Peter the Great
Saint-Petersburg Polytechnic University

Ocaml Функциональное программирование

30/10/2025

Презентацию подготовили:
- Дрекалов Никита
- Соколов Артём

POLYTECH

План презентации

- В чем суть подхода?
  - Каррирование
    - Вспоминаем, что такое
    - Реализовываем на JS
    - Пример полезного использования
  - Функции высшего порядка
    - Что и зачем?
    - Пример использования
  - Иммутабельность
    - Полезное свойство данных
    - Зачем?

Функциональное программирование

“Программа = композиция функций”

- First-Class Functions
  - Функцию можно поместить в переменную
  - “Функции высшего порядка”
- Pure Functions
  - Не изменяет внешние состояния
- Declarative Style
  - Описывай не “как?”, а “что?”
- Immutability

Каррирование

(снова)

Каррирование (1/4)

```
(* рассмотрим базовую функцию инкремента *)

let increment x = x + 1
(* val increment : int -> int = <fun> *)

let add a b = a + b
(* val add : int -> int -> int = <fun> *)
```

Каррирование (2/4)

```
(* А что если... *)

let pre_result = add 4
(* val result : int -> int = <fun> *)

let result = pre_result 5
```

Каррирование (3/4)

```
(* Как бы это выглядело на js *)

let add = (x) => {
    return (y) => {
        return (z) => {
            return x + y + z;
        }
    }
}
```

```
(* add func call *)

add(5)(6)(7) // 18
```


POLYTECH

Каррирование (4/4)

```
(* Пример использования *)

let super_math_func log_context func x y =
    Printf.printf "[%s] " log_context;
    let result = func x y in
    Printf.printf "Result: %d\n" result;
    result


let add = (super_math_func "Debugging sum") (+)
let result = add 5 6
```

Функции общего порядка

Функции высшего порядка (1/3)

- `Функции высшего порядка` - это функции, которые:
  - принимают функци(и/ю) в качестве аргуметр(а/ов)
  - возвращают функци(и/ю)

```
let pow2 x = x * x
let double_f f x = f (f x)

let result = double_f pow2 2 (* pow2 -> pow4 *)
```

Функции высшего порядка (2/3)

```
let give_me_flex () = "flex"

let _function = give_me_flex    (* val _function : unit -> string = <fun> *)
let _result   = give_me_flex () (* val _result : string = "flex" *)
```

Пример (3/3)

```
type user = { name: string; age: int; active: bool; }

let users = [
    { name = "Nikita"; age = 22; active = false };
    { name = "Artyom"; age = 22; active = true  };
    { name = "Artur";  age = 17; active = true  };
]

let process_users user_list filter_func map_func =
    List.filter filter_func user_list
    |> List.map map_func

let result = process_users
    users
    (fun usr -> usr.active)
    (fun usr -> { usr with name = usr.name ^ " is cool user" })
```

Иммутабельность

Иммутабельность

- Иммутабельность – это свойство данных, которое означает, что once созданные, они не могут быть изменены. Вместо изменения существующих данных создаются новые.

```
type user = { name: string; age: int }

let user1 = { name = "Alice"; age = 25 }
(* user1 теперь неизменяем - нельзя сделать user1.age = 26 *)

(* Вместо этого создаем нового пользователя: *)
let user2 = { user1 with age = 26 }
(* user1 остался {name = "Alice"; age = 25} *)
(* user2 стал {name = "Alice"; age = 26} *)
```

Зачем?

- Предсказуемость кода
- Сложно “выстрелить себе в колено”
- Безопасность в многопоточности

Спасибо за внимание!