameters can be used when arguments need not be compulsorily passed
- A parameter can be marked optional by appending a question mark to its name
- The optional parameter should be set as the last argument in a function

```
function myFunction(p1: number, p2?: number) {
  console.log(`p1 = ${p1}`)
  console.log(`p2 = ${p2}`)
}

// p1: 10, p2: 20
myFunction(10, 20)

// p1: 10, p2: undefined
myFunction(10)
```

Advanced Functions

Function Type

- Function type defines type of a function
- A function’s type has the same two parts
    - the type of the arguments
    - the return type
- When writing out the whole function type, both parts are required

```
const myFunction: (x: number, y: number) => number = function (x, y) {
  return x + y
}

console.log(myFunction(10, 20))
```

Function Overloads

- Functions have the capability to operate differently on the basis of the input provided to them
- In other words, a program can have multiple methods with the same name with different implementation
- This mechanism is termed as Function Overloading

Function Overloads

- A function can be overloaded by providing same name but different function signature
  - **The data type of the parameter**
    - function myFunction(string) : void
    - function myFunction(number) : void
  - **The number of parameters**
    - function myFunction(string) : void
    - function myFunction(string, string) : void
  - **The sequence of parameters**
    - function myFunction(string, number) : void
    - function myFunction(number, string) : void

Function Alias

- Another name given to an existing function
- Similar to function pointer
- With function alias, one function can be invoked by multiple names

```
function function1() {
  console.log('inside myFunction')
}

// function alias
const myFunction = function1

// both will call same function
function1()
myFunction()
```

Functional Programming

Introduction

- Functional programming is a programming paradigm - a style of building the structure and elements of computer programs - that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data
- Hence in functional programming, there are two very important rules
  - **No Data mutations**
    - It means a data object should not be changed after it is created
  - **No implicit state**
    - Hidden/Implicit state should be avoided. In functional programming state is not eliminated, instead, its made visible and explicit
- This means:
  - **No side effects**
    - A function or operation should not change any state outside of its functional scope. I.e, A function should only return a value to the invoker and should not affect any external state. This means programs are easier to understand
  - **Pure functions only**
    - Functional code is idempotent. A function should return values only based on the arguments passed and should not affect(side-effect) or depend on global state. Such functions always produce the same result for the same arguments

First-class and higher-order functions

- First-class functions(function as a first-class citizen) means you can assign functions to variables, pass a function as an argument to another function or return a function from another
- TypeScript supports this and hence makes concepts like closures, currying, and higher-order-functions easy to write
- A function can be considered as a higher-order-function only if it takes one or more functions as parameters or if it returns another function as a result

```
const addition = (x, y) => x + y
const subtraction = (x, y) => x - y

function execute(func: (x, y) => number) {
  console.log(`result = ${func(10, 20)}`)
}

// result = 30
execute(addition)

// result = -10
execute(subtraction)
```

Closures and Currying

- There are also many built-in declarative higher-order-functions in TypeScript like map, reduce, forEach, filter etc.

```
// this is a higher-order-function that returns a function
function add(x: number): (y: number) => number {
  // A function is returned here as closure
  // variable x is obtained from the outer scope
  // of this method and memorized in the closure
  return (y: number): number => x + y;
}

// we are currying the add method to create more variations
var add10 = add(10);
var add20 = add(20);
var add30 = add(30);

console.log(add10(5)); // 15
console.log(add20(5)); // 25
console.log(add30(5)); // 35
```

Pure functions

- A pure function should return values only based on the arguments passed and should not affect or depend on global state

```
// pure function
function sum(a: number, b: number): number {
  return a + b;
}
```

Lazy evaluation

- Lazy evaluation or non-strict evaluation is the process of delaying evaluation of an expression until it is needed
- In general, TypeScript does strict/eager evaluation but for operands like &&, || and ?: it does a lazy evaluation.

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

function multiply(x: number, y: number): number {
  return x * y;
}

function addOrMultiply(add: boolean, onAdd: number, onMultiply: number): number {
  return add ? onAdd : onMultiply;
}

console.log(addOrMultiply(true, add(10, 20), multiply(10, 20)))
console.log(addOrMultiply(false, add(10, 20), multiply(10, 20)))
```

Array

map()

- The **map()** method creates a new array with the results of calling a provided function on every element in the calling array

```
const numbers = [1, 2, 3, 4, 5]
const squares = numbers.map(x => x ** 2)

// [ 1, 4, 9, 16, 25 ]
console.log(squares)
```

filter()

- The **filter()** method creates a new array with all elements that pass the test implemented by the provided function

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const oddNumbers = numbers.filter(x => x % 2 == 0)

// [ 2, 4, 6, 8, 10 ]
console.log(oddNumbers)
```

reduce()

- The **reduce()** method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
const sum = numbers.reduce((x, y) => x + y)

// 55
console.log(sum)
```

forEach()

- **forEach()** is used to iterate an array with a function

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numbers.forEach(value => {
  console.log(`value = ${value}`)
})
```