



Monash
Association
of Coding

Functional Programming in TypeScript

MAC x LEARN

About



Lauren Yim

Projects Officer @ MAC

Bachelor of CS



Setup

Installing locally

- Install Node.js: <https://nodejs.org>
- Clone <https://github.com/monashcoding/fp-workshop>
- Run `npm install`

Using CodeSandbox (online)

- <https://codesandbox.io/p/devbox/mac-fp-workshop-r5ydkq>
 - This is also linked in the GitHub link above
- Sign in and fork the devbox
- Run `npm install`



What is functional programming?

—

- Declarative programming paradigm based on applying and composing functions
 - *Declarative* programming: specify **what** you want the program to do, rather than exactly **how** it should do it (imperative)
- Usually synonymous with '*pure* functional programming', which also tries to minimise side effects (e.g. printing to the console) where possible
- Code can be a lot shorter and easier to read and understand



What is functional programming?

Imperative (C++)

```
vector<int> numbers{2, 4, 3, 1, 6, 10, 5};
vector<int> result;
for (int i = 0; i < numbers.size(); i++) {
    int x = numbers[i] * 3;
    if (x % 2 == 0) {
        result.push_back(x);
    }
}
int sumOfResult = 0;
for (int i = 0; i <= result.size(); i++) {
    sumOfResult += result[i];
}
```

Functional (TypeScript)

```
const numbers = [2, 4, 3, 1, 6, 10, 5]
const result = numbers
    .map(x => x * 3)
    .filter(x => x % 2 === 0)

const sum = (xs: number[]): number =>
    xs.reduce((acc, x) => acc + x, 0)

const sumOfResult = sum(result)
```



What is functional programming?

Imperative (C++)

```
vector<int> numbers{2, 4, 3, 1, 6, 10, 5};
vector<int> result;
for (int i = 0; i < numbers.size(); i++) {
    int x = numbers[i] * 3;
    if (x % 2 == 0) {
        result.push_back(x);
    }
}
int sumOfResult = 0;
for (int i = 0; i <= result.size(); i++) {
    sumOfResult += result[i];
}
```

Functional (TypeScript)

```
const numbers = [2, 4, 3, 1, 6, 10, 5]
const result = numbers
    .map(x => x * 3)
    .filter(x => x % 2 === 0)

const sum = (xs: number[]): number =>
    xs.reduce((acc, x) => acc + x, 0)

const sumOfResult = sum(result)
```



What is functional programming?

Imperative (C++)

```
vector<int> numbers{2, 4, 3, 1, 6, 10, 5};
vector<int> result;
for (int i = 0; i < numbers.size(); i++) {
    int x = numbers[i] * 3;
    if (x % 2 == 0) {
        result.push_back(x);
    }
}
int sumOfResult = 0;
for (int i = 0; i < result.size(); i++) {
    sumOfResult += result[i];
}
```

Functional (TypeScript)

```
const numbers = [2, 4, 3, 1, 6, 10, 5]
const result = numbers
    .map(x => x * 3)
    .filter(x => x % 2 === 0)

const sum = (xs: number[]): number =>
    xs.reduce((acc, x) => acc + x, 0)

const sumOfResult = sum(result)
```



What is functional programming?

Imperative (Python)

```
def partition(array: list[int], low: int, high: int) -> int:
    pivot = low
    for i in range(low + 1, high + 1):
        if array[i] <= array[low]:
            pivot += 1
            swap(array, i, pivot)
            swap(array, low, pivot)
    return pivot

def quicksort_aux(array: list[int], low: int, high: int) -> None:
    if low >= high: return
    pivot = partition(array, low, high)
    quicksort_aux(array, low, pivot - 1)
    quicksort_aux(array, pivot + 1, high)

def quicksort(array: list[int]) -> None:
    quicksort_aux(array, 0, len(array) - 1)
```

Functional (TypeScript)

```
const partition = <T>(
    xs: T[],
    fn: (x: T) => boolean
): [T[], T[]] =>
    [xs.filter(fn), xs.filter(x => !fn(x))]

const quicksort = (array: number[]): number[] => {
    if (array.length === 0) return []
    const [pivot, ...rest] = array
    const [lt, gt] = partition(pivot, x => x < pivot)
    return [...quicksort(lt), pivot, ...quicksort(gt)]
}
```



What is functional programming?

Imperative (C++)

```
int result = 0;
int count = 0;
int a = 0;
int b = 1;
while (count < 10) {
    if (a % 2 != 0) {
        result += b;
        count++;
    }
    int tmp = a + b;
    a = b;
    b = tmp;
}
```

Functional (Haskell)

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

result :: Int
result = sum (take 10 (filter odd fibs))
```



What is functional programming?

Imperative (C++)

```
int result = 0;
int count = 0;
int a = 0;
int b = 1;
while (count < 10) {
    if (a % 2 != 0) {
        result += b;
        count++;
    }
    int tmp = a + b;
    a = b;
    b = tmp;
}
```

Functional (Haskell)

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

result :: Int
result = sum (take 10 (filter odd fibs))
```



What is functional programming?

Imperative (C++)

```
int result = 0;
int count = 0;
int a = 0;
int b = 1;
while (count < 10) {
    if (a % 2 != 0) {
        result += a;
        count++;
    }
    int tmp = a + b;
    a = b;
    b = tmp;
}
```

Functional (Haskell)

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

result :: Int
result = sum (take 10 (filter odd fibs))
```



What is functional programming?

Functional(ish) (Python)

```
from collections.abc import Iterable
```

```
from itertools import islice
```

```
def fibs() -> Iterable[int]:
```

```
    a = 0
```

```
    b = 1
```

```
    yield a
```

```
    while True:
```

```
        yield b
```

```
        a, b = b, a + b
```

```
result = sum(islice(filter(lambda x: x % 2 != 0, fibs()), 10))
```

Functional (Haskell)

```
fibs :: [Int]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```
result :: Int
```

```
result = sum (take 10 (filter odd fibs))
```



TypeScript

- Adds static typing to JavaScript
- Decent support for functional programming
 - Higher-order functions
 - Anonymous functions
 - Builtin higher-order functions (methods) for arrays
- Functional programming commonly used in TypeScript/JavaScript code, e.g. web development



TypeScript Syntax

```
const message: string = "Hello, world!"  
console.log(message) // prints to console
```

```
// Type annotations are optional
```

```
const someConstant = 123
```

```
// Single quotes also work for strings
```

```
let mutableVariable = 'abc'
```

```
mutableVariable = 'def'
```

```
const array: number[] = [1, 2, 3]
```

```
const object: {name: string, age: number} = {name: 'Lauren', age: 19}
```

```
const objectValue = object.name // or object['name']
```

```
const ternaryOperator = array[0] === 1 ? "it's 1" : "it's not 1"
```

```
// use triple equals to compare  
if (mutableVariable === 'abc') {  
    // do something  
} else if (someConstant > 100) {  
    // do something else  
} else {  
    // do another thing  
}
```



Functions in TypeScript

1. Function declaration

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

type of add:
(x: number, y: number) => number

2. Function expression

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

3. Arrow function (we will be using this one!)

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

// shorthand:

```
const add = (x: number, y: number): number => x + y
```



Higher-Order Functions

- In TypeScript (and many other languages), functions are *first-class*, meaning that they can be treated like any other value. This means you can:
 - assign a function to a variable
 - store functions in arrays and objects
 - use a function as an argument to another function
 - return a function from a function
- *Higher-order* functions are functions that either
 - accept a function as a parameter
 - return a function



Higher-Order Functions: Array Methods

—

If array is of type `T[]`:

// Applies a function to all the elements of an array and returns the results
// in a new array

`array.map: <U>(fn: (x: T) => U) => U[]`

// Only keeps elements in the list satisfying a predicate

`array.filter: (fn: (x: T) => boolean) => T[]`

`[1, 2, 3, 4, 5, 6].map(x => x * 2)` // `[2, 4, 6, 8, 10, 12]`

`[1, 2, 3, 4, 5, 6].filter(x => x % 3 == 0)` // `[3, 6]`



Higher-Order Functions: reduce

—
If array is of type T[]:

```
// Applies a function to each element of the list  
// (from left to right) and an accumulator value and  
// returns the final accumulator value
```

```
array.reduce: <U>(fn: (acc: U, x: T) => U, init: U) => U
```

```
[1, 2, 3, 4].reduce((acc, x) => acc + x, 0) // 10
```

x	acc
	0
1	1
2	3
3	6
4	10



Higher-Order Functions: Currying

—

- Instead of a function receiving multiple parameters, it can receive a single parameter and return a new function
- Helpful for reusing functions

```
const add = (x: number) => (y: number): number => x + y
const five = add(3)(2)
```

```
const startsWith = (prefix: string) => (string: string): boolean =>
  string.startsWith(prefix)
```

```
const units = ['FIT2004', 'FIT3139', 'MTH2025', 'MTH2141']
const itUnits = units.filter(startsWith('FIT'))
const mathsUnits = units.filter(startsWith('MTH'))
```



Immutability

```
const modifyTable1 = (table: number[][]): void => {  
  let i = 0  
  while (i < table.length) {  
    let sum = 0  
    for (const num of table[i]) sum += num  
    if (sum === 0) table.splice(i, 1) // remove from table at index i  
    else i += 1  
  }  
}
```




```
const modifyTable2 = (table: number[][]): number[][] =>
    table.filter(row => row.reduce((acc, x) => acc + x, 0) !== 0)
//          sum of row: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```



Immutability

- Avoiding mutating things (variables, arrays, etc.) makes code easier to reason about
- In TypeScript: `const` prevents the variable from being reassigned, but doesn't guarantee e.g. the elements of an array won't be modified
 - TypeScript is happy with this code:
 - `const array = [1, 2, 3]; array.push(4)`
- Using the `readonly` array type ensures this doesn't happen
 - TypeScript will complain about this:
 - `const array: readonly number[] = [1, 2, 3]; array.push(4)`



Exercises

Instructions are in the README.md file.

Installing locally

- Install Node.js: <https://nodejs.org>
- Clone <https://github.com/monashcoding/fp-workshop>
- Run `npm install`

Using CodeSandbox (online)

- <https://codesandbox.io/p/devbox/mac-fp-workshop-r5ydkq>
 - This is also linked in the GitHub link above
- Sign in and fork the devbox
- Run `npm install`



Learning More & Resources

- JavaScript/TypeScript libraries: Ramda.js, fp-ts...
- Functional programming languages: Haskell, Elm, Clojure, F#...
- Mostly Adequate Guide to Functional Programming (JavaScript)
- FIT2102: Programming Paradigms

