Формат мероприятия

Не совсем верно называть это воркшопом, потому что мы не будем программировать (но это не точно).

Цель мероприятия, познакомить с технологией. Показать, что это не марсианские письмена, а такой же прагматичный язык программирования как и мейнстримные, который должен решать проблемы бизнеса и помогать разработчику.

Главная проблема обучения - нежелание обучаемого воспринимать информацию. Главная цель настоящего обучения - развить интерес к теме, и ответить на главные вопросы.

Зачем вам это нужно?

Попробуем ответить на неприятные вопросы:

Эта технология даст вам карьерное преимущество здесь и сейчас?

Нет

Эта технология является популярной? На ней легко найти работу?

Нет

Зачем вообще тратить на нее свое время?

Потому что Clojure способна подарить кардинально новый опыт решения обыденных проблем. Не нужно смотреть на этот язык, как на потенциально новый вид деятельности. Нужно дать ему шанс, существовать в вашей карьере параллельно с основным источником дохода.

Введение

Lisp

Семейство языков программирования, программы и данные в которых представляются системами линейных списков символов.

```
(car '(A B C D))
;; => A

(write-line "Hello World")
;; => nil

(* 2 3)
;; => 6
```

Clojure

Современный диалект Лиспа, язык программирования общего назначения с поддержкой разработки в интерактивном режиме, поощряющий функциональное программирование и упрощающий поддержку многопоточности. Clojure работает на платформах JVM и CLR.

Синтаксис и его отсутствие

Функция сложения

Рассмотрим функцию сложения двух чисел. Например в языке Java есть синтаксические правила, объявления функции. Объявляем:

- Тип данных, результата
- Имя функции
- Аргументы
- Тело функции

```
int plus (x, y) {
    return x + y;
}
```

В Clojure мы делаем почти то же самое, но опускаем тип результата, потому что язык динамически типизированный. Через функцию* defn Объявляем:

- Имя функции

- Аргументы
- Тело функции

```
(defn plus [x y] (+ x y))
```

Объявление переменных

Объявляем для переменной:

- Тип
- Имя
- Значение

```
byte b = 216;
short s = 1123;
int i = 64536;
long l = 2147483648L;
float pi = 3.14f;
char a = 'a';
String a2 = "Hello";
```

Делаем в Clojure то же самое Через функцию* def

```
(def b 216)
(def s 1123)
(def i 64536)
(def l 2147483648L)
(def a \a)
(def a2 "Hello")
```

Условные выражения

Рассмотрим выражение, которое в зависимости от условия выводит что-то в консоль

Выражение состоит из:

- Условия
- Тела условия

```
if(true) {
    System.out.println("True");
} else {
    System.out.println("False");
}
```

Выражение на Clojure почти полностью соответствует

```
(if true
  (println "True")
  (println "False"))

;; NOTE: или если мы хотим немного оптимизировать код
(println (if true "True" "False"))
```

Поговорим про синтаксис

В Clojure почти все подчиняется одному синтаксическому правилу

- Все что исполняется, называется форма и выглядит как список (...)
- То что в форме на первом месте - функция
- Остальное аргументы

Безусловно есть исключения, в виде

- Обычных макросов
- Структур данных
- Ридеров

Но в контесте данного воркшопа, они нас не интересуют

Структуры данных

Список / List

```
(list 1 2 3 4 5 6)
;;Или
'(1 2 3 4 5 6)
;; => (1 2 3 4 5 6)
```

Вектор / Vector

```
(vector 1 :ke H "s" [1 \n] 4 5 6)
;;Или
[1 :ke \H "s" [1 \n] 4 5 6]
```

Строка / String

```
(def H \H)
(str H \e \l \l \o 1 2 3)
;;Или
"Hello123\n World"
```

Кейворд / Keyword

```
(keyword "hello")
;; Или
:hello
:myns/hello
```

Мапа / HashMap

```
(into {} :a 1 :b 2 :c 3)
;; Или
(def m
  {:a 1
   "b" 2
   1 1
   [1 2] '(1 2 3 4)})

(def r (get m :a))
;; => 1
(get m "b")
;; => 2
(get m 1)
;; => 1
(get m [1 2])
;; => (1 2 3 4)
```

Сет / Set

```
(set [1 2 3 4 3 5 5 5])
;; Или
#{1 3 2 5 4}
```