Язык программирования Elixir

- Конструкции потока управления
- Выполнили Фролов Иван и Ткачев Михаил, гр. 5030102/20202

Условные конструкции

- if / unless: Работают ровно так, как ты ожидаешь. Возвращают значение.

```
age = 18
status = if age >= 18 do
  "совершеннолетний"
else
  "несовершеннолетний"
end

IO.puts(status) # => "совершеннолетний"

# unless - это "если не"
unless age == 18 do
  IO.puts("Тебе не 18!")
end
```

- case: Мощнейшая конструкция для сопоставления с образцом (pattern matching). Заменяет собой цепочки if-else if.

```
result = {:error, "Connection timeout"}

case result do
  {:ok, data} -> IO.puts("Успех: #{data}")
  {:error, reason} -> IO.puts("Ошибка: #{reason}") # Сработает эта ветка
  _ -> IO.puts("Что-то совсем неизвестное") # _ - универсальный образец
end
```

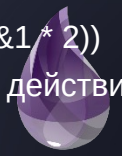
Циклы и Итераторы

- **В Elixir нет циклов** for **и** while **с изменяющимся состоянием.** Вместо этого используется рекурсия и функции высшего порядка.

- **Рекурсия**: Это основной способ итерации.
- *# Факториал через рекурсию*
- defmodule Math do
- def factorial(0), do: 1 # *Граничное условие*
- def factorial(n) when n > 0 do
- n * factorial(n - 1) # *Рекурсивный вызов*
- end
- end

- Enum.map/2: Преобразует каждый элемент коллекции.

- list = [1, 2, 3]
- doubled_list = Enum.map(list, fn x -> x * 2 end)
- IO.inspect(doubled_list) # => *[2, 4, 6]*
- *# Короткий синтаксис через &*
- doubled_list = Enum.map(list, &(&1 * 2))
  - Enum.each/2: Выполняет действие для каждого элемента (возвращает :ok).



Циклы и Итераторы

Списковые включения (for**)**: В Elixir for — это не цикл, а *генератор* списка (comprehension).

```
# Генерируем новый список
squares = for n <- [1, 2, 3, 4], do: n * n
IO.inspect(squares) # => [1, 4, 9, 16]

# С фильтром
evens = for n <- 1..10, rem(n, 2) == 0, do: n
IO.inspect(evens) # => [2, 4, 6, 8, 10]
```

В Elixir нет фигурных скобок {}. Вместо них используются блоки do-end. Это просто синтаксический сахар для ключевых списков. Всё, что начинается с do, — это блок.

```
# Это
if true do
  IO.puts("Hello")
  IO.puts("World")
end

# ...под капотом превращается в это
if true, do: (
  IO.puts("Hello")
  IO.puts("World")
)
```

elixir

Функциональное программирование

Чистые функции и явность

```
# Чистая функция. Всегда для (2, 3) вернет 5.
def add(a, b), do: a + b

# Нечистая функция. Она меняет состояние мира (вывод в консоль).
def greet(name) do
  IO.puts("Hello, #{name}") # Побочный эффект
end
```

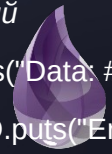
Функции — объекты первого класса

```
# Присваиваем функцию переменной
double = fn x -> x * 2 end
IO.puts(double.(5)) # => 10
```

Сопоставление с образцом (Pattern Matching):

```
# Распаковка кортежа
{:ok, message} = {:ok, "Hello World"}
IO.puts(message) # => "Hello World"

# Используется в аргументах функций
def handle_result({:ok, data}), do: IO.puts("Data: #{data}")
def handle_result({:error, reason}), do: IO.puts("Error: #{reason}")
```



Объектно-ориентированное программирование

Структура данных (вместо класса) — это **Структуры (Structs)**.

```elixir
defmodule User do
  # Определяем структуру с полями по умолчанию
  defstruct name: nil, age: 0
end


# Создаем "экземпляр" структуры
user = %User{name: "Alice", age: 25}
IO.inspect(user) # => %User{age: 25, name: "Alice"}
```

Структура — это просто карта (Map) с заданным набором полей и именем модуля. Она неизменяема.

elixir

Объектно-ориентированное программирование

Инкапсуляция (логика + данные) — это **Модули и Функции**.

Модуль группирует функции, которые работают с определенной структурой данных. Это и есть инкапсуляция.

```
defmodule User do
  defstruct name: "", age: 0
  # "Конструктор"
  def new(name, age) do
    %User{name: name, age: age}
  end
  # Функция, работающая со структурой User
  def is_adult?(%User{age: age}) do
    age >= 18
  end
end
alice = User.new("Alice", 25)
IO.puts(User.is_adult?(alice)) # => true
```

elixir

Объектно-ориентированное программирование

- **Полиморфизм** — достигается через **Протоколы (Protocols)**.

```
# Определяем протокол
defprotocol Greetable do
  def greet(entity)
end
# Реализуем его для нашей структуры User
defimpl Greetable, for: User do
  def greet(%User{name: name}) do
    "Hello, #{name}!"
  end
end

# Реализуем для встроенного типа String
defimpl Greetable, for: String do
```

Объектно-ориентированное программирование

- **Наследование в Elixir: его нет**

- **Почему отказались?**

- **Хрупкость базового класса**: Изменение в родительском классе ломает всех потомков

- Неоднозначность при множественном наследовании

- **Жесткая связность**: Дочерние классы тесно связаны с родителем

Объектно-ориентированное программирование

- **1. Композиция (Composition) —** Собираем функциональность из независимых модулей.

```
defmodule Engine do
  def start, do: "Дзынь-дзынь!"
  def stop, do: "Шшшш..."
end

defmodule Car do
  # Включаем функциональность Engine в Car (композиция)
  defstruct [:engine, model: "Unknown"]

  def start(%Car{engine: engine}) do
    Engine.start(engine)
```

Объектно-ориентированное программирование

- **2. Поведения (Behaviours) — контракты без реализации**
- Похоже на интерфейсы в Java. Задают контракт, но не реализацию.
- *# Определяем поведение (что должно быть реализовано)*
- defmodule Parser do
- @callback parse(String.t) :: {:ok, any()} | {:error, String.t}
- @callback supported_extensions() :: list(String.t)
- end
- *# Реализуем поведение для JSON*
- defmodule JSONParser do
- @behaviour Parser
- def parse(json_string) do
- *# Реальная реализация парсинга JSON*
- {:ok, %{"parsed" => true}}
- end
- def supported_extensions do
- [".json", ".jsonld"]

Объектно-ориентированное программирование

- **3. Макросы use и __using__ — ближайший аналог миксинов**

```
defmodule Loggable do
  # Этот блок выполнится при use Loggable
  defmacro __using__(_opts) do
    quote do
      def log(message) do
        IO.puts("[#{__MODULE__}] #{message}")
      end

      # "Наследуем" функцию по умолчанию
      def debug_mode?, do: false
    end
  end
end

# Используем в другом модуле
```

выводы

- **Поток управления**: вместо циклов - case, cond, рекурсия и Enum.map/2.
- **ФП**: Это основа. Неизменяемость, чистые функции, паттерн-матчинг. Это не причуда, а инструмент для написания надежного кода.
- **ООП**: Его нет в классическом виде. Вместо классов — модули и структуры. Вместо наследования — композиция и протоколы.

Источники

- elixir-lang.org – официальный сайт языка
- Wikipedia – статья Elixir (programming language)