SWA1, Session 7

Functional Programming

Objects vs "objects"

Object

Encapsulation

Behavior

```
class Account {
  private owner: Customer
  private balance: number

  constructor(...) { ... }

  balance(): number { return balance }
  withdraw(...) {...}
  deposit(...) {...}
}
```

C struct/Record/Associative array / Data

```
const account = {
  owner: {
    name: "...",
    cpr: "..."
  },
  balance: 234876
}
```

- technically an object

Procedure vs function

Procedure

```
function printSum(ns: number[]): void {
  let sum = 0
  for(let n of ns) {
    sum += n
  }
  console.log(sum)
}
```

Function functional/ programming

```
function sum(ns: number[]): number {
  let sum = 0
  for(let n of ns) {
    sum += n
  }
  return n
}
```

Functional programming

Object-oriented programming

- Data is *encapsulated*
- Data can be *mutated* (i.e. changed) through *well-chosen* methods
- Methods use changing local *variables* and *loops*
- Unit tests test *objects*
- A program consists of collaborating objects
  - A *simulation*
- Difficulty at sub-system boundaries (network, DB)
- Regresses to unencapsulated data

Functional programming

- Data is freely accessible
- Data is *immutable*
- Functions return *new* data
- Functions *call* other functions or use *recursion*
- Unit tests test *functions*
- A program consists of function calling functions
  - A *computation*
- Difficulty at system boundaries (GUI, DB)
- Regresses to (few) mutable variables

The promise of functional programming

- Parallelizable
- Testable
- Composable
- Easier to write correct code ?
- Easier to read (at least to verify that the code is correct) ??
- Shorter

The problem of functional programming

- Real life isn't functional
- The problem isn't functional
- The solution isn't functional
  - UI
  - Databases
  - Network
- The CPU isn't functional

  fp is slower
  Not on GPU, though
- Steep learning curve

Pure Functions

- Represents a mapping from input to output and <u>nothing else</u>
- Mathematical functions
- 2 Requirements:
  - Output only depends on input
  - Doesn’t change the state in any way
- Functional programming: All functions are pure
- Generally recommended: Only
  - Procedures
  - Pure functions

Impure functions

Changes environment

```
function sum(ns: number[]): number {
  let sum = 0
  for(let n of ns) {
    sum += n
  }
  console.log(sum)
  return sum
}
```

or; updates DB

Changes the input

```
function sum(ns: number[]): number {
  let sum = 0
  while(ns.length > 0) {
    sum += ns[0]
    ns.splice(0, 1)
  }
  return sum
}
```

Destructive

ns is empty now

Pure functions (I)

Imperative style

```
function sum(ns: number[]): number {
  let sum = 0
  for(let n of ns) {
    sum += n
  }
  return sum
}
```

Functional style

```
function sum(ns: number[]): number {
  return ns.reduce((sum, n) => sum + n, 0)
}
```

Pure functions (II)

Imperative style

```
function range(to: number): number[] {
  const result: number[] = []
  for(let i = 0; i < to; i++) {
    result.push(i)
  }
  return result
}
```

Functional style

```
function range(to: number): number[] {
  if (to <= 0)
    return []
  else
    return [...range(to - 1), to]
}
// Or
function range(to: number): number[] {
  return Array.from(new Array(to), (_, i) => i)
}
```

Recursion vs Utility functions

$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 6 \times 5!$

Recursion

```
function factorial(n: number): number {
  if (n === 0)
    return 1
  else
    return n * factorial(n - 1)
}
```

Utility functions

```
function factorial(n: number): number {
  return range(n)
    .map(i => i + 1)
    .reduce((fac, i) => fac * i, 1)
}
```

6

creating

2 arrays

0, 1, ..., 5

1, 2, ..., 6

The function type

- The function

```
function F(x: number, y: string):
     { name: string, age: number}
{   … }
```

- Has type

```
(x: number, y: string) => { name: string, age: number}
```

- The function

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

// JS:
x => y => x + y

- Has type

```
(x: number) => (y: number) => number
```

Higher-order functions

- Definition: A function that
  - takes a function as an argument
  - or returns a function as a result (or both)

- Use build-in higher-order functions and methods (map, filter, reduce, find ~~slice~~ on arrays)
  - Or: Use lodash, ramda, underscore, immutable.js …

- Code: hof.ts

Immutability

TypeScript

```
type Account = Readonly<
  {
    owner: Customer
    balance: number
  }
>

function withdraw(amt: number,
                  act: Account): Account
{
  ...
}
```

JavaScript

```
const account = Object.freeze({
  owner: Object.freeze({
    name: "...",
    cpr: "..."
  }),
  balance: 234876
})

function withdraw(acc, amt) { ... }
```

Note the argument order

- OO:
  - `account.withdraw(1000)`
- Traditional functional:
  - `withdraw(1000, account)` // account = withdraw(1000, acct)
- Allows this:
  - `const withdraw1000 = withdraw.bind(null, 1000)`
    `withdraw1000(account)`

    Partial application
- We'll use this next week

Immutable objects

- A hybrid between OO and functional
- Made with classes/objects containing methods
- Doesn't mutate the object
- Instead returns a new object

Immutable Object

```
class Account {
  private readonly owner: Customer
  private readonly balance: number

  constructor(...) { ... }

  balance(): number { return balance }
  withdraw(...): Account {...}
  deposit(...): Account {...}
}
```

Example

- mutable.js - classic OO style
- immutable.js - immutable object style
- functional.js - functional programming

Practical functional programming

- Read-only types
- Many, small functions ~ 5-6 lines
- Thinking in pipelines of operations
- Libraries to make operations easier (next week)
- Step-wise refinement to switch from imperative to functional style
- State management
- Side-effect management

Imperative origin

[{slot: '1', score: 4}, ..., {'pair', score: 6}]

```
const scores: any = {}
for(let {slot, score} of player_scores) {
  const number_slot = parseInt(slot)
  if (isDieValue(number_slot)) {
    if (typeof score !== 'number') continue
    scores[number_slot] = score
  }
}
return scores
```

{'1': 4, ..., 6: 24}

Working with arrays

```
const scores: any = []
for(let {slot, score} of player_scores) {
  const number_slot = parseInt(slot)
  if (isDieValue(number_slot)) {
    if (typeof score !== 'number') continue
    scores.push([number_slot, score])
  }
}
return Object.fromEntries(scores)
```

Pre-processing: parseInt

```
  const number_player_scores = player_scores.map(({slot, score}) => ({slot: parseInt(slot), score}))
  const scores: any = []
  for(let {slot, score} of number_player_scores) {
    if (isDieValue(slot)) {
      if (typeof score !== 'number') continue
      scores.push([slot, score])
    }
  }
  return Object.fromEntries(scores)
```

Filtering the relevant slots

```
  const number_player_scores = player_scores.map(({slot,
score}) => ({slot: parseInt(slot), score}))
  const die_value_scores = number_player_scores.filter(s
=> isDieValue(s.slot))
  const scores: any = []
  for(let {slot, score} of die_value_scores) {
    if (typeof score !== 'number') continue
    scores.push([slot, score])
  }
  return Object.fromEntries(scores)
```

Filtering out invalid scores

```
  const number_player_scores = player_scores.map(({slot, score}) => ({slot: parseInt(slot), score}))
  const die_value_scores = number_player_scores.filter(s => isDieValue(s.slot))
  const valid_scores = die_value_scores.filter(s => typeof s.score === 'number')
  const scores: any = []
  for(let {slot, score} of valid_scores) {
    scores.push([slot, score])
  }
  return Object.fromEntries(scores)
```

Recognizing the map pattern

```
function upper_section_scores(player_scores: { slot: string; score: number | null }[]): any {
  const number_player_scores = player_scores.map(({slot, score}) => ({slot: parseInt(slot), score}))
  const die_value_scores = number_player_scores.filter(score => isDieValue(score.slot))
  const valid_scores = die_value_scores.filter(score => typeof score.score === 'number')
  const scores = valid_scores.map(({slot, score}) => [slot, score])
  return Object.fromEntries(scores)
}
```

Pipelining

```
const scores = player_scores
    .map(({ slot, score }) => ({ slot: 
parseInt(slot), score }))
    .filter(score => isDieValue(score.slot))
    .filter(score => typeof score.score === 
'number')
    .map(({slot, score}) => [slot, score])
  return Object.fromEntries(scores)
```

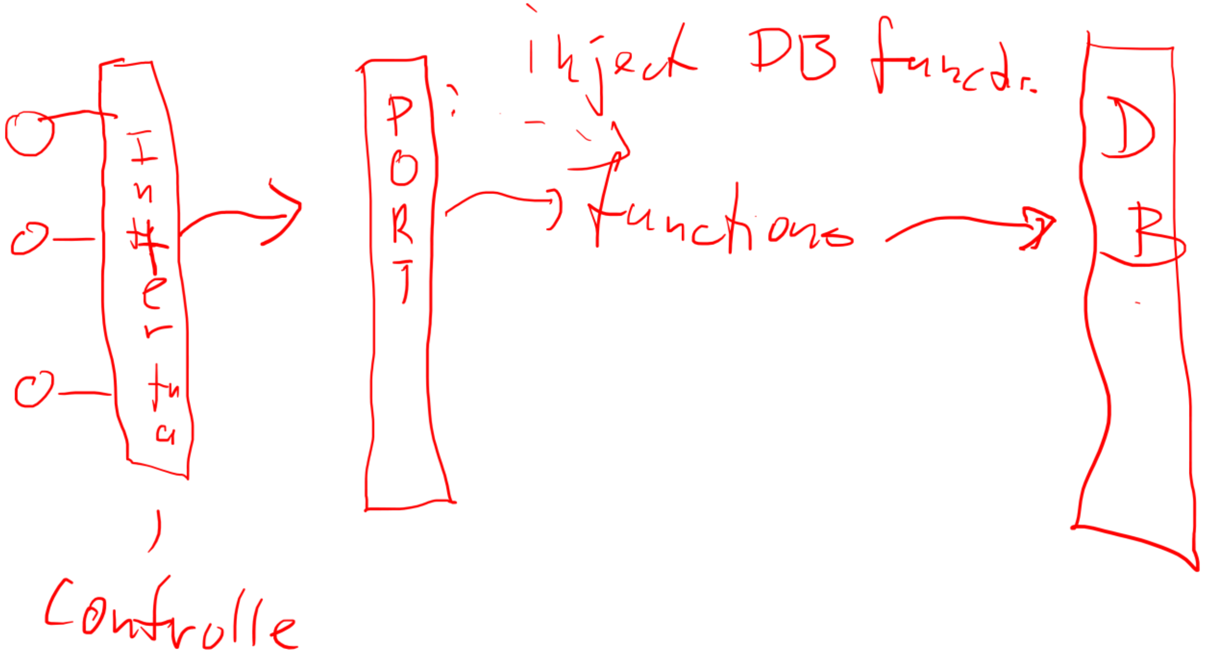
State

- State exists in (almost) all apps
- State is mutable
- So:
  - Have a `State` type (immutable)
  - Have a `state` variable
    - `let state: State = ...`
  - Imagine input (e.g. user input) `name, address, email`
  - Make a function to compute new state:
    - `function updateUser(name, address, email, state: State): State`
  - Update state like
    - `state = updateUser(name, address, email, state)`

Side-effects (DB, Call service, ...)

- Sandwich model
  - <read data>
    <compute functionally>] ~ pure functions } impure
    <write resulting data>
- Functional dependency injection
  - f: (impure1, impure2, impure3) => (x, y, z) => result
- Code: injection.ts

Injection architecture



Sandwich Model Architecture