СЕКЦІЯ 3. ПРОГРАМНА ІНЖЕНЕРІЯ

КЛАСИ ТИПІВ У МОВІ ПРОГРАМУВАННЯ JAVA / TYPE CLASSES IN JAVA

Заколенко Р.К.., Нагірна А.М. / Zakolenko R.K., Nahirna A.M.
Національний університет «Києво-Могилянська академія» / National University of Kyiv-Mohyla Academy
04655, м. Київ, вулиця Григорія Сковороди, 2, НаУКМА, ФІ, тел. (050) 182-33-61,
E-mail: r.zakolenko@ukma.edu.ua

This paper proposes a mechanism for implementing type classes in Java through annotation processing. Type classes, a powerful feature of functional programming languages, allow adding new behavior to existing types without inheritance or modifying their source code. While Java's type system lacks native support for type classes, we present an approach using a custom @Implicit annotation that enables compile-time resolution of type class instances. The proposed solution supports both simple type classes and higher-kinded types, necessary for abstractions like functors and monads. Our implementation leverages annotation processing to automatically discover and inject appropriate type class instances, eliminating the need for explicit instance passing and reducing boilerplate code. The approach maintains Java's static typing benefits while introducing the extensibility and code reuse capabilities typical of type classes in functional languages.

Мова програмування Java посідає провідне місце серед інструментів розробки програмного забезпечення завдяки своїй надійності та розвиненій екосистемі. Водночас аналіз її системи типів виявляє певні обмеження щодо можливостей абстракції та повторного використання коду, зокрема відсутність вбудованої підтримки класів типів.

Класи типів представляють собою конструкт системи типів, що забезпечує спеціального поліморфізм [1]. Він реалізується через абстрактний параметризований тип, що дозволяє додавати нову поведінку до будь-якого закритого типу даних без використання успадкування на відміну від об'єктно-орієнтованого підходу, де поліморфна поведінка досягається через механізми успадкування та підтипів. Така модель дозволяє абстрагувати спільну поведінку та операції над різними типами даних, зберігаючи при цьому строгу типізацію, можливості для статичної перевірки типів та реалізуючи принцип ретроактивного розширення.

Для ілюстрації концепції розглянемо приклад визначення Monoid мовою програмування Java:

```
interface Monoid<T> { T combine(T a, T b); T empty(); }

class IntegerMonoid implements Monoid<Integer> {
    public Integer combine(Integer a, Integer b) { return a + b; }
    public Integer empty() { return 0;  }
}

class MonoidUtils {
    public static <T> T combineAll(List<T> xs, Monoid<T> monoid) {
        return xs.stream().reduce(monoid.empty(), monoid::combine);
    }
}
```

```
Monoid<Integer> intMonoid = new IntegerMonoid();
List<Integer> numbers = List.of(1, 2, 3);
Integer result = MonoidUtils.combineAll(numbers, intMonoid);
```

Аналіз даного підходу виявляє суттєвий недолік: необхідність явного створення та передачі екземплярів класів-реалізацій у всі методи, де вони використовуються, що призводить до надлишкового шаблонного коду та ускладнює підтримку.

У рамках даного дослідження пропонується механізм розширення мови Java з використанням процесора анотацій та спеціальної анотації @Implicit, який базується на успішному досвіді реалізації подібних механізмів у мові програмування Scala [2]. Дане рішення дозволяє компілятору автоматично знаходити та підставляти відповідні реалізації класів типів під час компіляції. Процесор анотацій сканує код на наявність класів, позначених @Implicit, створює реєстр доступних реалізацій та генерує код для їх автоматичного розв'язання. Коли метод приймає параметр, позначений як @Implicit, компілятор автоматично знаходить відповідну реалізацію у реєстрі на основі типу параметра та підставляє її у місце виклику.

Розглянемо приклад використання такого механізму:

```
@Implicit class IntegerMonoid implements Monoid<Integer> {
    public Integer combine(Integer a, Integer b) { return a + b; }
    public Integer empty() { return 0; }
}

class MonoidUtils {
    public static <T> T combineAll(
        List<T> xs,
        @Implicit Monoid<T> monoid
    ) { return xs.stream().reduce(monoid.empty(), monoid::combine); }
}

List<Integer> numbers = List.of(1, 2, 3);
Integer result = MonoidUtils.combineAll(numbers);
```

Для реалізації більш складних абстракцій, таких як функтори та монади, які абстрагуються над конструкторами типів, необхідна підтримка типів вищого порядку [3]. Типи вищого порядку дозволяють конструкторам типів приймати інші конструктори типів як параметри.

На основі проведеного дослідження можна зробити висновок, що інтеграція класів типів у мову програмування Java через механізм анотацій @Implicit є перспективним напрямком розвитку мови. Подальші дослідження можуть бути спрямовані на реалізацію підтримки типів вищого порядку для класів типів, а також на розробку зручнішого синтаксису для роботи з класами типів через класи розширення. Крім того, перспективним напрямком є дослідження механізмів автоматичного виведення класів типів, які дозволять зменшити обсяг шаблонного коду, наприклад, шляхом автоматичного створення реалізацій для серіалізації чи перевірки еквівалентності.

Література

[1] Wadler, P., & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.

[2] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., ... & Zenger, M. (2004). An overview of the Scala programming language. Technical report.

[3] Yallop, J., & White, L. (2014). Lightweight higher-kinded polymorphism. In International Symposium on Functional and Logic Programming.