DOI: https://doi.org/10.36910/6775-2524-0560-2024-57-10
УДК 004.05
Коломоєць Геннадій Павлович, к.ф.-м.н., доцент,
https://orcid.org/0000-0003-3667-0316
Запорізький національний університет, м. Запоріжжя, Україна

ДОСЛІДЖЕННЯ ТЕХНОЛОГІЇ ОБРАННЯ ПЕРЕВАНТАЖЕНИХ МЕТОДІВ JAVA З ПОЛІМОРФНИМИ АРГУМЕНТАМИ

Коломоєць Г.П. Дослідження технології обрання перевантажених методів Java з поліморфними аргументами. Виконано дослідження реалізованої у Java технології поліморфізму – обрання на етапі компіляції перевантаженого методу у разі передачі йому як аргументів поліморфних об'єктів, тип яких стає відомим тільки на етапі виконання програми. Продемонстрована на прикладі можливість обрання компілятором та системою виконання Java непередбачуваного розробником метода та запропоноване рішення для детермінованого обрання шляхом приведення типу аргументу до необхідного (дійсного) типу. Наводиться рішення, що дозволяє завдяки використанню разом статичного обрання перевантаженого методу та динамічного обрання перевизначеного методу зменшити кількість перевантажених методів, які мають однакову реалізацію. Наголошується на можливому зниженні якості коду за рахунок великої кількості розгалужень при використанні пропонованого підходу у разі великої кількості перевантажених методів з різними реалізаціями. Як рішення для цього випадку пропонується використання засобів рефлексії Java для визначення дійсного типу аргументу та виклику необхідного перевантаженого метода. Для наведеного прикладу досліджена залежність часу виконання програми від кількості викликів перевантажених методів для рішення без використання рефлексії та з її використанням.
Ключові слова: перевантажений метод, аргумент метода, поліморфний об'єкт, обрання перевантаженого методу, перевизначений метод, точка на площині, точка у просторі, приведення типу, рефлексивний виклик методу.

Kolomoiets H. A study on the resolution of overloaded Java methods with polymorphic arguments. The study of the technology of polymorphism implemented in Java was carried out - the resolution of an overloaded method at the compile-time in the case of passing polymorphic objects as arguments to it, the type of which becomes known only at run-time. The possibility of the compiler and the Java runtime execution choosing a method not foreseen by the developer is demonstrated on an example, and a solution is proposed for correcting the resolution by type casting of the argument to the required (actual) type. A solution is given, which allows to reduce the number of overloaded methods with the same implementation thanks to the combined use of static resolution of an overloaded method and dynamic selection of an overridden method. Emphasis is placed on the possible decrease in code quality due to a large number of branches when using the proposed solution in the case of a large number of overloaded methods with different implementations. As a solution for this case, it is proposed to use Java reflection tools to determine the actual type of the argument and call the necessary overloaded method. For the given example, the dependence of the program execution time on the number of overloaded methods invokes was investigated for the solution without the use of reflection and with its use.
Keywords: overloaded method, method argument, polymorphic object, overloaded method resolution, overridden method, point on plane, point in space, type casting, reflexive method call.

Постановка проблеми. Перевантаження методів є однією з технологій поліморфізму і дозволяє використовувати різні реалізації однойменного методу для різних наборів аргументів, що сприяє гнучкості та зрозумілості коду [1]. Водночас наявність декількох реалізацій методу може призвести до невизначеності та виклику методу, який важко передбачити [2]. Java Language Specification у розділі 15.12.2 Compile-Time Step 2: Determine Method Signature визначає алгоритм з трьох фаз, за яким виконується обрання методу з декількох варіантів, що підходять для заданих аргументів [3].

Зауважимо, що обрання найбільш сприйнятого для аргументів методу виконується ще на етапі компіляції вихідного коду, тому у разі, якщо такі аргументи є поліморфними об'єктами, тип яких визначається тільки при виконанні програми, обрання правильного методу може стати проблемою. У цій роботі ми зосередимось саме на дослідженні обрання перевантаженого методу у разі використання поліморфних об'єктів як аргументів.

Аналіз останніх публікацій. Перевантаження методів – це технологія, яка дозволяє визначати у межах класу (або ієрархії класів) декілька методів з однаковим ім'ям але різним набором параметрів. При цьому під різним набором параметрів розуміється їх різна кількість та/або різні типи та/або різна послідовність [2]. При цьому тип, що повертається перевантаженим методом до уваги не береться, оскільки, якщо повертається тип з ієрархії класів, більшої, аніж успадкування від java.lang.Object (будемо називати такі типи поліморфними), у якості такого типу можуть бути зазначені різні поліморфні типи, а дійсний тип з'ясовується тільки при виконанні програми (тобто на етапі Run-time). Оскільки обрання перевантаженого методу виконується ще на етапі компіляції програми, ігнорування цією технологією типу, що повертається методом, є зрозумілим.

У разі, якщо набір параметрів перевантажених методів являє собою дані примітивних типів, або типів, для яких відсутня ієрархія класів (звісно, окрім успадкування від java.lang.Object), наприклад, String, технологія перевантаження методів працює передбачено. Але, якщо параметром (або параметрами) перевантажених методів є поліморфні об'єкти, обрання перевантаженого методу може бути неочікуваним, оскільки дійсний тип такого параметру з'ясовується пізніше (на етапі Run-time), аніж виконується обрання перевантаженого методу (на етапі компіляції).

У дискусії на сайті Stack Overflow [4] згадується ця проблема. Автор шукає спосіб правильного визначення поліморфного типу-аргументу перевизначеного методу при великій кількості таких методів. Автор пропонує використання рефлексії з динамічним визначенням та приведенням до необхідного типу аргумента, але не досягає поставленої мети. У обговоренні зазначається, що використання рефлексії є хибним підходом, який не рекомендується розробниками Java, і пропонується обмежитись використанням перевизначення методів замість перевантаження. Але, на наш погляд, цей підхід не завжди відповідає контексту задачі, тому подальші дослідження описаної проблеми залишаються актуальними.

Постановка завдання. Метою цієї роботи було дослідження технології обрання перевантаженого методу у разі, якщо його аргументами є об'єкти поліморфних типів, та визначення шляхів обрання очікуваного методу.

Виклад основного матеріалу. Відтворити проблему обрання перевантаженого методу пропонується для класичної задачі вимірювання відстані між точками на площині та у просторі. Батьківський клас має наступний вигляд:

```
public class Point {

    protected int x;
    protected int y;

    /* Constructors */
    ...

    /* Calculates distance between two Point instances */
    public double distance(Point p) {
        System.out.println("Point.distance(Point)");
        int dx = p.x - this.x;
        int dy = p.y - this.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    /* Calculates distance between this Point instance and Point3D instance
       projection on X0Y plane */
    public double distance(Point3D p)
        System.out.println("Point.distance(Point3D)");
        int dx = p.x - this.x;
        int dy = p.y - this.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    /* toString() implementation */
    ...
}
```

Клас-спадкоємець має наступний вигляд:

```
public class Point3D extends Point {

    private int z;

    /* Constructors */
    ...

    /* Calculates distance between two Point3D instances */
    @Override
    public double distance(Point3D p) {
        System.out.println("Point3D.distance(Point3D)");
        int dx = p.x - this.x;
        int dy = p.y - this.y;
```

```
        int dz = p.z - this.z;
        return Math.sqrt(dx * dx + dy * dy + dz * dz);
    }

    /* Calculates distance between this Point3D instance perspective
projection
       and 2D Point instance */
    @Override
    public double distance(Point p) {
        System.out.println("Point3D.distance(Point)");
        int dx = this.x / z - p.x;
        int dy = this.y / z - p.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    /* toString() implementation */
    ...
}
```

Можна побачити, що кожен клас має перевантажені методи, що визначають відстань як між двома об'єктами поточного класу, так і між об'єктом поточного класу та об'єктом іншого класу ієрархії успадкування. У контексті задачі у разі обчислення відстані між поточним об'єктом – точкою на площині та точкою у просторі, для точки у просторі ігнорується третя координата (`z`), що відповідає проєкції точки на площину X0Y. А у разі обчислення відстані між поточним об'єктом – точкою у просторі та точкою на площині для точки у просторі знаходиться її перспективна двовимірна проєкція, яка імітує, як об'єкти сприймаються на різних глибинах [5]. Також з метою ідентифікації викликів методів до них додані виведення до консолі у вигляді `Поточний-тип.distance(Тип-аргумент)`.

Для моделювання проблеми створимо у Run-time методі `main` масив об'єктів обох класів у послідовності, що передбачає виклики усіх визначених методів:

```
public class Main {

    public static void main(String[] args) {

        Point[] pointArr = {new Point(4, 5), new Point3D(1, 1, 1),
                            new Point3D(2, 5, 9), new Point(1, 1),
                            new Point(4, 5), new Point3D(1, 1, 1)};

        for (int i = 0; i < pointArr.length; i++) {
            System.out.println(pointArr[i]);
        }
        System.out.println("**********");

        for(int i=0; i<pointArr.length-1; i++) {
            System.out.println("Distance between " + pointArr[i]
                    + " and " + pointArr[i+1] + " is: "
                    + pointArr[i].distance(pointArr[i+1]));
        }
    }
}
```

Перший цикл демонструє, що оголошені як `Point`-елементи масиву на етапі Run-time правильно розпізнаються відповідно як об'єкти батьківського або дочірнього класів:

```
Point{x=4, y=5}
Point3D{x=1, y=1, z=1}
Point3D{x=2, y=5, z=9}
Point{x=1, y=1}
Point{x=4, y=5}
Point3D{x=1, y=1, z=1}
```

Але при виконанні другого циклу виникає проблема (виділено жирним шрифтом):

```
Point.distance(Point)
```

Distance between Point{x=4, y=5} and Point3D{x=1, y=1, z=1} is: 5.0
Point3D.distance(**Point**)
Distance between Point3D{x=1, y=1, z=1} and Point3D{x=2, y=5, z=9} is: 4.123105625617661
Point3D.distance(Point)
Distance between Point3D{x=2, y=5, z=9} and Point{x=1, y=1} is: 1.4142135623730951
Point.distance(Point)
Distance between Point{x=1, y=1} and Point{x=4, y=5} is: 5.0
Point.distance(**Point**)
Distance between Point{x=4, y=5} and Point3D{x=1, y=1, z=1} is: 5.0

Всюди, де у якості аргумента повинен використовуватись об'єкт дочірнього класу `Point3D`, використовується об'єкт батьківського класу `Point`. При цьому у першому та останньому випадках відстань розраховується правильно внаслідок фактичного збігання реалізації методів розрахунку відстані між двома точками на площині та між проєкцією тривимірної точки на площину X0Y та двовимірною точкою. Відстань між двома тривимірними точками обрахована неправильно (правильне значення дорівнює `9.0`). Зауважимо, що виведення до консолі типу об'єктів у рядку зі значенням відстані всюди є правильним, оскільки воно визначається на етапі виконання програми. А перевантажені методи обираються на етапі компіляції і усі елементи масиву на цьому етапі є об'єктами класу `Point`.

Для описаної задачі вирішення проблеми може бути досить простим – необхідно при виконанні програми перевірити тип кожного елемента на відповідність дочірньому типу і отримати посилання на приведений до цього типу об'єкт, який і передати як аргумент (завдяки доданій у Java 14 можливості визначення автоматично приведеної змінної у операторі `instanceof`, такий код є спрощеним, зміни показані жирним шрифтом):

```
for (int i = 0; i < pointArr.length - 1; i++) {
    if (pointArr[i + 1] instanceof Point3D nextPoint) {
        System.out.println("Distance between " + pointArr[i]
                + " and " + nextPoint
                + " = " + pointArr[i].distance(nextPoint));
    } else {
        System.out.println("Distance between " + pointArr[i]
                + " and " + pointArr[i + 1]
                + " = " + pointArr[i].distance(pointArr[i + 1]));
    }
}
```

Наразі програма демонструє правильний результат (відповідні місця виділені жирним шрифтом):

Point.distance(**Point3D**)
Distance between Point{x=4, y=5} and Point3D{x=1, y=1, z=1} = 5.0
Point3D.distance(**Point3D**)
Distance between Point3D{x=1, y=1, z=1} and Point3D{x=2, y=5, z=9} = 9.0
Point3D.distance(Point)
Distance between Point3D{x=2, y=5, z=9} and Point{x=1, y=1} = 1.4142135623730951
Point.distance(Point)
Distance between Point{x=1, y=1} and Point{x=4, y=5} = 5.0
Point.distance(**Point3D**)
Distance between Point{x=4, y=5} and Point3D{x=1, y=1, z=1} = 5.0

Якщо звернути увагу на перевантажені методи `distance` класу `Point`, можна побачити дублювання коду обчислення відстані між точками. Правильно було б винести цей код в окремий приватний метод. Але ми використаємо поліморфізм для вирішення цієї задачі, просто видаливши цей метод з класу `Point` (у методі `double distance(Point p)`, що залишився, скоригуємо виведення до консолі, а у класі `Point3D` необхідно тільки прибрати анотацію `@Override` над методом, який перевизначав видалений).

Запуск програми виведе неочікувану помилку (виділено жирним шрифтом):

Point.distance(Point) or Point.distance(Point3D)
Distance between Point{x=4, y=5} and Point3D{x=1, y=1, z=1} = 5.0
Point3D.distance(**Point**)

Distance between Point3D{x=1, y=1, z=1} and Point3D{x=2, y=5, z=9} = 4.123105625617661
Point3D.distance(Point)
Distance between Point3D{x=2, y=5, z=9} and Point{x=1, y=1} = 1.4142135623730951
Point.distance(Point) or Point.distance(Point3D)
Distance between Point{x=1, y=1} and Point{x=4, y=5} = 5.0
Point.distance(Point) or Point.distance(Point3D)
Distance between Point{x=4, y=5} and Point3D{x=1, y=1, z=1} = 5.0

Для розуміння цієї помилки розглянемо діаграму класів (Рис. 1) та проаналізуємо роботу програми для етапу, коли виникла помилка. На другій ітерації циклу з об'єкта `Point3D` повинен викликатися метод обчислення відстані до об'єкту `Point3D`. Однак, оскільки усі елементи масиву оголошені як об'єкти `Point`, метод `double distance(Point3D p)` шукається у класі `Point`. Раніше він був присутнім і обирався компілятором, але на етапі Run-time виконувався перевизначений метод класу `Point3D`, оскільки на цьому етапі системі виконання Java зрозуміло, що фактичний тип об'єкта – `Point3D`. Наразі ж метод `double distance(Point3D p)` у класі `Point` відсутній і компілятор автоматично приводить тип аргумента `nextPoint` до `Point`, оскільки `Point3D` є підтипом `Point`, і обирає метод `double distance(Point p)` класу `Point`. Далі, аналогічно, на етапі Run-time виконується перевизначений метод класу `Point3D`, оскільки на цьому етапі системі виконання Java зрозуміло, що фактичний тип об'єкта – `Point3D`. Підтверджує такі висновки виконання на другій ітерації циклу методу `double distance(Point p)` класу `Point` у разі коментування методу його перевизначеної версії у класі `Point3D`.

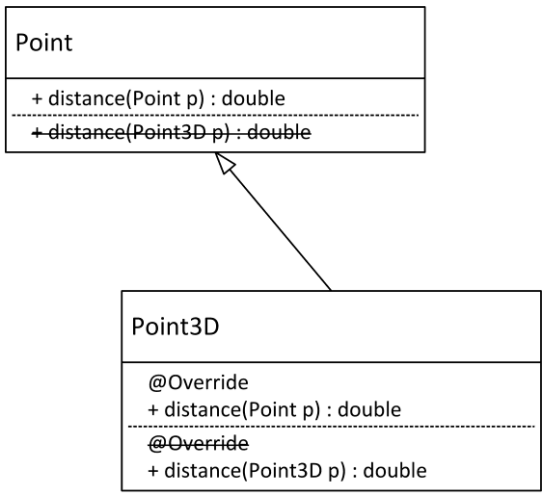

Рисунок 1 – Діаграма класів з видаленим перевантаженим методом

Вочевидь, для правильної роботи програми необхідно виконати приведення до `Point3D` у відповідних випадках також і об'єктів, з яких викликається метод (зміни показані жирним шрифтом):

```
for (int i = 0; i < pointArr.length - 1; i++) {
    if (pointArr[i] instanceof Point3D currentPoint
            && pointArr[i + 1] instanceof Point3D nextPoint) {
        System.out.println("Distance between " + currentPoint
                + " and " + nextPoint
                + " = " + currentPoint.distance(nextPoint));
    } else {
        System.out.println("Distance between " + pointArr[i]
                + " and " + pointArr[i + 1]
                + " = " + pointArr[i].distance(pointArr[i + 1]));
    }
}
```

При цьому пошук перевантаженого метода на етапі компіляції буде виконуватися у класі `Point3D` і програма працюватиме правильно. Таким чином, у даній ситуації ми бачимо комбіновану роботу технологій поліморфізму, реалізованих у Java – обрання перевантаженого

методу на етапі компіляції та обрання перевизначеного методу, який фактично виконується, на етапі Run-time.

На цьому можна було б завершувати дослідження обрання перевантаженого методу з поліморфними аргументами, але, на наш погляд, залишається цікавим питання, зазначене у дискусії [4]: як оптимізувати програму у разі, якщо існує багато перевантажених методів, що приймають поліморфні аргументи. Використання додаткових розгалужень підвищує цикломатичну складність коду [6], бажано було б динамічно виконувати приведення до фактичного типу аргументу. Оскільки фактичний тип зрозумілий тільки на етапі виконання програми, можна використати технологію рефлексії. При її використанні цикл у методі `main` може мати наступний вигляд:

```
for (int i = 0; i < pointArr.length - 1; i++) {
    System.out.println("Distance between " + pointArr[i]
            + " and " + pointArr[i + 1] + " is: "
            + argCastDistance(pointArr[i], pointArr[i + 1]));
}
```

Визначення фактичного типу об'єктів на етапі виконання програми та виклик відповідного перевантаженого методу засобами рефлексії виконується у методі `double argCastDistance(Point currentPoint, Point nextPoint)`:

```
private static double argCastDistance(Point currentPoint, Point nextPoint) {
    Class currentPointClass = currentPoint.getClass();
    Class nextPointClass = nextPoint.getClass();
    try {
        Method method = currentPointClass.getMethod("distance",
nextPointClass);
        return (double) method.invoke(currentPoint, nextPoint);
    } catch (Exception e) {
        e.printStackTrace();
        return -1.0;
    }
}
```

Для перевірки рішення додаймо до ієрархії клас `Point4D` з відповідними методами обчислення відстані між точками `Point4D`, а також між `Point4D` та `Point3D` і `Point4D` та `Point` (в їх реалізаціях залишимо тільки ідентифікаційне виведення до консолі, модифікатор видимості властивості z у класі `Point3D` змінимо на `protected`):

```
public class Point4D extends Point3D {

    private int t;

    /* Constructors */
    ...

    /* Calculates distance between two Point4D instances */
    public double distance(Point4D p) {
        System.out.println("Point4D.distance(Point4D)");
        return 400.0;
    }

/* Calculates distance between this Point4D instance and Point3D instance */
    @Override
    public double distance(Point3D p) {
        System.out.println("Point4D.distance(Point3D)");
        return 300.0;
    }

/* Calculates distance between this Point4D instance and Point instance */
    @Override
    public double distance(Point p) {
        System.out.println("Point4D.distance(Point)");
        return 200.0;
    }
```

```
    /* toString() implementation */
    ...
}
```

Також додаймо методи обчислення відстані до 4-вимірної точки до класу `Point`:

```
public double distance(Point4D p) {
    System.out.println("Point.distance(Point4D)");
    int dx = p.x - this.x;
    int dy = p.y - this.y;
    return Math.sqrt(dx * dx + dy * dy);
}
```

та класу `Point3D`:

```
public double distance(Point4D p) {
    System.out.println("Point3D.distance(Point4D)");
    return 300.0;
}
```

У методі `main` доповнимо масив новими членами так, щоби були задіяні усі методи, наприклад:

```
Point[] pointArr = {new Point(4, 5), new Point3D(1, 1, 1), new Point3D(2, 5,
9),
        new Point(1, 1), new Point(4, 5), new Point4D(1, 1, 1, 1),
        new Point4D(2, 5, 9, 1), new Point3D(2, 5, 9), new Point4D(1, 1, 1,
1),
        new Point(4, 5)};
```

Запуск програми з викликом відповідних методів за допомогою рефлексії виводить правильний результат:

```
Point.distance(Point3D)
Distance between Point{x=4, y=5} and Point3D{x=1, y=1, z=1}  is: 5.0
Point3D.distance(Point3D)
Distance between Point3D{x=1, y=1, z=1}  and Point3D{x=2, y=5, z=9}  is: 9.0
Point3D.distance(Point)
Distance between Point3D{x=2, y=5, z=9}  and Point{x=1, y=1} is: 1.4142135623730951
Point.distance(Point)
Distance between Point{x=1, y=1} and Point{x=4, y=5} is: 5.0
Point.distance(Point4D)
Distance between Point{x=4, y=5} and Point4D{x=1, y=1, z=1, t=1} is: 5.0
Point4D.distance(Point4D)
Distance between Point4D{x=1, y=1, z=1, t=1} and Point4D{x=2, y=5, z=9, t=1} is: 400.0
Point4D.distance(Point3D)
Distance between Point4D{x=2, y=5, z=9, t=1} and Point3D{x=2, y=5, z=9}  is: 300.0
Point3D.distance(Point4D)
Distance between Point3D{x=2, y=5, z=9}  and Point4D{x=1, y=1, z=1, t=1} is: 300.0
Point4D.distance(Point)
Distance between Point4D{x=1, y=1, z=1, t=1} and Point{x=4, y=5} is: 200.0
```

Таким чином, рефлексія дозволяє зробити код більш універсальним та коротшим, порівняно з реалізацією, що передбачає приведення типів аргументів у гілках розгалуження (наведемо її для використаного прикладу):

```
for (int i = 0; i < pointArr.length - 1; i++) {
    if (pointArr[i + 1] instanceof Point4D nextPoint4D) {
        pointArr[i].distance(nextPoint4D);
    } else if (pointArr[i + 1] instanceof Point3D nextPoint3D) {
        pointArr[i].distance(nextPoint3D);
    } else {
        pointArr[i].distance(pointArr[i + 1]);
    }
}
```

Для оцінки вартості використання рефлексії ми розробили програму, що генерує випадковим чином об'єкти `Point`, `Point3D` та `Point4D`, задаючи їм випадкові координати у діапазоні від 1 до 9 включно та розміщує їх у масиві, розмір якого задається як параметр. На Рис. 2 наведені результати вимірювання часу роботи фрагментів програми, що містять цикли обробки масиву без та з використанням рефлексії (в методах були закоментовані оператори виведення до консолі).

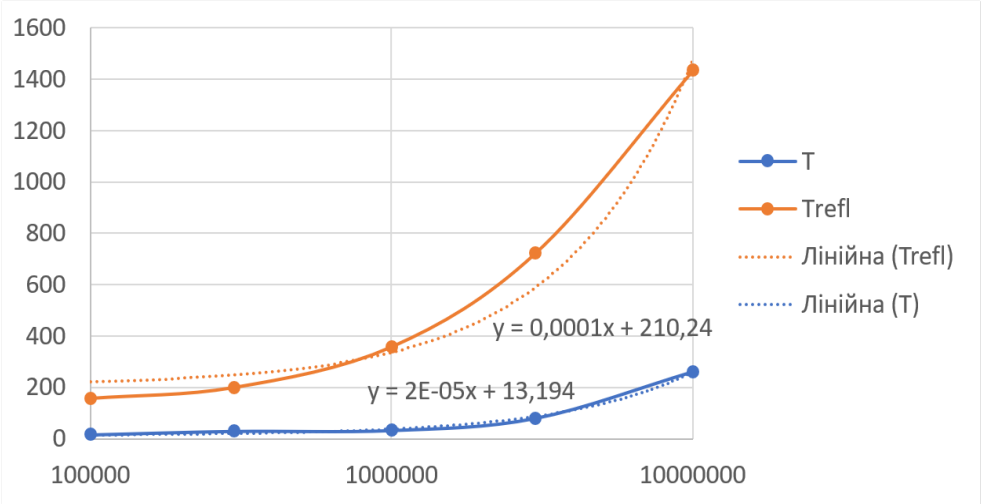

Рисунок 2 – Порівняння часу виконання програми з різними реалізаціями приведення типу аргументу перевантажених методів

Як і передбачалось, рефлексія збільшує час виконання програми приблизно на порядок і зі збільшенням розміру масиву цей час також зростає відносно реалізації без використання рефлексії.

Висновки. Обрання перевантаженого методу у мові програмування Java – є однією з технологій поліморфізму, яка дозволяє визначати різні реалізації однойменних методів за допомогою різних наборів аргументів, що дозволяє писати гнучкий та зрозумілий код. Неоднозначності, що можуть виникати при обранні перевантаженого метода зводяться до мінімуму реалізованим у Java алгоритмом обрання перевантаженого методу. Внаслідок реалізації обрання перевантажених методів на етапі компіляції програми, у разі використання у якості аргументів об'єктів поліморфних типів, обрання перевантаженого методу може бути непередбаченим. У такому випадку необхідно застосовувати примусове приведення типу аргументу метода до потрібного відповідного типу. Завдяки використанню разом обрання перевантаженого метода на етапі компіляції та обрання перевизначеного метода на етапі виконання можливе зменшення кількості перевантажених методів у разі дублювання їх реалізацій.

Для класів та ієрархій класів з великою кількістю перевантажених методів з різними реалізаціями запропонований вище підхід приводить до коду з великою кількістю приведень до необхідних типів у гілках розгалужень, що робить його великим та ускладненим. Можливе зменшення розміру такого коду за рахунок динамічного визначення типу аргументу метода та виклику перевантаженого метода засобами рефлексії, які надає Java. Але при цьому суттєво (приблизно на порядок для наведеного прикладу) збільшується час виконання програми. Розробник, в залежності від конкретної ситуації, може обирати між більш швидким рішеннями з розгалуженим кодом та оптимізованим за розміром кодом, що потребує більшого часу обчислень.

Це дослідження вивчає деталі обрання перевантажених методів при передачі їм одного поліморфного аргументу. Подальші дослідження можуть вивчати механізм обрання перевантаженого методу при передачі йому декількох поліморфних аргументів або довільної кількості таких аргументів (varargs).

Список бібліографічного опису

1. Герберт Шилдт, Др. Денні Ковард Java™. Повний довідковий матеріал. Тринадцята редакція. Всебічне охоплення мови Java. McGraw Hill, 2024. 2750 с.
2. Рамеш Фадатаре. Перевантаження методів у Java з прикладами. [Електронний ресурс] – Режим доступу: https://www.javaguides.net/2018/09/method-overloading-in-java-with-examples.html.

3. Джеймс Гослінг та ін. Специфікація мови Java®. Редакція Java SE 22. Розділ 15.12. Вирази виклику методів. [Електронний ресурс] – Режим доступу: https://docs.oracle.com/javase/specs/jls/se22/html/jls-15.html#jls-15.12.
4. Java Class.cast до найбільш конкретного з перевантаженням методів. Сайт Stack Overflow. [Електронний ресурс] – Режим доступу: https://stackoverflow.com/questions/51639899/java-class-cast-to-most-specific-with-method-overloading.
5. Перспективна проекція за 5 хвилин: Частина 2 – математика! [Електронний ресурс] – Режим доступу: https://youtu.be/g7Pb8mrwcJ0.
6. Крістіан Майєр Мистецтво чистого коду. Найкращі методи усунення складності та спрощення життя. No Starch Press, Inc. 2022. 178 с.

References

1. Herbert Schildt, Dr. Danny Coward Java™. The Complete Reference. Thirteenth Edition. Comprehensive Coverage of the Java Language. McGraw Hill, 2024. 2750 p.
2. Ramesh Fadatare. Method Overloading in Java with Examples. [Electronic resource] - Access mode: https://www.javaguides.net/2018/09/method-overloading-in-java-with-examples.html.
3. James Gosling et al. The Java® Language Specification. Java SE 22 Edition. Chapter 15.12. Method Invocation Expressions. [Electronic resource] - Access mode: https://docs.oracle.com/javase/specs/jls/se22/html/jls-15.html#jls-15.12.
4. Java Class.cast to most specific with method overloading. Stack Overflow site. [Electronic resource] - Access mode: https://stackoverflow.com/questions/51639899/java-class-cast-to-most-specific-with-method-overloading.
5. Perspective projection in 5 minutes: Part 2 – the math! [Electronic resource] - Access mode: https://youtu.be/g7Pb8mrwcJ0.
6. Christian Mayer The Art of Clean Code. Best Practices to Eliminate Complexity and Simplify Your Life. No Starch Press, Inc. 2022. 178 p.