Karazin Scala Users' Group

Course 2023 Autumn

2. Introduction to Course

2. Introduction to course
    2.1. Programming paradigms
    2.2. Theories
    2.3. Imperative programming
    2.4. Functional programming

“Є лише два типи мов програмування: ті, на які всі скаржаться та ті, якими ніхто не користується.”

— Б'ярн Страуструп

2.1 Programming Paradigms

2.1 - Programming Paradigms

Томас Кун і “парадигма” у науково-технічній галузі

2.1 - Programming Paradigms

Роберт Флойд і “парадигма програмування”

“Якщо прогрес мистецтва програмування загалом потребує постійного винаходу та вдосконалення парадигм, то вдосконалення мистецтва окремого програміста вимагає, щоб він розширював свій репертуар парадигм.”

— Роберт Флойд

“Слово «парадигма» використовується у програмуванні для визначення сімейства позначень (нотацій), що розділяють загальний спосіб (методику) реалізацій програм.”

— Діомідіс Спінелліс

"[парадигма програмування - це] стиль програмування як опису намірів програміста"

— Дэниел Бобров

"[парадигма програмування - це] модель або підхід до вирішення проблеми"

— Брюс Шрайвер

"[парадигма програмування - це] підхід до вирішення проблем програмування"

— Лінда Фрідман

"[парадигма програмування - це] способ роздумів про комп'ютерні системи"

— Памела Зейв

“[парадигма програмування – це] правила класифікації мов програмування відповідно до деяких умов, які можуть бути перевірені”

— Пітер Вегнер

“[парадигма програмування - це] спосіб концептуалізації того, що означає "здійснювати обчислення", і як задачі, що підлягають вирішенню на комп'ютері, повинні бути структуровані та організовані”

— Тімоті Бадд

2.1 - Programming Paradigms

- Імперативне програмування
  - процедурне
  - об'єктно-орієнтоване *
- Декларативне програмування
  - логічне
  - функціональне

2.1 - Programming Paradigms

імперативне програмування =

дані + алгоритми

2.1 - Programming Paradigms

функціональне програмування =

функції + функції

2.1 - Programming Paradigms

логічне програмування =

факти + правила

2.1 - Programming Paradigms

об'єктно-орієнтоване програмування =

об'єкти + повідомлення

?

2.2 Imperative programming

“Об'єктно-орієнтоване програмування - це виключно погана ідея, яку могли вигадати тільки в Каліфорнії.”

— Едсгер Дейкстра

2.2 - Imperative programming

Імперативне програмування включає:

- зміна стану змінних;
- використання присвоювання;
- керуючі конструкції, такі як **`if-then-else`**, **`loops`**, **`break`**, **`continue`**, **`return`**.

Найпростіший спосіб зрозуміти концепцію імперативного програмування — це розглянути її з погляду послідовності інструкцій для фон-нейманівської архітектури комп'ютера.

2.2 - Imperative programming

Існує залежність:

змінні	~	комірки пам'яті
розіменування змінних	~	завантаження інструкцій
присвоєння	~	збереження інструкцій
керуючі конструкції	~	jumps

Проблеми: вертикальне масштабування. Як уникнути опису програми, інструкція за інструкцією.

2.2 - Imperative programming

Усі чисті імперативні мови програмування обмежені таким:

Вони прагнуть концептуально описати структури даних
інструкція за інструкцією

Нам потрібні техніки, які дозволяють описувати високорівневі абстракції, такі як колекції, поліноми, геометричні фігури, рядки, документи тощо.

В ідеалі: створення *теорій* колекцій, фігур, рядків тощо.

2.3 Theories

2.3 - Theories

Теорія складається з:

- одного або кількох типів даних;
- операцій з цих типами;
- законів, які описують взаємозв'язок між значеннями та операціями.

Зазвичай теорії припускають жодних мутацій, тобто. змін значень.

2.3 - Theories

Наприклад, теорія поліномів визначає суму двох поліномів таким чином:

$$(a \cdot x + b) + (c \cdot x + d) = (a + c) \cdot x + (b + d)$$

Але теорія не визначає жодного оператора, який би дозволяв змінювати коефіцієнти полінома таким чином, щоб поліном при цьому залишався б тим самим!

2.3 - Theories

Розглянемо код імперативної програми:

```
class Polynomial {
   double[] coefficients;
   ...
}
```

2.3 - Theories

Розглянемо код імперативної програми:

```
class Polynomial {
   double[] coefficients;
   ...
}

Polynomial p = new Polynomial();
p.coefficients[0] = 42;
```

2.3 - Theories

Ще один приклад

Теорія рядків визначає оператор конкатенації ++, який є асоціативним

$$(a \mathbin{++} b) \mathbin{++} c = a \mathbin{++} (b \mathbin{++} c)$$

Але теорія не визначає жодного оператора, який змінював елементи у цій послідовності символів, не змінюючи самої послідовності.

2.3 - Theories

Ще один приклад

Теорія рядків визначає оператор конкатенації ++, який є асоціативним

$$(a \mathbin{++} b) \mathbin{++} c = a \mathbin{++} (b \mathbin{++} c)$$

Але теорія не визначає жодного оператора, який змінював елементи у цій послідовності символів, не змінюючи самої послідовності.

Деякі мови все-таки на правильному шляху, наприклад Java рядки незмінні.

2.3 - Theories

Якщо ми хочемо створити високорівневу концепцію на основі будь-якої математичної теорії, ми не повинні допускати мутацій.

- Теорії не допускають мутацій.
- Мутації можуть "зламати" деякі закони теорії.

Таким чином, нам необхідно

- сконцентруватися на визначенні теорії для вираза операторів через функції
- уникати мутацій
- мати потужні інструменти для опису абстракцій та композиції функцій

2.4 Functional programming

2.2 - Functional programming

- В *узком смысле* под функциональным программированием (FP) понимают программирование без мутирующих переменных, присвоений, циклов и других императивных управляющих конструкций.
- В *широком смысле* под функциональным программированием понимают программирование которое базируется на функциях.
- В частности, функции могут быть значениями, которые создаются, используются и композируются.
- Все это становится легче и нативнее в функциональных языках.

2.4 - Enough to be functional?

```
int Factorial(int n) {
    Log.Info($"Computing factorial of {n}");
    return Enumerable.Range(1, n).Aggregate((x, y) => x * y);
}
```


```
int Factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }

    return result;
}
```

2.4 - Functional programming, another approach

Функціональна програма — це програма, що складається з *чистих* функцій.

2.4 - Functional programming, another approach

Функціональна програма — це програма, що складається з *чистих* функцій.

Функція f є чистою якщо вираз f(x) є прозорим для всіх прозорих посилань x.

2.4 - Functional programming, another approach

Функціональна програма — це програма, що складається з *чистих* функцій.

Функція f є чистою якщо вираз f(x) є прозорим для всіх прозорих посилань x.

Прозорість посилань — властивість, у якому заміна висловлювання на обчислений результат цього виразу не змінює бажаних властивостей програми.

2.4 - Example

```
boolean flag;

int f(int n) {
   int value = 2 * n;

   if(flag) value = 2 * n;
   else value = n;

   flag = !flag;

   return value;
}

void test() {
   flag = true;

   System.out.println("f(1) + f(2) = " + (f(1) + f(2)));
   System.out.println("f(2) + f(1) = " + (f(2) + f(1)));
}
```

2.4 - Functional programming languages

- В *вузькому сенсі* функціональні мови програмування - ті, які не мають змінних, присвоєнь і імперативних керуючих конструкцій.
- В *широкому сенсі* функціональні мови програмування дозволяють конструювати елегантні програми, побудовані із функцій.
- Зокрема, функції в FP — це “first-class citizens”.

Це означає що

- вони можуть бути оголошені будь-де, у тому числі всередині інших функцій;
- як і будь-які інші значення, вони можуть бути передані як параметр у функцію і повернуті як результат;
- як й для інших значень, існує оператор композиції функций.

Pure [edit]

- Agda
- Clean
- Coq (Gallina)
- Cuneiform
- Curry
- DAML
- Elm
- Futhark
- Haskell
- Hope
- Idris
- Joy
- Lean
- Mercury
- Miranda
- PureScript
- Ur
- KRC
- SAC
- SASL
- SequenceL

Impure [edit]

- APL
- ATS
- CAL
- C++ (since C++11)
- C#
- VB.NET
- Ceylon
- D
- Dart
- Curl
- ECMAScript
  - ActionScript
  - ECMAScript for XML
  - JavaScript
  - JScript
  - Source
- Erlang
  - Elixir
  - LFE
  - Gleam
- F#
- Groovy
- Hop
- J
- Java (since version 8)
- Julia
- Kotlin
- Lisp
  - Clojure
  - Common Lisp
  - Dylan
  - Emacs Lisp
  - LFE
  - Little b
  - Logo
  - Scheme
    - Racket (formerly PLT Scheme)
  - Tea
- Mathematica
- ML
  - Standard ML (SML)
    - Alice
  - OCaml
- Nemerle
- Nim
- Opal
- OPS5
- Perl
- PHP
- Python
- Q (equational programming language)
- Q (programming language from Kx Systems)
- R
- Raku
- REBOL
- Red
- Ruby
- REFAL
- Rust
- Scala
- Spreadsheets
- Tcl
- Wolfram Language

2.4 - Short history of FP languages

- 1959 Lisp
- 1975-77 ML, Scheme
- 1978 Smalltalk
- 1986 Standard ML
- 1990 Haskell, Erlang
- 1999 XSLT
- 2000 OCaml
- 2003 Scala, XQuery
- 2005 F#
- 2007 Clojure

2.4 - Pros et cons

2.4 - Pros et cons

Переваги функціонального програмування:

- більш "точна" семантика
- деяка свобода під час виконання (наприклад, паралелізація з допомогою асоціативності);
- виразність;
- параметризація та модульність;
- можливість роботи із нескінченними структурами.

2.4 - Pros et cons

Переваги функціонального програмування:

- більш "точна" семантика
- деяка свобода під час виконання (наприклад, паралелізація з допомогою асоціативності);
- виразність;
- параметризація та модульність;
- можливість роботи із нескінченними структурами.

Недоліки функціонального програмування:

- IO;
- швидкодія (виконання в чужій архітектурі);
- високий поріг входження.

Q&A