Міністерство освіти і науки України
Волинський національний університет імені Лесі Українки
Навчально-науковий фізико-технологічний інститут

В. П. Муляр

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій

Луцьк
Вежа-друк
2022

УДК 004.438(07)
М 90

*Рекомендовано до друку науково-методичною радою
Волинського національного університету імені Лесі Українки
(протокол № 3 від 16.11.2022 р.)*

Рецензенти:

Яцюк С. М. **–** кандидат педагогічних наук, доцент кафедри загальної математики та методики навчання інформатики, декан факультету інформаційних технологій і математики Волинського національного університету імені Лесі Українки;

Багнюк Н. В. **–** кандидат технічних наук, доцент кафедри комп'ютерної інженерії та кібербезпеки Луцького національного технічного університету.

Муляр В. П.

М 90 Об'єктно-орієнтоване програмування: конспект лекцій. Луцьк : Вежа-Друк, 2022. 122 с.

У конспекті лекцій подано теоретичні основи об'єктно-орієнтованого програмування на мові Java. На конкретних прикладах розглянуто основні поняття і термінологію ООП та основні її принципи: інкапсуляцію, успадкування, поліморфізм та ін.

Для студентів ЗВО педагогічного та технічного спрямування, які вивчають дисципліни «Об'єктно-орієнтоване програмування», «Програмування на Java», «Проєктування і розробка користувацьких інтерфейсів», «Алгоритми і структури даних», «Комп'ютерна фізика», «Моделювання фізичних явищ і процесів», а також для тих, хто починає освоювати об'єктно-орієнтоване програмування.

УДК 004.438(07)

© Муляр В. П., 2022
© Волинський національний університет
імені Лесі Українки, 2022

ЗМІСТ

ВСТУП 5
ЛЕКЦІЯ 1. КОНЦЕПЦІЯ ООП 6
    Основні поняття ООП 6
    Правила іменування Java 9
ЛЕКЦІЯ 2. ОБ’ЄКТИ ТА КЛАСИ 12
    Об’єкт у Java 13
    Клас у Java 14
    Поле в Java 16
    Метод у Java 16
    Анонімний об’єкт 23
ЛЕКЦІЯ 3. МЕТОД У JAVA 26
    Заздалегідь визначений метод 28
    Метод, визначений користувачем 30
    Статичний метод 32
    Метод екземпляра 33
    Абстрактний метод 35
ЛЕКЦІЯ 4. КОНСТРУКТОРИ 36
    Конструктор Java за замовчуванням 37
    Перевантаження конструктора в Java 40
    Відмінність між конструктором та методом у Java 41
    Конструктор копіювання у Java 42
    Копіювання значень без конструктора 43
ЛЕКЦІЯ 5. КЛЮЧОВЕ СЛОВО STATIC У JAVA 45
    Статичне поле в Java 45
    Статичний метод у Java 49
    Статичний блок у Java 51
ЛЕКЦІЯ 6. КЛЮЧОВЕ СЛОВО THIS У JAVA 52
    this: посилання на змінну екземпляра поточного класу 53
    this: виклик методу поточного класу 56
    this(): виклик конструктора поточного класу 57
    this: передати як аргумент у методі 60
    this: передати як аргумент у виклик конструктора 60
    this: повернення поточного екземпляра класу 61
ЛЕКЦІЯ 7. ІНКАПСУЛЯЦІЯ В JAVA 62
ЛЕКЦІЯ 8. ПАКЕТ У JAVA 66
    Пакет у Java 66
    Підпакет у Java 70

ЛЕКЦІЯ 9. УСПАДКУВАННЯ В JAVA 73
Однорівневе успадкування... 76
Багаторівневе успадкування 77
Ієрархічне успадкування 78
ЛЕКЦІЯ 10. ВІДНОСИНИ МІЖ КЛАСАМИ В JAVA 80
1. IS-A відносини.. 80
2. HAS-A відносини 80
ЛЕКЦІЯ 11. ПЕРЕВАНТАЖЕННЯ МЕТОДУ В JAVA....................... 84
Перевантаження методу: зміна кількості аргументів 85
Перевантаження методу: зміна типу даних аргументів.............. 86
Метод перевантаження та просування типу 87
ЛЕКЦІЯ 12. ПЕРЕВИЗНАЧЕННЯ МЕТОДУ В JAVA 90
ЛЕКЦІЯ 13. ПОЛІМОРФІЗМ У JAVA 95
Поліморфізм під час виконання в Java 95
Модернізація... 95
Приклад поліморфізму під час виконання в Java 96
ЛЕКЦІЯ 14. АБСТРАКТНИЙ КЛАС У JAVA................................... 102
Абстракція в Java.. 103
Абстрактний клас 103
Абстрактний метод 104
Абстрактний клас, що містить конструктор, поля та методи .. 107
ЛЕКЦІЯ 15. ІНТЕРФЕЙС У JAVA 109
Поняття інтерфейсу в Java 109
Зв’язок між класами та інтерфейсами 111
Множинне успадкування за інтерфейсом у Java 114
Статичний метод в інтерфейсі 117
Інтерфейс з тегами 117
Різниця між абстрактним класом та інтерфейсом 118
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ... 121

ВСТУП

У процесі пізнання навколишнього світу людина стикається з тим, що її мозок просто не може сприймати цей світ цілком з усіма його проявами. Мозок спрощує отримувану інформацію, доводячи її до якоїсь системи. Щоб щось зрозуміти, ви повинні це спростити. Поступово ви будете заглиблюватися в знаннях, але на початку вам потрібно зрозуміти загальну схему. Так ось парадигма програмування – це загальна схема, яка показує, яким способом людина перетворює технічне завдання в програмний код.

Парадигма ООП досить проста і вирішує головну проблему – що робити зі складною предметною областю і складним кодом. Крім того, така парадигма ще й універсальна. Саме тому ООП так добре прижилася.

Об'єктно-орієнтоване програмування розглядає всю систему у вигляді об'єктів, які якимось чином один з одним взаємодіють. Оскільки людському мозку легше мислити об'єктами, ми автоматично розуміємо, що і у якого об'єкта має бути. Людині легко зрозуміти, де розташувати ті чи інші методи в коді. Завдяки цьому ООП забезпечує дуже легку і прозору структуру розташування коду.

На сьогоднішній день тільки об'єктно-орієнтована парадигма є абсолютно універсальною. Переважна кількість завдань на ній вирішується максимально ефективно.

У конспекті лекцій подано теоретичні відомості з об'єктно-орієнтованого програмування, розкрито основні поняття і принципи ООП на Java: об'єкти та класи, методи та конструктори, ключові слова, успадкування, агрегування, поліморфізм, абстрагування, пакети та інтерфейси, інкапсуляція та ін. На конкретних прикладах проілюстровано переваги об'єктно-орієнтованого підходу над процедурно-орієнтованим програмуванням.

Для студентів ЗВО педагогічного та технічного спрямування, які вивчають дисципліни «Об'єктно-орієнтоване програмування», «Програмування на Java», «Проєктування і розробка користувацьких інтерфейсів», «Алгоритми і структури даних», «Комп'ютерна фізика», «Моделювання фізичних явищ і процесів», а також для тих, хто починає освоювати об'єктно-орієнтоване програмування.

ЛЕКЦІЯ 1. КОНЦЕПЦІЯ ООП

План

1. Основні поняття ООП
2. Правила іменування в Java

Основною метою об'єктно-орієнтованого програмування (ООП) є реалізація сутностей реального світу, наприклад, об'єкт, класи, абстрагування, успадкування, поліморфізм тощо.

Уважають, що першою об'єктно-орієнтованою мовою програмування була Simula, а першою повністю об'єктно-орієнтованою мовою програмування – Smalltalk.

Популярними об'єктно-орієнтованими мовами є Java, C#, PHP, Python, C++ тощо.

Основні поняття ООП

Об'єкт означає об'єкт реального світу, такий як ручка, стілець, стіл, комп'ютер, годинник тощо.

Об'єктно-орієнтоване програмування – це методологія або парадигма для розробки програм з використанням класів та об'єктів. Вона спрощує розробку та використання програмного забезпечення, надаючи деякі концепції:

- об'єкт;
- клас;
- успадкування;
- поліморфізм;
- абстрагування;
- інкапсуляція.

Крім цих принципів, у об'єктно-орієнтованому програмуванні використовуються інші концепції:

- зв'язування;
- згуртованість;
- асоціація;
- агрегування;
- композиція.

Об'єкт

Будь-яка сутність, яка має властивості та поведінку, відома як *об'єкт*. Наприклад, стілець, ручка, стіл, клавіатура, велосипед тощо. Об'єкт може бути фізичним або логічним.

Об'єкт можна визначити як екземпляр класу. Об'єкт має адресу і займає деяке місце в пам'яті. Об'єкти можуть спілкуватись між собою, не вдаючись до деталей даних або коду один одного. Єдине, що необхідно, – це тип прийнятого повідомлення та тип відповіді, що повертається об'єктами.

Наприклад, собака – це об'єкт, оскільки вона має такі стани, як колір, ім'я, порода тощо, а також поведінку, наприклад, махає хвостом, гавкає, їсть тощо.

Клас

Колекція об'єктів називається *класом*. Це логічна сутність.

Клас також можна визначити як шаблон, за яким можна створити окремий об'єкт. Клас не займає жодного місця.

Успадкування

Коли один об'єкт набуває всіх властивостей та поведінки батьківського об'єкта, це називається *успадкуванням*. Він забезпечує повторне використання коду. Він використовується для досягнення поліморфізму під час виконання.

Поліморфізм

Якщо *одне завдання виконується по-різному,* це називається *поліморфізмом*. Наприклад: переконати клієнта, намалювати щось, наприклад, форму, трикутник, прямокутник тощо.

У Java ми використовуємо перевантаження та перевизначення (заміну) методів для досягнення поліморфізму.

Інші приклади поліморфізму: кішка нявкає, собака гавкає тощо.

Абстракція

Приховування внутрішніх деталей та демонстрація функціональності називається *абстракцією*. Наприклад, під час прийому телефонного дзвінка нам невідомо, як проходить процес внутрішньої обробки сигналів.

У Java ми використовуємо абстрактний клас та інтерфейс для досягнення абстрагування.

Інкапсуляція

Зв'язування (або обгортання) коду та даних разом в єдину одиницю називається *інкапсуляцією*. Наприклад, капсула містить різні ліки.

Клас Java є прикладом інкапсуляції. Java – це повністю інкапсульований клас, оскільки всі поля тут є приватними.

Зв'язування

Зв'язування відноситься до знань або інформації або залежності іншого класу. Зв'язування виникає тоді, коли класи використовують один одного. Якщо клас містить детальну інформацію про інший класу, то існує сильний зв'язок. У Java ми використовуємо приватні, захищені та відкриті модифікатори для відображення рівня видимості класу, методу та поля. Ви можете використовувати інтерфейси для слабшого зв'язку у випадку відсутності конкретної реалізації.

Згуртованість

Згуртованість відноситься до рівня компонента, який виконує єдине чітко визначене завдання. Єдине чітко визначене завдання виконується дуже згуртованим методом. Слабо згуртований метод розділить завдання на окремі частини. Пакет java.io – це дуже згуртований пакет, оскільки він має класи та інтерфейс, пов'язані з введенням-виведенням. Однак пакет java.util є слабо згуртованим, оскільки має непов'язані класи та інтерфейси.

Асоціація

Асоціація означає зв'язок між об'єктами. Тут один об'єкт може бути пов'язаний з одним об'єктом або з багатьма об'єктами. Між об'єктами може бути чотири типи зв'язку:

- один до одного;
- один до багатьох;
- багато до одного;
- багато до багатьох.

Давайте розберемося у відносинах на прикладах у реальному часі. Наприклад, в одній країні може бути один прем'єр-міністр (один до одного), а у прем'єр-міністра може бути багато міністрів (один до багатьох). Крім того, багато депутатів можуть мати одного прем'єр-міністра (багато до одного), а багато міністрів можуть мати багато відомств (багато до багатьох).

Агрегування

Агрегування – це спосіб досягнення асоціації. Агрегування передбачає такі відносини, коли один об'єкт містить інші об'єкти як частину свого стану. Це спосіб, який забезпечує слабкий зв'язок між об'єктами. Це ще один спосіб повторного використання об'єктів.

Композиція

Композиція також є способом досягнення асоціації. Композиція – це відносини, при якому один об'єкт містить інші об'єкти як частину свого стану. Між об'єктом, що містить інші об'єкти, і залежним об'єктом існує міцний зв'язок. Це стан, коли об'єкти, що містять інші

об'єкти, не мають незалежного існування. Якщо видалити батьківський об'єкт, усі дочірні об'єкти будуть видалені автоматично.

Переваги ООП над процедурно-орієнтованим програмуванням

1. На відміну від процедурно-орієнтованого програмування ООП полегшує розробку та використання програм у випадку зростання коду із збільшенням розміру проєкту.

2. ООП забезпечує приховування даних, тоді як у процедурно-орієнтованій мові програмування до глобальних даних можна отримати доступ із будь-якого місця.

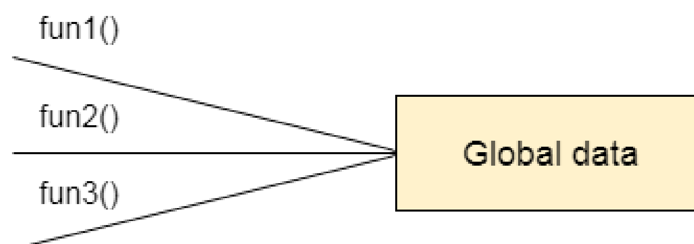

Рис. 1. Представлення даних у процедурно-орієнтованому програмуванні

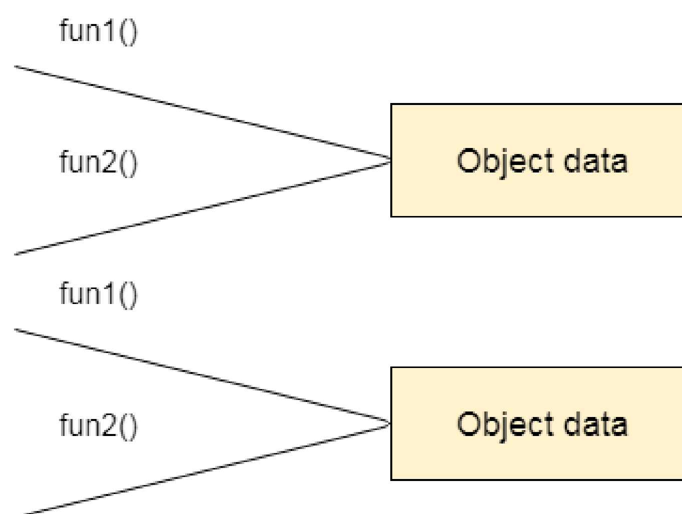

Рис. 2. Представлення даних в об'єктно-орієнтованому програмуванні

3. ООП дає можливість значно ефективніше імітувати події в реальному світі.

Правила іменування Java

Правила іменування Java – це правила, яких слід дотримуватися, коли потрібно назвати свої ідентифікатори, такі як клас, пакет, змінна, константа, метод тощо.

Правила іменування запропоновані кількома спільнотами Java, такими як Sun Microsystems та Netscape.

Усі класи, інтерфейси, пакети, методи та поля мови програмування Java наведені згідно з умовами іменування Java. Якщо не дотримуватись цих правил, то це може призвести до плутанини або помилкового коду.

Перевага правил іменування в Java

Дотримання стандартних правил іменування Java полегшує читання коду програми, при цьому менше часу витрачається на з'ясування того, що робить код.

Нижче наведені основні правила, яких потрібно дотримуватись, надаючи назву кожному ідентифікатору:

- ім'я не повинно містити пробілів;
- ім'я не повинно починатися зі спеціальних символів, таких як & (амперсанд), $ (долар), _ (підкреслення).

Давайте розглянемо інші правила, яких слід дотримуватись під час іменування ідентифікаторів.

Клас

- Починати слід з великої літери;
- це має бути іменник, такий як Color, Button, System, Thread тощо;
- використовуйте відповідні слова замість скорочень.

Приклад:

```
public class Employee {
    // фрагмент коду
}
```

Інтерфейс

- Починати слід з великої літери;
- це має бути прикметник, як Runnable, Remote, ActionListener;
- використовуйте відповідні слова замість скорочень.

Приклад:

```
interface Printable {
    // фрагмент коду
}
```

Метод

- Починати слід з малої літери;
- це має бути дієслово, таке як main(), print(), println();
- якщо назва містить кілька слів, почніть її з малої літери, а потім з великої, наприклад actionPerformed().

Приклад:

```
class Employee {
    // метод
    void draw(){
        // фрагмент коду
    }
}
```

Змінна

– Повинна починатися з малої літери;

– вона не повинна починатися зі спеціальних символів, таких як & (амперсанд), $ (долар), _ (підкреслення);

– якщо назва містить кілька слів, почніть її з малої літери, а потім із великої літери, наприклад firstName, lastName;

– уникайте використання односимвольних змінних, таких як x, y, z.

Приклад:

```
class Employee {
    // змінна
    int id;
    // фрагмент коду
}
```

Пакет

– Він має записаним малими літерами, наприклад java, lang;

– якщо назва містить кілька слів, її слід розділити крапками (.), Такими як java.util, java.lang.

Приклад:

```
package com.javatpoint; // пакет
class Employee {
    // фрагмент коду
}
```

Константа

– Вона має бути написана великими літерами, такими як RED, YELLOW;

– якщо назва містить кілька слів, її слід відокремити підкресленням (_), таким як MAX_PRIORITY;

– вона може містити цифри, але не на першому місці.

Приклад:

```
class Employee {
    // константа
```

```
    static final int MIN_AGE = 18;
    // фрагмент коду
}
```

CamelCase в найменуваннях Java

Java дотримується синтаксису верблюжого регістру для іменування класу, інтерфейсу, методу та поля.

Якщо ім'я поєднує два слова, друге слово має завжди починатися з великої літери, наприклад actionPerformed(), firstName, ActionEvent, ActionListener тощо.

Контрольні запитання

1. Що таке об'єкт?
2. Що таке ООП?
3. Які концепції ООП ви знаєте?
4. Що таке клас?
5. Які переваги ООП над процедурно-орієнтованим програмуванням?
6. Яких правил іменування потрібно дотримуватися у процесі написання коду програм на Java?
7. Наведіть приклади об'єктів і класів у Java.

ЛЕКЦІЯ 2. ОБ'ЄКТИ ТА КЛАСИ

План

1. Об'єкт у Java
2. Клас у Java
3. Поле в Java
4. Метод у Java
5. Анонімний об'єкт

За методологією об'єктно-орієнтованого програмування ми розробляємо програми з використанням об'єктів та класів.

Об'єкт у Java є фізичною, а також логічною сутністю, тоді як клас у Java є лише логічною сутністю.

Об’єкт у Java

Рис. 3. Реальні приклади об’єктів

Сутність, яка має стан та поведінку, – це об’єкт, наприклад, стілець, велосипед, маркер, ручка, стіл, автомобіль тощо. Вона може бути фізичною чи логічною (матеріальною та нематеріальною). Прикладом нематеріального об’єкта є банківська система.

Об’єкт має три характеристики:

– *стан:* представляє дані (значення) об’єкта;

– *поведінка:* представляє поведінку (функціональність) такого об’єкта, як депозит, зняття тощо;

– *ідентичність:* ідентичність об’єкта зазвичай реалізується за допомогою унікального ідентифікатора. Зовнішній користувач не бачить значення ідентифікатора. Однак він використовується JVM внутрішньо для унікальної ідентифікації кожного об’єкта.

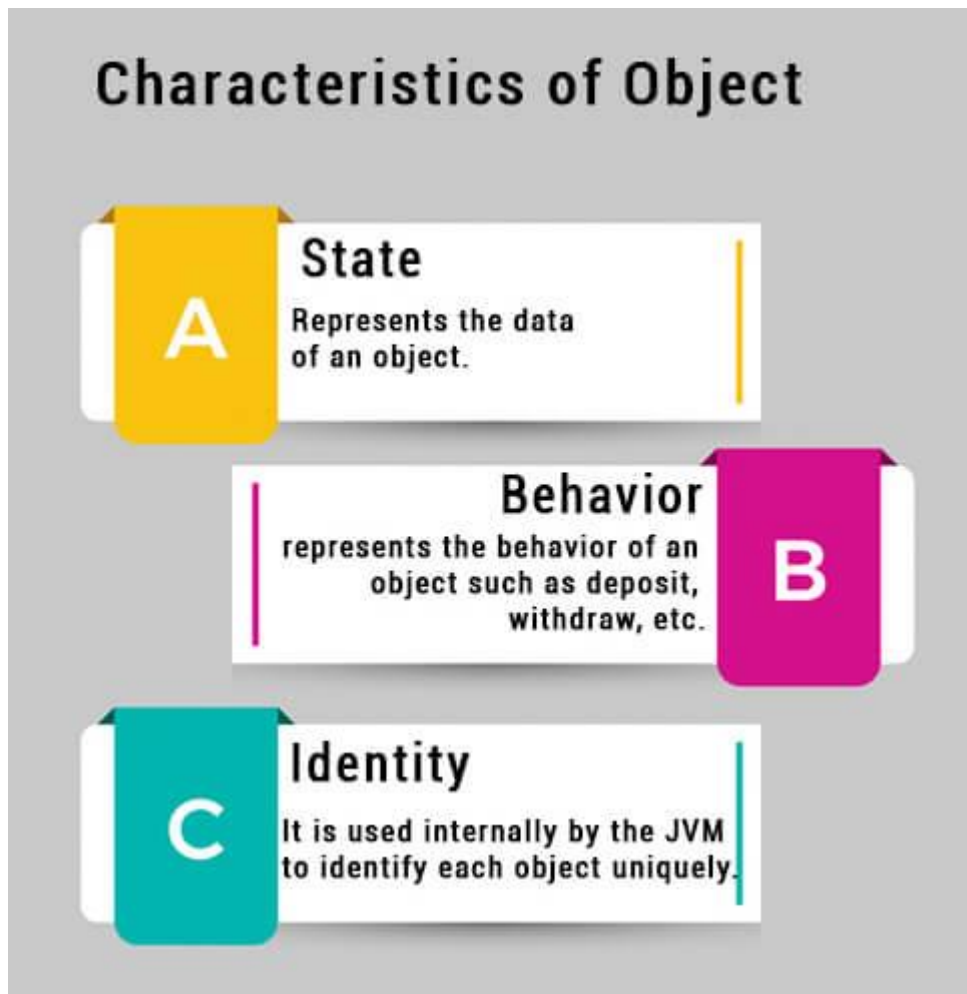

Рис. 4. Характеристики об’єкта

Наприклад, перо – це об’єкт. Його ім’я Reynolds; колір білий – це його стан. Він використовується для письма, тому письмо – це його поведінка.

Об’єкт – це екземпляр класу. Клас – це шаблон або план, за яким створюються об’єкти. Отже, об’єкт – це екземпляр (результат) класу.

Визначення об’єктів:

- об’єкт – *це сутність реального світу*;
- об’єкт – *це сутність під час виконання*;
- об’єкт – *це сутність, яка має стан та поведінку*;
- об’єкт є *екземпляром класу*.

Клас у Java

Клас – це група об’єктів, які мають спільні властивості. Це шаблон або план, за яким створюються об’єкти. Це логічна сутність. Клас не може бути фізичним.

Клас у Java може містити:

– поля;

– методи;
– конструктори;
– блоки;
– вкладений клас та інтерфейс.

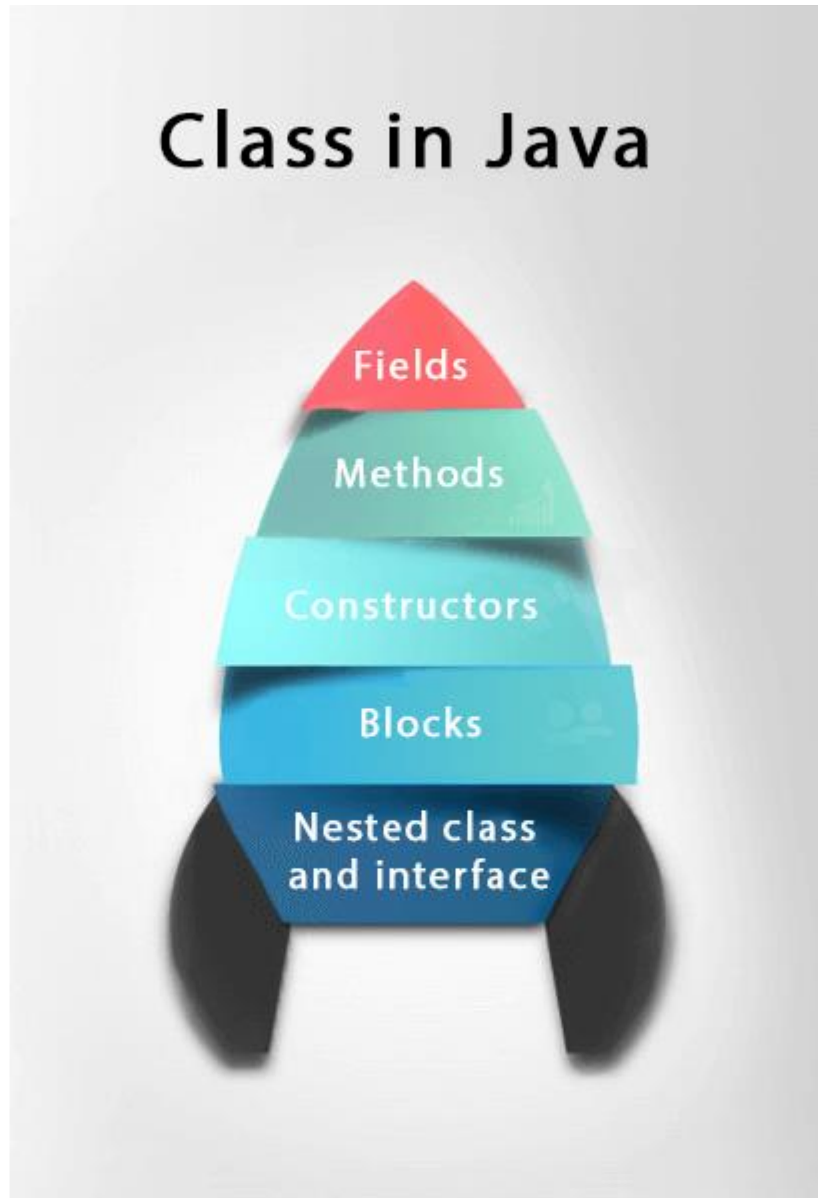

Рис. 5. Клас у Java

Синтаксис оголошення класу:

```
class <ім’я_класу> {
    поле;
    метод;
}
```

Поле в Java

Змінна, яка створюється всередині класу, але поза методом, називається *полем*. Під поле не виділяється пам'ять під час компіляції. Для нього виділяється пам'ять під час виконання, коли створюється об'єкт або екземпляр класу. Тому поле називають *змінною екземпляра*.

Метод у Java

У Java метод подібний до функції, яка використовується для опису поведінки об'єкта.

Перевага методу:

– повторне використання коду;

– оптимізація коду.

Ключове слово new в Java

Ключове слово new використовується для виділення пам'яті під час виконання. Усі об'єкти отримують пам'ять у зоні пам'яті кучі.

Приклад об'єкта та класу: main у класі

У цьому прикладі створюємо клас Student, який має два ідентифікатори з іменами полів даних. Ми створюємо об'єкт класу Student за допомогою ключового слова new і друкуємо значення об'єкта.

Тут ми створюємо метод main() всередині класу.

Файл: Student.java

```
// Програма Java для ілюстрації того, як визначити клас і поля
// Визначення класу Student
class Student {
    // Визначення полів
    int id;
    String name;
    // створення основного методу всередині класу Student
    public static void main(String args[]){
        // Створення об'єкта або екземпляра
        // Створення об'єкта Student
        Student s1 = new Student();
        // Друк значень об'єкта
        // Доступ до об'єкта через посилальну змінну
        System.out.println(s1.id);
        System.out.println(s1.name);
    }
}
```

Результат:

```
0
null
```

Приклад об’єкта та класу: main поза класом

У режимі реального часу ми створюємо класи та використовуємо їх з іншого класу. Це кращий підхід, ніж попередній. Давайте розглянемо простий приклад, де у нас є метод main() в іншому класі.

Ми можемо мати кілька класів у різних файлах Java або в одному файлі Java. Якщо ви визначаєте кілька класів в одному вихідному файлі Java, було б добре зберегти ім’я файлу з назвою класу, що має метод main().

Файл: TestStudent1.java

```
// Програма Java для демонстрації наявності основного методу
// в іншому класі
// Створення класу Student
class Student{
    int id;
    String name;
}
// Створення іншого класу TestStudent1, який містить
// основний метод
class TestStudent1 {
    public static void main(String[] args){
        Student s1 = new Student();
        System.out.println(s1.id);
        System.out.println(s1.name);
    }
}
```

Виведення на екран:

```
0
null
```

3 способи ініціалізації об’єкта

Існує 3 способи ініціалізації об’єкта в Java:

1) за посилальною змінною;

2) за методом;

3) за конструктором.

1. Приклад об’єкта та класу: ініціалізація за допомогою посилальної змінної

Ініціалізація об’єкта означає зберігання даних у об’єкті. Давайте розглянемо простий приклад, коли ми збираємося ініціалізувати об’єкт за допомогою посилальної змінної.

Файл: TestStudent2.java

```
class Student{
        int id;
        String name;
}
class TestStudent2{
        public static void main(String args[]){
                Student s1=new Student();
                s1.id=101;
                s1.name="Sonoo";
                // друк елементів з пробілом
                System.out.println(s1.id+" "+s1.name);
        }
}
```

Виведення на екран:

```
101 Sonoo
```

Ми також можемо створювати кілька об’єктів і зберігати в ньому інформацію за допомогою посилальної змінної.

Файл: TestStudent3.java

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  // Створення об’єктів
  Student s1=new Student();
  Student s2=new Student();
// Ініціалізація об’єктів
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  // Друк даних
  System.out.println(s1.id+" "+s1.name);
```

```
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Виведення на екран:

```
101 Sonoo
102 Amit
```

2. Приклад об’єкта та класу: ініціалізація методом

У цьому прикладі ми створюємо два об’єкти класу Student та ініціалізуємо значення для цих об’єктів, викликаючи метод insertRecord. Тут ми відображаємо стан (дані) об’єктів, викликаючи метод displayInformation().

Файл: TestStudent4.java

```
class Student {
	int rollno;
	String name;
	void insertRecord(int r, String n){
		rollno=r;
		name=n;
 }
void displayInformation() {
	System.out.println(rollno+" "+name);}
}
class TestStudent4{
	public static void main(String args[]){
		 Student s1=new Student();
		Student s2=new Student();
		s1.insertRecord(111,"Karan");
		s2.insertRecord(222,"Aryan");
		s1.displayInformation();
		s2.displayInformation();
	 }
}
```

Виведення на екран:

```
111 Karan
222 Aryan
```

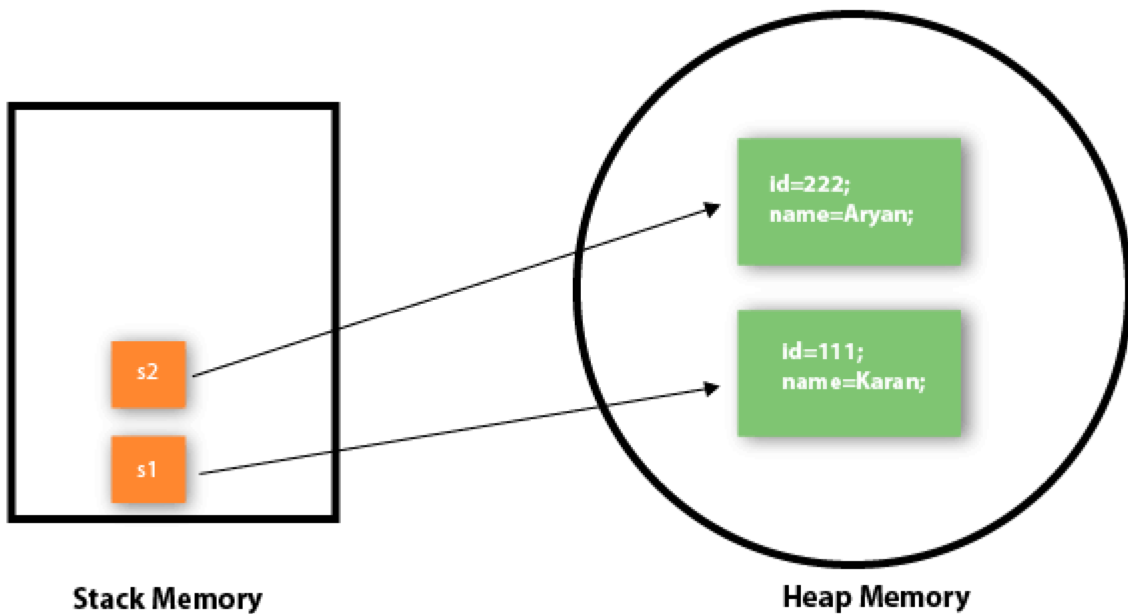

Рис. 6. Отримання об’єктами пам’яті в області пам’яті кучі

Як ви можете бачити на наведеному вище рисунку, об’єкт отримує пам’ять в області пам’яті кучі. Посилальна змінна відноситься до об’єкта, виділеного в області пам’яті кучі. Тут s1 і s2 – це посилальні змінні, які посилаються на об’єкти, виділені в пам’яті.

3. Приклад об’єкта та класу: ініціалізація за допомогою конструктора

Про конструктори в Java ми дізнаємося пізніше.

Приклад об’єкта та класу: Employee

Давайте розглянемо приклад, коли ми ведемо облік працівників.

Файл: TestEmployee.java

```
class Employee{
	int id;
	String name;
	float salary;    // заробітна плата
	void insert(int i, String n, float s) {
		id=i;
		name=n;
		salary=s;
	}
	void display(){
		System.out.println(id+" "+name+" "+salary);
	}
}
public class TestEmployee {
```

```
    public static void main(String[] args) {
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
        e1.insert(101,"ajeet",45000);
        e2.insert(102,"irfan",25000);
        e3.insert(103,"nakul",55000);
        e1.display();
        e2.display();
        e3.display();
    }
}
```

Виведення на екран:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Приклад об'єкта та класу: Rectangle

Наведено ще один приклад, який ілюструє використання класу Rectangle.

Файл: TestRectangle1.java

```
class Rectangle{
    int length;   // довжина
    int width;    // ширина
    void insert(int l, int w){
        length = l;
        width = w;
    }
    void calculateArea(){
        System.out.println(length*width);
    }
}
class TestRectangle1{
    public static void main(String args[]){
        Rectangle r1=new Rectangle();
        Rectangle r2=new Rectangle();
        r1.insert(11,5);
        r2.insert(3,15);
        r1.calculateArea();
```

```
        r2.calculateArea();
    }
}
```

Виведення на екран:

```
55
45
```

Які є різні способи створення об’єкта в Java?

Існує багато способів створення об’єкта в Java, а саме:

- за допомогою ключового слова new;
- із використанням методу newInstance();
- із застосуванням методу clone();
- шляхом десеріалізації;
- заводським способом тощо.

Про ці способи створення об’єкта ми дізнаємося пізніше.

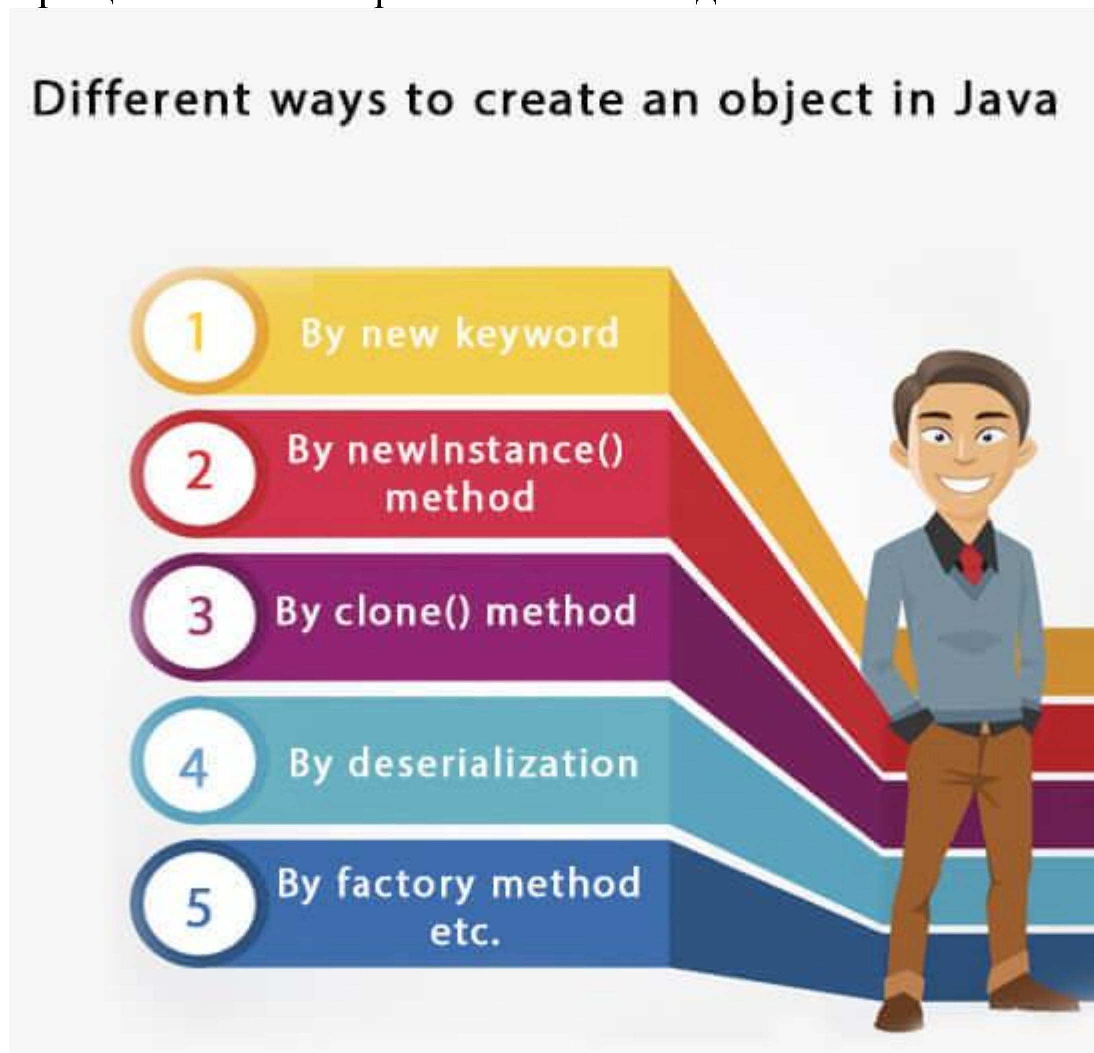

Рис. 7. Шляхи створення об’єктів у Java

Анонімний об’єкт

Анонім просто означає безіменний. Об’єкт, на який немає посилань, відомий як анонімний об’єкт. Його можна використовувати лише під час створення об’єкта.

Якщо вам доводиться використовувати об’єкт лише один раз, то у цьому випадку підходить анонімний об’єкт. Наприклад:

```
new Calculation(); // анонімний об’єкт
```

Метод виклику через посилання:

```
Calculation c = new Calculation();
c.fact(5);
```

Метод виклику через анонімний об’єкт

```
new Calculation().fact(5);
```

Давайте подивимося повний приклад анонімного об’єкта в Java.

```
class Calculation{
	void fact(int   n){
		int fact=1;
		for(int i=1;i<=n;i++){
			fact=fact*i;
		}
		System.out.println("factorial is "+fact);
	}
	public static void main(String args[]){
		// виклик методу анонімним об’єктом
		new Calculation().fact(5);
	}
}
```

Виведення на екран:

```
factorial is 120
```

Створення кількох об’єктів одного типу

Ми можемо створювати декілька об’єктів одного типу, як це робимо у випадку примітивів.

Ініціалізація примітивних змінних:

```
int a = 10, b = 20;
```

Ініціалізація посилальних змінних:

```
// створення двох об’єктів
Rectangle r1 = new Rectangle(), r2 = new Rectangle();
```

Розглянемо приклад:

```
// Програма для ілюстрації використання
// класу Rectangle, який містить дані про довжину та ширину
```

```
class Rectangle{
	int length;
	int width;
	void insert(int l, int w){
		length =l;
		width = w;
	}
	void calculateArea(){
		System.out.println(length*width);
	}
}
class TestRectangle2{
	public static void main(String args[]){
		// створення двох об’єктів
		Rectangle r1 = new Rectangle(), r2 = new Rectangle();
		r1.insert(11,5);
		r2.insert(3,15);
		r1.calculateArea();
		r2.calculateArea();
	}
}
```

Виведення на екран:

```
55
45
```

Приклад із реального світу: Account

Файл: TestAccount.java

```
// Програма для демонстрації роботи банківської системи,
// де ми вносимо та знімаємо суму зі свого рахунку
// Створення класу Account, який має
// методи deposit() та withdraw()
class Account{
	int acc_no;
	String name;
	float amount;
	// Метод ініціалізації об’єкта
	void insert(int a, String n, float amt){
		acc_no = a;
		name = n;
		amount = amt;
```

```
	}
	// Метод депозиту
	void deposit(float amt){
		amount = amount+amt;
		System.out.println(amt+" deposited");
	}
	// Метод зняття
	void withdraw(float amt){
		if(amount<amt){
			System.out.println("Insufficient Balance");
		}
		else{
			amount = amount-amt;
			System.out.println(amt+" withdrawn");
		}
	}
	// метод перевірки залишку на рахунку
	void checkBalance(){
		System.out.println("Balance is: "+amount);
	}
	// метод відображення значень об’єкта
	void display(){
		System.out.println(acc_no+" "+name+" "+amount);
	}
}
// Створення тестового класу для внесення та зняття суми
class TestAccount{
	public static void main(String[] args){
		Account a1 = new Account();
		a1.insert(832345,"Ankit",1000);
		a1.display();
		a1.checkBalance();
		a1.deposit(40000);
		a1.checkBalance();
		a1.withdraw(15000);
		a1.checkBalance();
	}
}
```

Виведення на екран:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Контрольні запитання

1. Що таке об'єкт?
2. Що таке клас?
3. Що таке поле?
4. Що таке метод?
5. Яке призначення ключового слова new?
6. Чи може метод main() знаходитись всередині чи поза класом? Наведіть приклади.
7. Які є способи ініціалізації об'єкта в Java? Наведіть приклади.
8. Які існують способи створення об'єкта в Java?
9. Що таке анонімний об'єкт? Яке його призначення?

ЛЕКЦІЯ 3. МЕТОД У JAVA

План

1. Заздалегідь визначений метод
2. Метод, визначений користувачем
3. Статичний метод
4. Метод екземпляра
5. Абстрактний метод

Загалом, *метод* – це спосіб виконання певного завдання. Подібним чином *метод у Java* – це сукупність інструкцій, які виконують певне завдання. Він забезпечує повторне використання коду. Ми також можемо легко змінювати код за допомогою *методів.* У цій лекції ми дізнаємося, що таке метод у Java, види методів, оголошення методів і як викликати метод у Java.

Що таке метод у Java?

Метод – це блок коду або набору інструкцій, які згруповані для виконання певного завдання або операції. Його використовують для досягнення *повторного використання* коду. Ми пишемо метод один раз і використовуємо його багато разів. Нам не потрібно писати код знову і знову. Метод також забезпечує *легку зміну* та *читаність* коду,

просто додаючи або видаляючи частину коду. Метод виконується лише тоді, коли ми його викликаємо.

Найважливішим методом в Java є метод main().

Декларація методу

Декларація методу містить інформацію про атрибути методу, такі як видимість, тип повернення, ім’я та аргументи. Він містить шість компонентів, які відомі як *заголовок методу*, як показано на наступному рисунку.

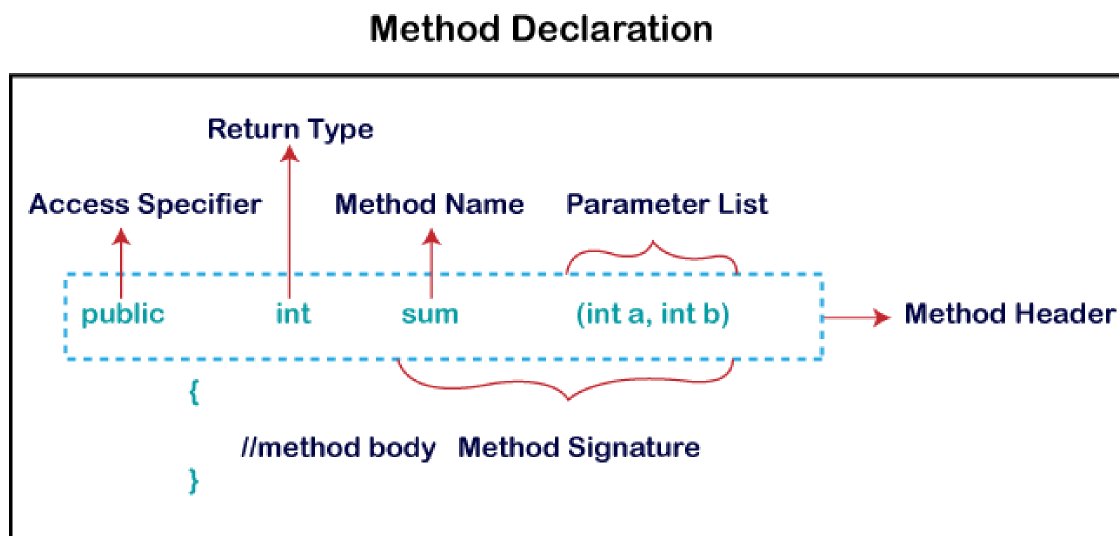

Рис. 8. Декларація методу

Заголовок методу: він містить назву *методу* та *список параметрів.*

Специфікатор доступу: Специфікатор або модифікатор доступу – це тип доступу методу. Він визначає видимість методу. Java пропонує *чотири* типи специфікатора доступу, а саме:

Public: метод доступний для всіх класів, коли ми використовуємо відкритий специфікатор у своїй програмі.

Private: коли ми використовуємо специфікатор приватного доступу, метод доступний лише у класах, у яких він визначений.

Protected: коли ми використовуємо специфікатор захищеного доступу, метод доступний у межах одного пакету або підкласів у іншому пакеті.

Default: коли ми не використовуємо жодного специфікатора доступу в оголошенні методу, Java використовує специфікатор доступу за замовчуванням. Його видно лише з того самого пакета.

Тип повернення: це тип даних, який повертає метод. Він може мати примітивний тип даних, об'єкт, колекцію, void тощо. Якщо метод нічого не повертає, ми використовуємо ключове слово void.

Назва методу: це унікальна назва, яка використовується для визначення назви методу. Вона повинна відповідати функціоналу методу. Наприклад, якщо ми створюємо метод для віднімання двох чисел, ім'я методу повинно бути subtraction(). Метод викликається за його іменем.

Список параметрів: це список параметрів, розділених комою та укладених у пару дужок. Він містить тип даних та назву змінної. Якщо метод не має параметра, залиште дужки порожніми.

Тіло методу: це частина декларації методу. Воно містить усі дії, які необхідно виконати. Тіло методу вкладають у пару фігурних дужок.

Іменування методу

Визначаючи метод, пам'ятайте, що ім'я методу має бути *дієсловом* і починатися з *малої* літери. Якщо назва методу містить більше двох слів, перше ім'я має бути дієсловом, за яким слідує прикметник або іменник. У назві методу з кількох слів перша буква кожного слова має бути *написана великими літерами*, крім першого слова. Наприклад:

назва методу з одного слова: sum(), area();

назва методу з кількох слів: areaOfCircle(), stringComparision().

Допускається, що метод має таку ж назву, що й інший метод у тому ж класі, це називають *перевантаження методу.*

Типи методів

У Java є два типи методів:

– заздалегідь визначений метод;

– метод, визначений користувачем.

Заздалегідь визначений метод

У Java заздалегідь визначений метод – це метод, який уже визначений у бібліотеках класів Java. Його називають *стандартним методом бібліотеки* або *вбудованим методом*. Ми можемо безпосередньо використовувати ці методи, просто викликавши їх у програмі в будь-який момент. Прикладами заздалегідь визначених методів є length(), equals(), compareTo(), sqrt() тощо. Коли ми викликаємо будь-який із заздалегідь визначених методів у нашій

програмі, у фоновому режимі працює ряд кодів, пов’язаних із відповідним методом, який зберігається в бібліотеці.

Кожен заздалегідь визначений метод визначається всередині класу. Наприклад, метод print() визначений у класі java.io.PrintStream. Він друкує рядок, яке ми пишемо всередині методу. Наприклад, print("Java") виводить на консоль рядок "Java".

Давайте розглянемо приклад наперед визначеного методу.

Demo.java

```
public class Demo {
        public static void main(String[] args){
                // використовується метод max() класу Math
                System.out.print("The maximum number is: " +
        Math.max(9,7));
                }
}
```

Виведення на екран:

```
The maximum number is: 9
```

У наведеному вище прикладі ми використовували три попередньо визначені методи main(), print() і max(). Ми використовували ці методи безпосередньо без оголошення, оскільки вони заздалегідь визначені. Метод print() – це метод класу PrintStream, який друкує результат на консолі. Метод max() – це метод класу Math, який повертає найбільше з двох чисел.

Ми також можемо побачити опис методу будь-якого попередньо визначеного методу за допомогою посилання https://docs.oracle.com/. Коли ми переходимо за посиланням і бачимо опис методу max(), ми бачимо наступне:

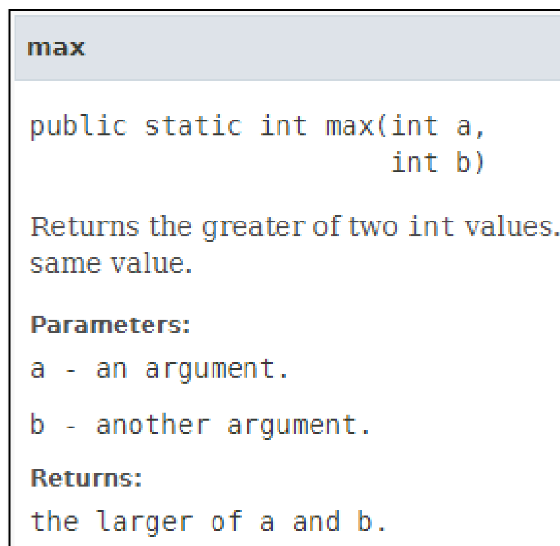

Рис. 9. Опис методу max()

У наведеному вище описі методу ми бачимо, що декларація методу має специфікатор доступу *public,* модифікатор доступу *static,* тип повернення *int*, ім'я методу max(), список параметрів (int a, int b). У наведеному вище прикладі замість визначення методу ми просто викликали метод. Це перевага заздалегідь визначеного методу. Це робить програмування менш складним.

Так само ми можемо побачити опис методу print().

Метод, визначений користувачем

Метод, написаний користувачем або програмістом, називають *методом, який визначений користувачем.* Ці методи визначаються відповідно до вимог.

Як створити метод, визначений користувачем

Давайте створимо метод, який перевірятиме число, чи воно парне чи непарне. Спочатку ми визначимо метод.

```
// визначений користувачем метод
public static void findEvenOdd(int num) {
	// тіло методу
	if (num%2==0)
		System.out.println(num+" is even");
	else
		System.out.println(num+" is odd");
}
```

Ми визначили вищезгаданий метод з назвою findEvenOdd(). Він має параметр *Num* типу int. Метод не повертає жодного значення, тому ми використали void. Тіло методу містить кроки для перевірки парного чи непарного числа. Якщо число парне, воно друкує *is even***,** інакше друкує число *is odd*.

Як викликати метод, визначений користувачем

Після того, як ми визначили метод, його слід викликати. Виклик методу в програмі простий. Коли ми викликаємо визначений користувачем метод, програма керування переходить на викликаний метод.

```
import java.util.Scanner;
public class EvenOdd {
	public static void main (String args[]) {
		// створення об'єкта класу Scanner
		Scanner scan=new Scanner(System.in);
```

```
            System.out.print("Enter the number: ");
            // зчитування значення від користувача
            int num=scan.nextInt();
            // виклик методу
            findEvenOdd(num);
        }
}
```

У наведеному вище фрагменті коду, як тільки компілятор досягає рядка findEvenOd(num), керування переходить до методу і відповідно видає результат.

Давайте об'єднаємо обидва фрагменти кодів в одній програмі і виконаємо її.

EvenOdd.java

```
import java.util.Scanner;
public class EvenOdd {
    public static void main (String args[]) {
        // створення об'єкта класу Scanner
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        // зчитування значення від користувача
        int num=scan.nextInt();
        // виклик методу
        findEvenOdd(num);
    }
    // визначений користувачем метод
    public static void findEvenOdd(int num) {
        // тіло методу
        if(num%2==0)
            System.out.println(num+" is even");
        else
            System.out.println(num+" is odd");
    }
}
```

Виведення на екран:

```
Enter the number: 12
12 is even
```

Виведення на екран:

```
Enter the number: 99
99 is odd
```

Давайте розглянемо іншу програму, яка повертає значення під час виклику методу.

У наступній програмі ми визначили метод із назвою add(), який обчислює суму двох чисел. Він має два параметри n1 і n2 цілого типу. Значення n1 та n2 відповідають значенню a та b відповідно. Тому метод заходить суму значень a та b, зберігає його у змінній s та повертає суму.

Addition.java

```
public class Addition {
    public static void main(String[] args) {
        int a = 19;
        int b = 5;
        // виклик методу
        int c = add(a, b);     // a і b – фактичні параметри
        System.out.println("The sum of a and b is = " + c);
    }
    // визначений користувачем метод
    public static int add(int n1, int n2) {
        // n1 і n2 – це формальні параметри
        int s;
        s=n1+n2;
        return s; // повернення суми
    }
}
```

Виведення на екран:

```
The sum of a and b is= 24
```

Статичний метод

Метод з ключовим словом *static* називають *статичним*. Іншими словами, метод, який належить класу, а не екземпляру класу, – це статичний метод. Ми також можемо створити статичний метод, використовуючи ключове слово *static* перед назвою методу.

Основною перевагою статичного методу є те, що ми можемо викликати його без створення об’єкта. Він може отримати доступ до статичних полів даних, а також змінити їх значення. Він використовується для створення методу екземпляра. Він викликається за допомогою назви класу. Найкращим прикладом статичного методу є метод main().

Приклад статичного методу

Display.java

```
public class Display {
    public static void main(String[] args) {
        show();
    }
    static void show() {
        System.out.println("It is an example of static method.");
    }
}
```

Виведення на екран:

```
It is an example of a static method.
```

Метод екземпляра

Метод класу називають *методом екземпляра*. Це *нестатичний* метод, визначений у класі. Перш ніж викликати метод екземпляра, необхідно створити об’єкт його класу. Давайте розглянемо приклад методу екземпляра.

InstanceMethodExample.java

```
public class InstanceMethodExample {
    public static void main(String [] args) {
        // Створення об’єкта класу
        InstanceMethodExample obj = new
InstanceMethodExample();
        // виклик методу екземпляра
        System.out.println("The sum is: "+obj.add(12, 13));
    }
    int s;
    // визначений користувачем метод, оскільки
    // ми не використовували ключове слово static
    public int add(int a, int b) {
        s = a + b;
        // повернення суми
        return s;
    }
}
```

Виведення на екран:

```
The sum is: 25
```

Існує два типи методів екземпляра:

- метод доступу;
- метод модифікатора.

Методи доступу: методи, що зчитують змінні екземпляра. Ми можемо легко їх ідентифікувати, оскільки методи мають префікс слова *get*. Їх називають *гетерами*. Вони повертають значення приватного поля. Вони використовуються для отримання значення приватного поля.

Приклад

```
public int getId() {
    return Id;
}
```

Методи модифікатора: методи зчитують змінні екземпляра, а також змінюють їх значення. Ми можемо легко їх ідентифікувати, оскільки методи мають префікс слова *set*. Їх називають *сетерами* або *модифікаторами*. Вони нічого не повертають. Вони приймають параметри того самого типу, що й тип поля. Вони використовуються для встановлення значення приватного поля.

Приклад

```
public class Student {
    private int roll;
    private String name;
    // метод доступу
    public int getRoll() {
        return roll;
    }
    // метод модифікатора
    public void setRoll(int roll){
        this.roll = roll;
    }
    public String getName(){
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void display() {
        System.out.println("Roll no.: "+roll);
        System.out.println("Student name: "+name);
    }
}
```

Абстрактний метод

Метод, який не має тіла методу, називається *абстрактним методом.* Іншими словами, метод без реалізації є *абстрактним методом*. Він завжди оголошується в *абстрактному класі*. Це означає, що сам клас повинен бути абстрактним, якщо він має абстрактний метод. Для створення абстрактного методу ми використовуємо ключове слово *abstract*.

Синтаксис

```
abstract void method_name();
```

Приклад абстрактного методу

Demo.java

```
// абстрактний клас
abstract class Demo {
    // анотація абстрактного методу
    abstract void display();
}
public class MyClass extends Demo {
    // абстрактний метод
    void display() {
        System.out.println("Abstract method?");
    }
    public static void main(String args[]) {
        // створення об’єкта абстрактного класу
        Demo obj = new MyClass();
        // виклик абстрактного методу
        obj.display();
    }
}
```

Виведення на екран:

```
Abstract method...
```

Це метод, який повертає об’єкт класу, до якого він належить. Усі статичні методи є абстрактними. Наприклад, NumberFormat obj = NumberFormat.getNumberInstance();

Контрольні запитання

1. Що таке метод?
2. Яку інформацію містить декларація методу в Java?
3. Що таке специфікатор доступу?
4. Які специфікатори доступу є в Java?

5. Які правила іменування методу в Java?

6. Що таке стандартний метод? Наведіть приклади.

7. Що розуміють під методом, визначеним користувачем? Наведіть приклади їх визначення та використання.

8. Що таке статичний метод?

9. Що таке метод екземпляра?

10. Що таке методи доступу? Наведіть приклади.

11. Що таке методи модифікатори? Наведіть приклади.

12. Який метод називають абстрактним?

ЛЕКЦІЯ 4. КОНСТРУКТОРИ

План

1. Конструктор Java за замовчуванням
2. Перевантаження конструктора в Java
3. Відмінність між конструктором та методом у Java
4. Конструктор копіювання Java
5. Копіювання значень без конструктора

У Java *конструктор* – це блок кодів, подібний до методу. Він викликається, коли створюється екземпляр класу. Під час виклику конструктора для об’єкта виділяється пам’ять.

Це особливий тип методу, який використовується для ініціалізації об’єкта.

Щоразу, коли об’єкт створюється за допомогою ключового слова new(), викликається принаймні один конструктор.

Викликається конструктор за замовчуванням, якщо в класі немає конструктора. У такому випадку компілятор Java надає конструктор за замовчуванням.

У Java є два типи конструкторів: конструктор без аргументів та конструктор з параметрами.

Примітка. Немає необхідності писати конструктор для класу. Причиною цього є те, що компілятор Java створює конструктор за замовчуванням, якщо у вашому класі його немає.

Правила створення конструктора в Java

Для конструктора визначено наступні правила.

1. Ім’я конструктора має збігатися з назвою його класу.
2. Конструктор не повинен мати явного типу повернення.
3. Конструктор у Java не може бути абстрактним, статичним, фінальним та синхронізованим.

Примітка. Ми можемо використовувати модифікатори доступу під час оголошення конструктора. Вони контролюють створення об’єкта. Іншими словами, у нас може бути приватний, захищений, загальнодоступний конструктор або конструктор за замовчуванням.

Типи конструкторів Java

У Java є два типи конструкторів:

– конструктор за замовчуванням (конструктор без параметрів);

– конструктор з параметрами.

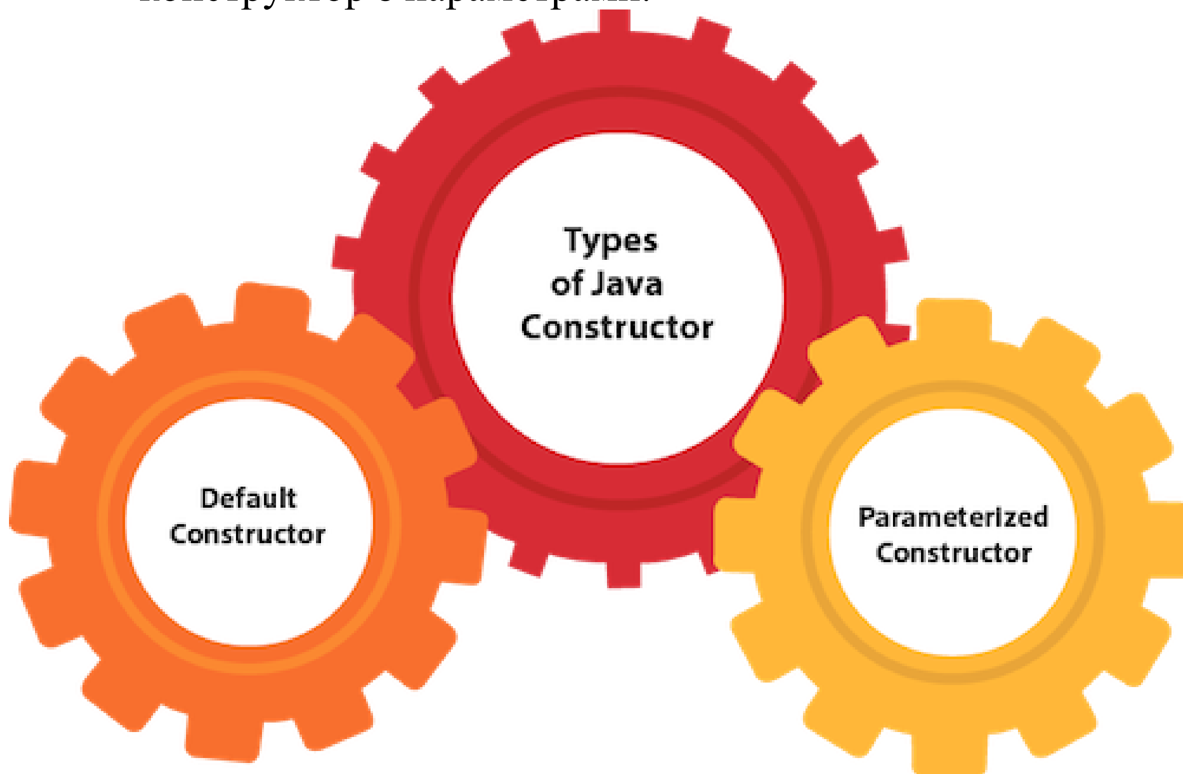

Рис. 10. Типи конструкторів у Java

Конструктор Java за замовчуванням

Конструктор називається "Конструктор за замовчуванням", коли у нього немає жодного параметра.

Синтаксис конструктора за замовчуванням:

```
<ім’я_класу> () {
}
```

Приклад конструктора за замовчуванням

У цьому прикладі ми створюємо конструктор без параметрів у класі Bike. Його викликають під час створення об’єкта.

```
// Програма Java для створення та виклику конструктора за замовчуванням
class Bike1 {
```

```
        // створення конструктора за замовчуванням
        Bike1() {
            System.out.println("Bike is created");
        }
        // основний метод
        public static void main(String args[]){
            // виклик конструктора за замовчуванням
            Bike1 b = new Bike1();
        }
    }
```

Виведення на екран:

```
Bike is created
```

Правило: якщо в класі немає конструктора, компілятор автоматично створює конструктор за замовчуванням.

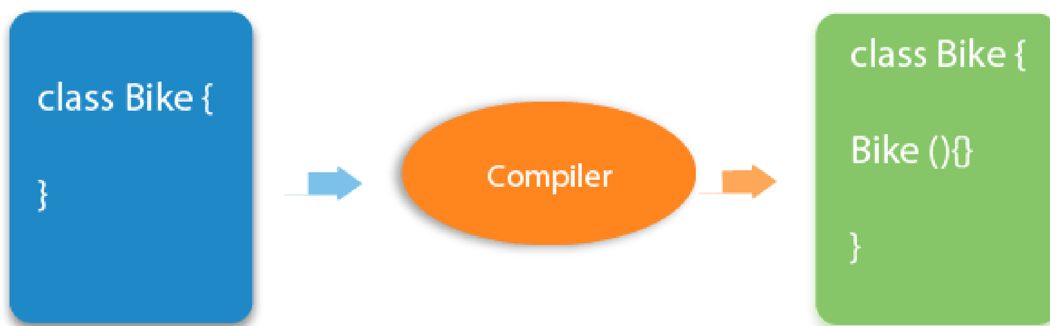

Рис. 11. Автоматичне створення компілятором конструктора за замовчуванням

Яке призначення конструктора за замовчуванням?

Конструктор за замовчуванням використовується для надання об’єкту значень за замовчуванням, таких як 0, null тощо, залежно від типу.

Приклад конструктора за замовчуванням, який відображає значення за замовчуванням

```
    // Давайте розглянемо інший приклад конструктора за
    // замовчуванням, який відображає значення за замовчуванням
class Student3{
    int id;
    String name;
    // метод відображення значення id та name
    void display(){
```

```
        System.out.println(id+" "+name);
    }
    public static void main(String args[]){
        // створення об’єктів
        Student3 s1=new Student3();
        Student3 s2=new Student3();
        // відображення значень об’єкта
        s1.display();
        s2.display();
    }
}
```

Виведення на екран:

```
0 null
0 null
```

Пояснення: у наведеному вище класі ми не створюємо жодного конструктора, тому компілятор надає вам конструктор за замовчуванням. Тут 0 і null надаються конструктором за замовчуванням.

Конструктор з параметрами

Конструктор з певною кількістю параметрів називається *конструктором з параметрами*.

Навіщо використовувати конструктор з параметрами?

Конструктор з параметрами використовується для надання різних значень різним об’єктам. Однак ми також можемо вказати ті самі значення.

Приклад конструктора з параметрами

У цьому прикладі ми створили конструктор класу Student, який має два параметри. У конструкторі ми можемо мати будь-яку кількість параметрів.

```
// Програма Java для демонстрації використання
// конструктора з параметрами
class Student4 {
    int id;
    String name;
    // створення конструктора з параметрами
    Student4(int i, String n) {
        id = i;
        name = n;
    }
```

```
    // метод відображення значень
    void display() {
        System.out.println(id+" "+name);
    }
    public static void main(String args[]) {
        // створення об’єктів та передавання значень
        Student4 s1 = new Student4(111,"Karan");
        Student4 s2 = new Student4(222,"Aryan");
        // виклик методу для відображення значень об’єкта
        s1.display();
        s2.display();
    }
}
```

Виведення на екран:

```
111 Karan
222 Aryan
```

Перевантаження конструктора в Java

У Java конструктор подібний до методу, але без типу повернення. Його також можна перевантажити, як методи Java.

Перевантаження конструктора в Java – це техніка, що містить кілька конструкторів із різними списками параметрів. Вони розташовані таким чином, що кожен конструктор виконує інше завдання. Вони розрізняються компілятором за кількістю параметрів у списку та їх типами.

Приклад перевантаження конструктора

```
// Програма для перевантаження конструкторів
class Student5{
    int id;
    String name;
    int age;
    // створення конструктора з двома параметрами
    Student5(int i, String n){
        id = i;
        name = n;
    }
    // створення конструктора з трьома параметрами
    Student5(int i, String n, int a){
```

```
            id = i;
            name = n;
            age=a;
        }
        void display(){
            System.out.println(id+" "+name+" "+age);
        }
          public static void main(String args[]){
            Student5 s1 = new Student5(111,"Karan");
            Student5 s2 = new Student5(222,"Aryan",25);
            s1.display();
            s2.display();
      }
}
```

Виведення на екран:

```
111 Karan 0
222 Aryan 25
```

Відмінність між конструктором та методом у Java

Між конструкторами та методами є багато відмінностей. Вони наведені нижче.

Конструктор	**Метод**
Конструктор використовується для ініціалізації стану об’єкта.	Метод використовується для виявлення поведінки об’єкта.
Конструктор не повинен мати тип повернення.	Метод повинен мати тип повернення.
Конструктор викликається неявно.	Метод викликається явно.
Компілятор Java надає конструктор за замовчуванням, якщо у немає жодного конструктора в класі.	Метод у жодному разі не надається компілятором.
Ім’я конструктора має бути таким же, як і ім’я класу.	Назва методу може збігатися, а може і не збігатися з назвою класу.

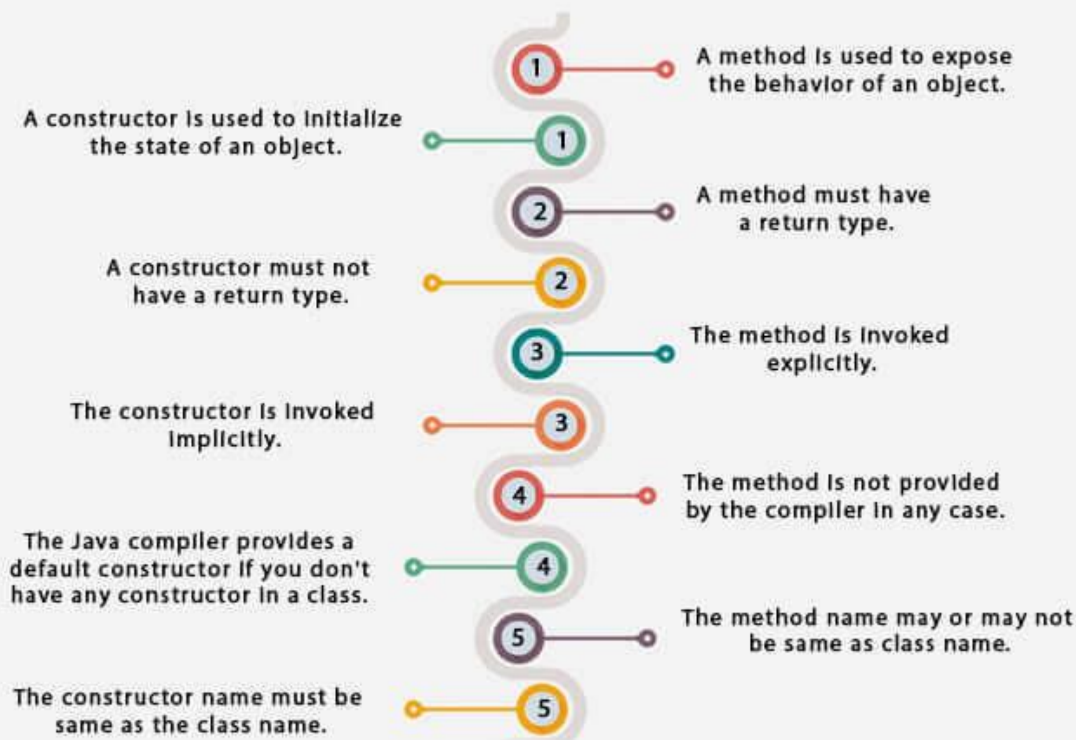

Рис. 12. Відмінність між конструктором та методом у Java

Конструктор копіювання у Java

У Java немає конструктора копіювання. Однак ми можемо копіювати значення з одного об’єкта в інший, аналогічно конструктору копіювання в C ++.

Існує багато способів копіювання значень одного об’єкта в інший у Java, а саме:

– за конструктором;
– присвоюючи значення одного об’єкта іншому;
– методом clone() класу Object.

У цьому прикладі ми збираємося скопіювати значення одного об’єкта в інший за допомогою конструктора Java.

```
// Програма Java для ініціалізації значень
// від одного об’єкта
class Student6{
    int id;
    String name;
```

```
    // конструктор для ініціалізації цілого числа та рядка
    Student6(int i, String n){
        id = i;
        name = n;
    }
    // конструктор для ініціалізації іншого об’єкта
    Student6(Student6 s){
        id = s.id;
        name =s.name;
    }
    void display(){
        System.out.println(id+" "+name);
    }
    public static void main(String args[]){
        Student6 s1 = new Student6(111,"Karan");
        Student6 s2 = new Student6(s1);
        s1.display();
        s2.display();
   }
}
```

Виведення на екран:

```
111 Karan
111 Karan
```

Копіювання значень без конструктора

Ми можемо скопіювати значення одного об’єкта в інший, призначаючи значення об’єктів іншому об’єкту. У цьому випадку немає необхідності створювати конструктор.

```
class Student7{
    int id;
    String name;
    Student7(int i, String n){
        id = i;
        name = n;
    }
    Student7(){
    }
    void display(){
```

```
        System.out.println(id+" "+name);
    }
      public static void main(String args[]){
        Student7 s1 = new Student7(111,"Karan");
        Student7 s2 = new Student7();
        s2.id=s1.id;
        s2.name=s1.name;
        s1.display();
        s2.display();
   }
}
```

Виведення на екран:

```
111 Karan
111 Karan
```

Чи повертає конструктор якесь значення?

Так, це поточний екземпляр класу (Ви не можете використовувати тип повернення, але він повертає значення).

Чи може конструктор виконувати інші завдання замість ініціалізації?

Такі, як створення об'єктів, запуск потоку, виклик методу тощо. Ви можете виконувати будь-які операції в конструкторі так само, як і в методі.

Чи є клас конструктора в Java?

Так.

Яке призначення класу конструктора?

Java надає клас Constructor, який можна використовувати для отримання внутрішньої інформації конструктора в класі. Він міститься в пакеті java.lang.reflect.

Контрольні запитання

1. Що таке конструктор?
2. Яке призначення конструкторів в Java?
3. Чи можна використовувати конструктор для створення об'єктів? Наведіть приклади.
4. Чи можна використовувати конструктор для ініціалізації об'єктів? Наведіть приклади.
5. Яка відмінність між методом і конструктором в Java?
6. Що таке конструктор за замовчуванням?
7. Що таке конструктор з параметрами? Наведіть приклади.

8. Що таке перевантаження конструктора? Наведіть приклади.

ЛЕКЦІЯ 5. КЛЮЧОВЕ СЛОВО STATIC У JAVA

План

1. Статичне поле
2. Статичний метод
3. Статичний блок

Ключове слово *static* в Java використовується головним чином для управління пам'яттю. Ми можемо застосувати ключове слово *static* з полями, методами, блоками та вкладеними класами. Ключове слово *static* належить до класу, а не до екземпляру класу.

Статичним може бути:
– поле (змінна класу);
– метод;
– блок;
– вкладений клас.

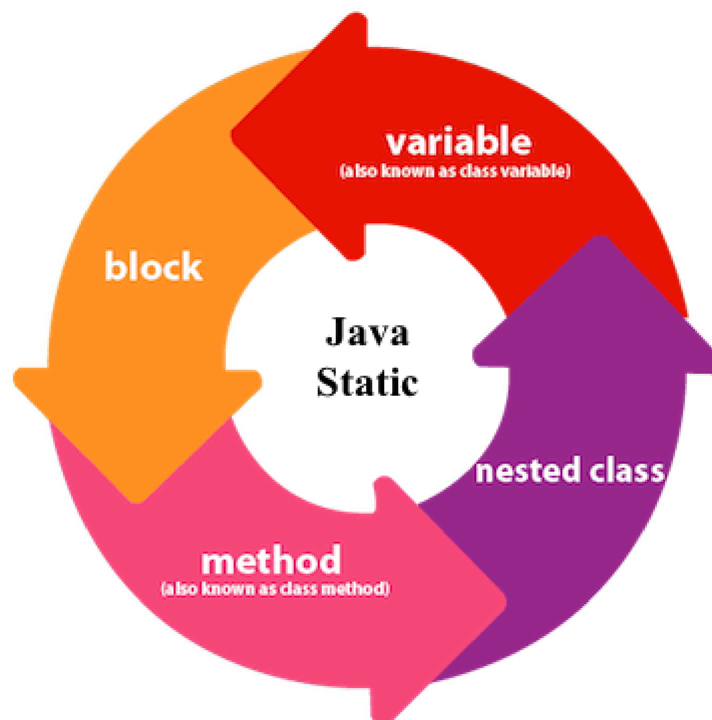

Рис. 13. Ключове слово *static* в Java

Статичне поле в Java

Якщо ви оголошуєте будь-яку змінну із використанням ключового слова *static*, тоді це буде статична змінна.

– Статична змінна може бути використана для посилання на загальну властивість усіх об'єктів (яка не є унікальною для кожного об'єкта), наприклад, на назву компанії співробітників, назву студентів коледжу тощо.

– Статична змінна отримує пам'ять лише один раз у зоні класу під час завантаження класу.

Переваги статичної змінної

Це робить *пам'ять* вашої програми *ефективною* (тобто економить пам'ять).

Розуміння проблеми без статичної змінної

```
class Student {
    int rollno;
    String name;
    String college="ITS";
}
```

Припустимо, у коледжі є 500 студентів, тепер усі поля даних екземплярів будуть отримувати пам'ять щоразу, коли створюється об'єкт. Усі студенти мають свій унікальний номер та ім'я, що добре в такому випадку. Тут "коледж" відноситься до спільної власності всіх об'єктів. Якщо ми зробимо його статичним, це поле отримає пам'ять лише один раз.

Статична властивість Java є спільною для всіх об'єктів.

Приклад статичної змінної

```
// Програма Java для демонстрації використання
// статичної змінної
class Student{
    int rollno; // змінна екземпляра
    String name;
    static String college ="ITS"; // статична змінна
    // конструктор
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    // метод відображення значень
    void display (){
        System.out.println(rollno+" "+name+" "+college);
    }
}
```

```
// Тестовий клас для відображення значень об’єктів
public class TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        // ми можемо змінити коледж усіх об’єктів
        // одним рядком коду
        // Student.college = "BBDIT";
        s1.display();
        s2.display();
 }
}
```

Виведення на екран:

```
111 Karan ITS
222 Aryan ITS
```

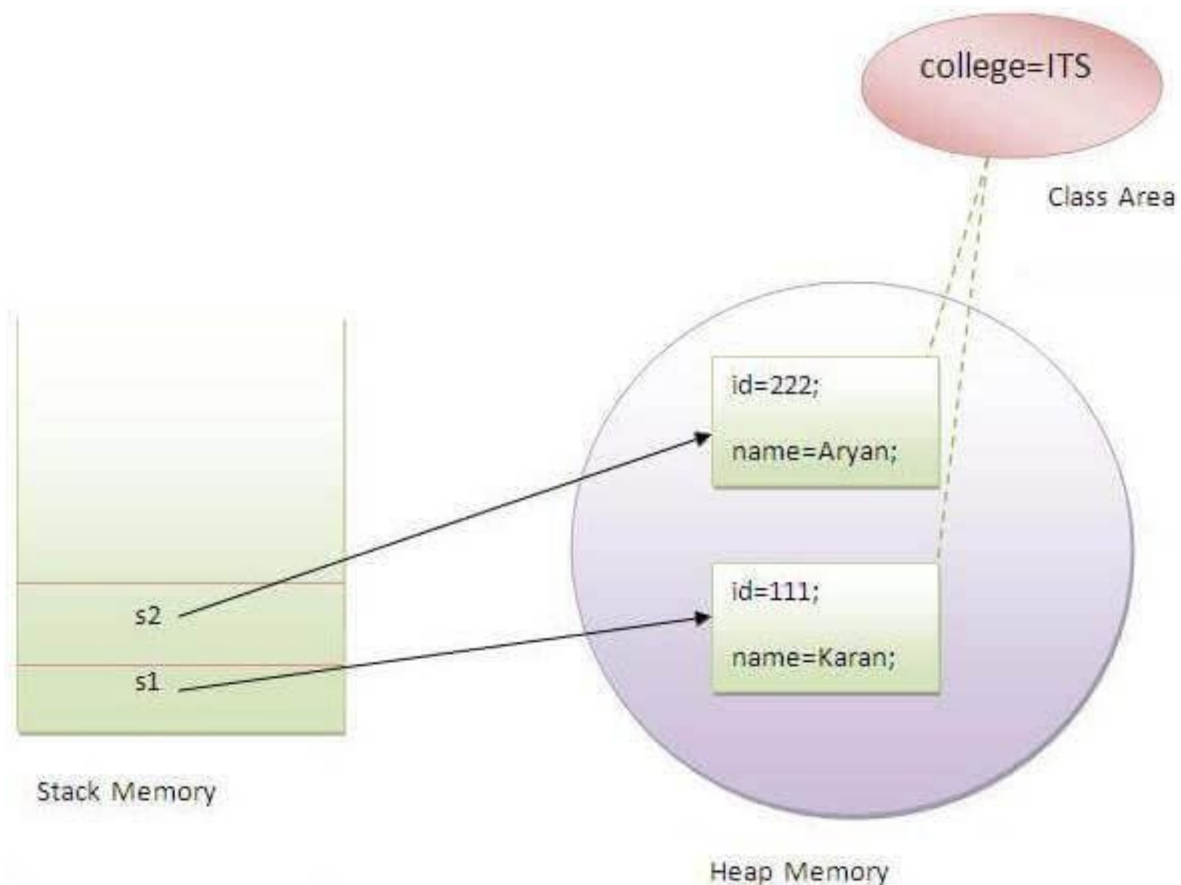

Рис. 14. Приклад використання статичної змінної

Програма лічильника без статичної змінної

У цьому прикладі ми створили змінну екземпляра з іменем count, яка збільшується у конструкторі. Оскільки змінна екземпляра отримує

пам’ять під час створення об’єкта, кожен об’єкт матиме копію змінної екземпляра. Якщо його збільшити, він не відображатиме інші об’єкти. Отже, кожен об’єкт матиме значення 1 у змінній count.

```
// Програма Java для демонстрації використання
// змінної екземпляра, які отримують пам’ять
// кожного разу, коли ми створюємо об’єкт класу
class Counter{
    // буде отримувати пам’ять кожного разу при створенні екземпляра
    int count = 0;
    Counter(){
        count++; // приріст значення
        System.out.println(count);
    }

    public static void main(String args[]){
        // Створення об’єктів
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();
    }
}
```

Виведення на екран:

```
1
1
1
```

Програма лічильника з статичним полем

Як ми вже згадували вище, статична змінна отримає пам’ять лише один раз, якщо будь-який об’єкт змінить значення статичної змінної, він збереже своє значення.

```
// Програма Java для ілюстрації використання статичної
// змінної, яка   має спільний доступ до всіх об’єктів
class Counter2{
    // отримає пам’ять лише один раз і збереже своє значення
    static int count = 0;
    Counter2(){
        count++; // збільшення значення статичної змінної
        System.out.println(count);
```

```
    }
    public static void main(String args[]){
        // створення об'єктів
        Counter2 c1 = new Counter2();
        Counter2 c2 = new Counter2();
        Counter2 c3 = new Counter2();
    }
}
```

Виведення на екран:

```
1
2
3
```

Статичний метод у Java

Якщо ви застосуєте ключове слово *static* до будь-якого методу, це називається статичним методом.

– Статичний метод належить класу, а не об’єкту класу.

– Статичний метод можна викликати без необхідності створення екземпляра класу.

– Статичний метод може отримати доступ до статичного члена даних і може змінити його значення.

Приклад статичного методу

```
// Програма Java для демонстрації використання
// статичного методу
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    // статичний метод зміни значення статичної змінної
    static void change(){
        college = "BBDIT";
    }
    // конструктор для ініціалізації змінної
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    // метод відображення значень
```

```
        void display(){
            System.out.println(rollno+" "+name+" "+college);
        }
}
// Тестовий клас для створення та відображення значень об’єкта
public class TestStaticMethod{
    public static void main(String args[]){
        Student.change(); // виклик методу зміни
        // створення об’єктів
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        Student s3 = new Student(333,"Sonoo");
        // виклик методу відображення
        s1.display();
        s2.display();
        s3.display();
    }
}
```

Вихід:

```
111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT
```

Ще один приклад статичного методу, який виконує обчислення

```
// Програма Java для отримання куба даного числа
// за допомогою статичного методу
class Calculate{
    static int cube(int x){
        return x*x*x;
    }

    public static void main(String args[]){
        int result=Calculate.cube(5);
        System.out.println(result);
    }
}
```

Виведення на екран:

```
125
```

Обмеження для статичного методу

Існує два основних обмеження для статичного методу, а саме:

1) статичний метод не може використовувати нестатичне поле або безпосередньо викликати нестатичний метод;

2) this і super не можна використовувати в статичному контексті.

```
class A{
	int a=40; // нестатичний
	public static void main(String args[]){
		System.out.println(a);
	}
}
```

Виведення на екран:

```
Compile Time Error
```

Чому основний метод у Java є статичним?

Відповідь: Це тому, що об'єкту не потрібно викликати статичний метод. Якби це був нестатичний метод, JVM спочатку створив би об'єкт, а потім викликав метод main(), що призвело б до проблеми додаткового виділення пам'яті.

Статичний блок у Java

– Використовується для ініціалізації статичного поля.

– Він виконується перед основним методом під час завантаження класу.

Приклад статичного блоку

```
class A2{
	static{
		System.out.println("static block is invoked");
	}
	public static void main(String args[]){
		System.out.println("Hello main");
	}
}
```

Виведення на екран:

```
static block is invoked
Hello main
```

Чи можна виконувати програму без методу main()?

Відповідь: ні, одним із способів був статичний блок, але це було можливо до JDK 1.6. Починаючи з JDK 1.7, неможливо виконати клас Java без методу main.

```
class A3{
```

```
        static {
                System.out.println("static block is invoked");
                System.exit(0);
        }
}
```

Виведення на екран:

```
Error: Main method not found in class A3, please define the main method as:
    public static void main(String[] args)
    or a JavaFX application class must extend javafx.application.Application
```

Контрольні запитання

1. Яке призначення ключового слова *static* у Java?
2. Що таке статичне поле? Яке його призначення? Наведіть приклади.
3. Що таке статичний метод у Java? Для чого він використовується? Наведіть приклади.
4. Чому основний метод у Java є статичним?
5. Що таке статичний блок у Java? Яке його призначення?
6. Чи можна виконувати програму без методу main()?
7. Що таке перевантаження конструктора? Наведіть приклади.

ЛЕКЦІЯ 6. КЛЮЧОВЕ СЛОВО THIS У JAVA

1. this: посилання на змінну екземпляра поточного класу
2. this: виклик поточного методу класу
3. this(): виклик конструктора поточного класу
4. this: передати як аргумент у методі
5. this: передати як аргумент у виклик конструктора
6. this: повернення поточного екземпляра класу

Ключове слово *this* широко використовується у Java. У Java це *змінна,* яка посилається на поточний об'єкт.

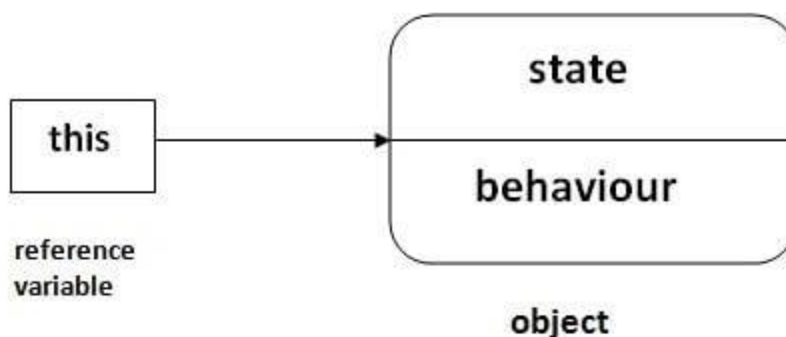

Рис. 15. Ключове слово *this* у Java

Використання ключового слова this у Java

Існує 6 випадків використання ключового слова *this* у Java:

– *this* може бути використано для посилання на змінну екземпляра поточного класу;

– *this* може бути використано для виклику методу поточного класу (неявно);

– this() можна використовувати для виклику конструктора поточного класу;

– *this* може бути передано як аргумент у виклику методу;

– *this* може бути передано як аргумент у виклику конструктора;

– *this* може бути використано для повернення поточного екземпляра класу з методу.

Пропозиція: якщо ви новачок у Java, шукайте лише три варіанти використання цього ключового слова.

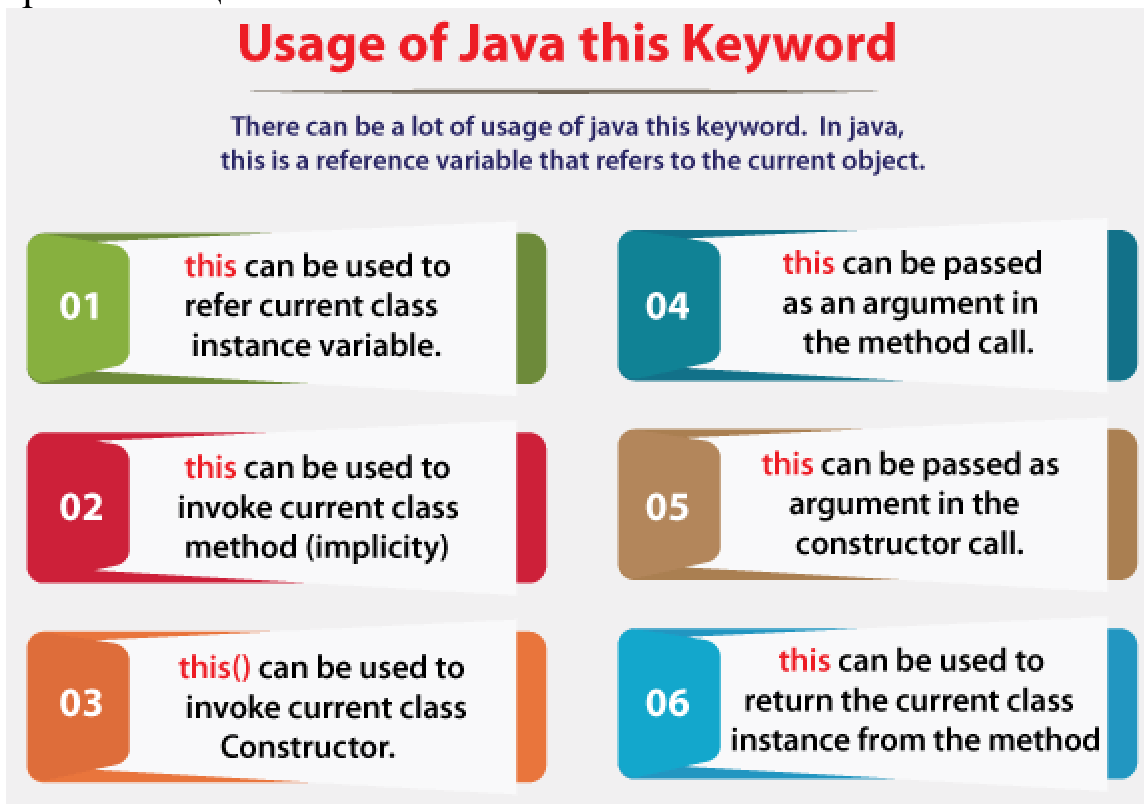

Рис. 15. Використання ключового слова *this* у Java

this: посилання на змінну екземпляра поточного класу

Це ключове слово можна використовувати для посилання на змінну екземпляра поточного класу. Якщо між змінними екземпляра

та параметрами існує неоднозначність, ключове слово *this* вирішує проблему двозначності.

Розуміння проблеми без цього ключового слова

Давайте розберемося в проблемі, якщо ми не використаємо це ключове слово на прикладі, наведеному нижче:

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno, String name, float fee){
        rollno = rollno;
        name = name;
        fee = fee;
    }
    void display(){
        System.out.println(rollno+" "+name+" "+fee);
    }
}
class TestThis1{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

Виведення на екран:

```
0 null 0.0
0 null 0.0
```

У наведеному вище прикладі параметри (формальні аргументи) та змінні екземпляра збігаються. Отже, ми використовуємо ключове слово *this* для розрізнення локальної змінної та змінної екземпляра.

Рішення вищезазначеної проблеми за допомогою ключового слова this

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno, String name, float fee){
```

```
        this.rollno = rollno;
        this.name = name;
        this.fee = fee;
    }
    void display(){
        System.out.println(rollno+" "+name+" "+fee);
    }
}
class TestThis2{
    public static void main(String args[]){
        Student s1 = new Student(111,"ankit",5000f);
        Student s2 = new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

Виведення на екран:

```
111 ankit 5000.0
112 sumit 6000.0
```

Якщо локальні змінні (формальні аргументи) та змінні екземпляра різні, немає необхідності використовувати ключове слово this, як у такій програмі:

Програма, де ключове слово this не потрібно

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int r, String n, float f){
        rollno = r;
        name = n;
        fee = f;
    }
    void display(){
        System.out.println(rollno+" "+name+" "+fee);
    }
}
class TestThis3{
    public static void main(String args[]){
        Student s1 = new Student(111,"ankit",5000f);
```

```
        Student s2 = new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

Виведення на екран:

```
111 ankit 5000.0
112 sumit 6000.0
```

Слід краще підходити до використання змістовних імен для змінних. Тому, коли ми використовуємо одну і ту ж назву для змінних і параметрів екземпляра в режимі реального часу, то завжди використовуємо ключове слово *this*.

this: виклик методу поточного класу

Ви можете викликати метод поточного класу, використовуючи ключове слово *this*. Якщо ви не використовуєте це ключове слово, компілятор автоматично додає це ключове слово під час виклику методу. Розглянемо приклад

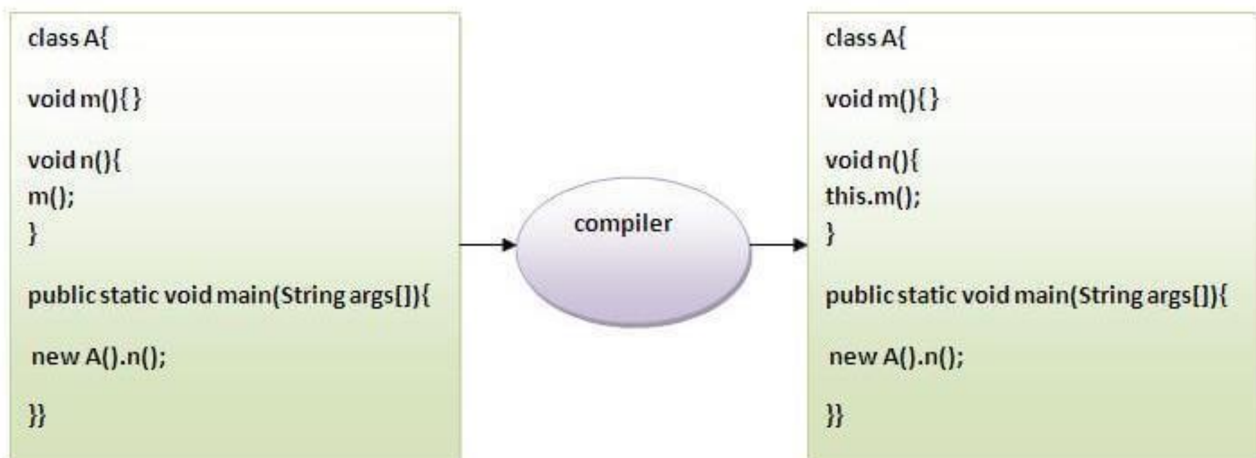

Рис. 17. Виклик поточного методу класу

```
class A{
    void m(){
        System.out.println("hello m");
    }
    void n(){
        System.out.println("hello n");
        // m(); // те саме, що this.m()
        this.m();
```

```
        }
    }
    class TestThis4{
        public static void main(String args[]){
            A a = new A();
            a.n();
        }
    }
```

Виведення на екран:

```
hello n
hello m
```

this(): виклик конструктора поточного класу

Виклик конструктора this() можна використовувати для виклику конструктора поточного класу. Він використовується для повторного використання конструктора. Іншими словами, він використовується для ланцюгового конструктора.

Виклик конструктора за замовчуванням з конструктора з параметрами:

```
class A{
    A(){
        System.out.println("hello a");
    }
    A(int x){
        this();
        System.out.println(x);
    }
}
class TestThis5{
    public static void main(String args[]){
        A a = new A(10);
    }
}
```

Виведення на екран:

```
hello a
10
```

Виклик конструктора з параметром із конструктора за замовчуванням:

```
class A{
    A(){
        this(5);
        System.out.println("hello a");
    }
    A(int x){
        System.out.println(x);
    }
}
class TestThis6{
    public static void main(String args[]){
        A a = new A();
    }
}
```

Виведення на екран:

```
5
hello a
```

Реальне використання виклику конструктора this()

Виклик конструктора this() слід використовувати для повторного використання конструктора з конструктора. Він підтримує ланцюжок між конструкторами, тобто використовується для ланцюгового конструктора. Давайте розглянемо наведений нижче приклад, який відображає фактичне використання цього ключового слова.

```
class Student{
    int rollno;
    String name, course;
    float fee;
    Student(int rollno, String name, String course){
        this.rollno = rollno;
        this.name = name;
        this.course = course;
    }
    Student(int rollno, String name, String course, float fee){
        this(rollno, name, course);//reusing constructor
        this.fee = fee;
    }
    void display(){
        System.out.println(rollno+" "+name+" "+course+" "+fee);
```

```
        }
    }
    class TestThis7{
        public static void main(String args[]){
            Student s1 = new Student(111,"ankit","java");
            Student s2 = new Student(112,"sumit","java",6000f);
            s1.display();
            s2.display();
        }
    }
```

Виведення на екран:

```
111 ankit java 0.0
112 sumit java 6000.0
```

Правило: виклик this() повинен бути першим оператором у конструкторі.

```
class Student{
    int rollno;
    String name, course;
    float fee;
    Student(int rollno, String name, String course){
        this.rollno=rollno;
        this.name=name;
        this.course=course;
    }
    Student(int rollno, String name, String course, float fee){
        this.fee=fee;
        this(rollno, name, course); // Error
    }
    void display(){
        System.out.println(rollno+" "+name+" "+course+" "+fee);
    }
}
class TestThis8{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit","java");
        Student s2=new Student(112,"sumit","java",6000f);
        s1.display();
        s2.display();
    }
```

```
}
```

Виведення на екран:

```
Error: Could not find or load main class
```

this: передати як аргумент у методі

Ключове слово *this* також можна передати як аргумент у методі. В основному він використовується при обробці подій. Розглянемо приклад:

```
class S2{
    void m(S2 obj){
        System.out.println("method is invoked");
    }
    void p(){
        m(this);
    }
    public static void main(String args[]){
        S2 s1 = new S2();
        s1.p();
    }
}
```

Виведення на екран:

```
method is invoked
```

Застосування this, яке можна передати як аргумент:

При обробці подій (або) у ситуації, коли ми маємо надати посилання класу на інший. Він використовується для повторного використання одного об'єкта в багатьох методах.

---

this: передати як аргумент у виклик конструктора

Ми також можемо передати це ключове слово в конструктор. Це корисно, якщо нам доводиться використовувати один об'єкт у кількох класах. Розглянемо приклад:

```
class B{
    A4 obj;
    B(A4 obj){
        this.obj=obj;
    }
    void display(){
        // з використанням даних класу A4
```

```
        System.out.println(obj.data);
    }
}
class A4{
    int data=10;
    A4(){
        B b=new B(this);
        b.display();
    }
    public static void main(String args[]){
        A4 a = new A4();
    }
}
```

Виведення на екран:

```
10
```

this: повернення поточного екземпляра класу

Ми можемо повернути ключове слово *this* як оператор методу. У такому випадку тип методу повернення повинен бути типом класу (не примітивним). Розглянемо приклад:

Синтаксис this, який можна повернути як оператор

```
return_type method_name(){
    return this;
}
```

Приклад використання ключового слова this, яке ви повертаєте як оператор методу

```
class A{
    A getA(){
        return this;
    }
    void msg(){
        System.out.println("Hello java");
    }
}
class Test1{
    public static void main(String args[]){
        new A().getA().msg();
    }
}
```

Виведення на екран:

```
Hello java
```

Контрольні запитання

1. Яке призначення ключового слова *this* у Java?
2. Наведіть приклади використання ключового слова *this* у якості посилання на поле поточного класу.
3. Чи можна використовувати ключове слово *this* для виклику методу поточного класу? Наведіть приклади.
4. Як використовують ключове слово *this* для виклику конструктора поточного класу? Наведіть приклади.
5. Чи можна використати ключове слово *this* для передачі аргументу у методі?
6. Чи можна використати ключове слово *this* для передачі аргументу під час виклику конструктора?
7. Чи можна використати ключове слово *this* для повернення екземпляра поточного класу? Наведіть приклади.

ЛЕКЦІЯ 7. ІНКАПСУЛЯЦІЯ В JAVA

Інкапсуляція в Java – це *процес обгортання коду та даних в єдиний блок*, наприклад, у капсулі змішано декілька лікарських засобів.

Ми можемо створити повністю інкапсульований клас на Java, зробивши всі поля класу приватними. Тепер ми можемо використовувати методи setter і getter для встановлення та отримання даних.

Клас Java Bean є прикладом повністю інкапсульованого класу.

Перевага інкапсуляції в Java

Використовуючи лише метод встановлення або отримання, ви можете зробити клас доступним лише для читання чи запису. Іншими словами, ви можете використати методи геттера або сеттера.

Це веде до *контролю над даними*. Припустимо, ви хочете встановити значення id, яке має бути більше ніж 100, ви можете написати умову в методі setter. Ви можете написати умову, щоб не зберігати від'ємні числа в методах встановлення.

Це спосіб *приховати дані* в Java, оскільки інші класи не зможуть отримати доступ до даних через приватні поля даних.

Інкапсульований клас *легко перевірити*. А це зручно у випадку модульного тестування.

Стандартні IDE дають можливість генерувати геттери та сеттери. Це дозволяє *легко і швидко створити інкапсульований клас* у Java.

Простий приклад інкапсуляції в Java

Давайте подивимося на простий приклад інкапсуляції класу, який має лише одне поле з методами сеттера і геттера.

Файл: Student.java

```
// Клас Java, який є повністю інкапсульованим класом.
// У нього є приватне поле, а також методи
// отримання та встановлення.
package com.javatpoint;
public class Student{
    // приватне поле даних
    private String name;
    // метод getter для name
    public String getName(){
        return name;
    }
    // метод setter для name
    public void setName(String name){
        this.name = name;
    }
}
```

Файл: Test.java

```
// Клас Java для перевірки інкапсульованого класу.
package com.javatpoint;
class Test{
    public static void main(String[] args){
        // створення екземпляра інкапсульованого класу
        Student s = new Student();
        // встановлення значення поля name
        s.setName("vijay");
        // отримання значення поля name
        System.out.println(s.getName());
    }
}
```

Компіляція: javac -d . Test.java

Виконання: java com.javatpoint.Test

Виведення на екран:

```
vijay
```

Клас лише для читання

```
// Клас Java, який має лише методи getter
public class Student{
    // приватне поле
    private String college="AKG";
    // метод getter для college
    public String getCollege(){
        return college;
    }
}
```

Тепер ви не можете змінити значення поля college, яке є "AKG".

```
s.setCollege("KITE"); // відображатиме помилку часу компіляції
```

Клас лише для запису

```
// Клас Java, який має лише методи setter
public class Student{
    // приватне поле даних
    private String college;
    // getter method для college
    public void setCollege(String college){
        this.college = college;
    }
}
```

Тепер ви не можете отримати значення college, ви можете його лише змінити.

System.out.println(s.getCollege()); // Помилка часу компіляції, тому що такого методу немає

System.out.println(s.college); // Помилка часу компіляції, оскільки поле college є приватним.

// Отже, до нього неможливо отримати доступ поза межами класу

Ще один приклад інкапсуляції в Java

Давайте подивимося на інший приклад інкапсуляції, яка має лише чотири поля з методами сеттера і геттера.

Файл: Account.java

```
// Клас облікового запису, який є повністю
// інкапсульованим класом.
// У нього є приватне поле, а також методи
// отримання та встановлення.
class Account {
    // приватні поля даних
```

```
    private long acc_no;
    private String name, email;
    private float amount;
    // загальнодоступні методи гетера і сеттера
    public long getAcc_no() {
        return acc_no;
    }
    public void setAcc_no(long acc_no) {
        this.acc_no = acc_no;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public float getAmount() {
        return amount;
    }
    public void setAmount(float amount) {
        this.amount = amount;
    }
}
```

Файл: TestAccount.java

```
// Клас Java для перевірки інкапсульованого класу Account
public class TestEncapsulation {
    public static void main(String[] args) {
        // створення екземпляра класу Account
        Account acc = new Account();
        // встановлення значень за допомогою методів
встановлення
        acc.setAcc_no(7560504000L);
        acc.setName("Sonoo Jaiswal");
```

```
        acc.setEmail("sonoojaiswal@javatpoint.com");
        acc.setAmount(500000f);
        // отримання значень за допомогою методів гетера
        System.out.println(acc.getAcc_no()+" "+acc.getName()+
                          " "+acc.getEmail()+" "+acc.getAmount());
    }
}
```

Виведення на екран:

```
7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0
```

Контрольні запитання

1. Що таке інкапсуляція?
2. Що таке геттери? Яке їх призначення? Наведіть приклади.
3. Що таке сеттери? Яке їх призначення? Наведіть приклади.
4. Що таке повністю інкапсульований клас? Наведіть приклади.

ЛЕКЦІЯ 8. ПАКЕТ У JAVA

План

1. Пакет у Java
2. Підпакет у Java

Пакет у Java

Пакет у Java – це група класів, інтерфейсів і підпакетів.

Пакет можна розділити на дві види: вбудований пакет і пакет, визначений користувачем.

Існує багато вбудованих пакетів, таких як java, lang, awt, javax, swing, net, io, util, sql тощо.

Розглянемо детальніше, як можна створювати та використовувати пакети, які визначені користувачем.

Перевага пакету в Java

1) Пакет у Java використовується для групування класів та інтерфейсів, щоб їх можна було легко підтримувати.

2) Пакет забезпечує захист доступу.

3) Пакет усуває зіткнення імен.

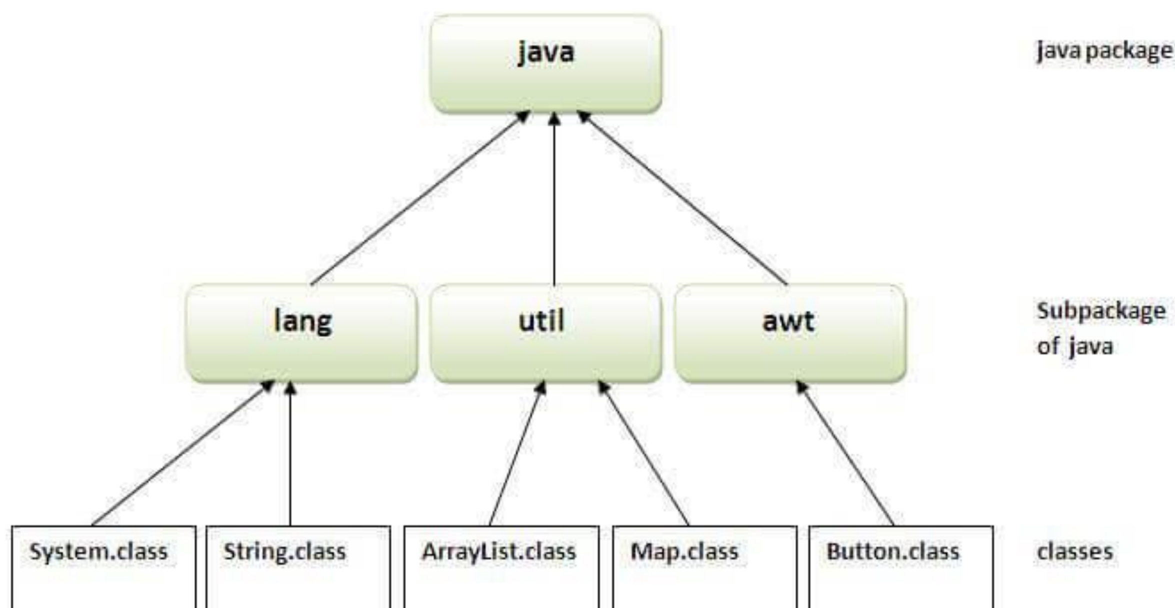

Рис. 18. Вбудовані пакети у Java

Простий приклад пакету Java

Ключове слово package використовується для створення пакета в Java.

```
// Файл Simple.java
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

Як зібрати пакет

Якщо ви не використовуєте будь-яку IDE, вам потрібно дотримуватися наведеного нижче синтаксису:

```
javac -d directory javafilename
```

для прикладу

```
javac -d . Simple.java
```

Перемикач -d вказує на місце призначення, куди розмістити сформований файл класу. Ви можете використовувати будь-яке ім'я каталогу, наприклад /home (у випадку Linux), d:/abc (у випадку Windows) тощо. Якщо ви хочете зберегти пакет у тому самому каталозі, ви можете використовувати . (крапка).

Як запустити програму для пакетів Java

Для запуску класу потрібно використовувати повне ім’я, наприклад mypack.Simple тощо.

Для складання: javac -d . Simple.java

Для запуску: java mypack.Simple

Виведення на екран:

```
Welcome to package
```

-d – це перемикач, який повідомляє компілятору, куди помістити файл класу, тобто він представляє призначення. . (крапка) представляє поточну папку.

Як отримати доступ до пакета з іншого пакета?

Існує три способи доступу до пакета поза пакетом:

1) імпортувати пакет.*;

2) імпорт package.classname;

3) повне ім’я.

1) Використання назви пакета.*

Якщо ви використовуєте package.*, то всі класи та інтерфейси цього пакета будуть доступні, але не підпакети.

Ключове слово import використовується, щоб зробити класи та інтерфейс іншого пакета доступними для поточного пакета.

Приклад пакета, який імпортує ім’я пакета.*

```
// Файл A.java
package pack;
public class A{
    public void msg(){
        System.out.println("Hello");
    }
}
// Файл B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Виведення на екран:

```
Hello
```

2) Використання packagename.classname

Якщо ви імпортуєте package.classname, тоді буде доступним лише оголошений клас цього пакета.

Приклад пакета за імпортом package.classname

```
// Файл A.java
package pack;
public class A{
    public void msg(){
        System.out.println("Hello");
    }
}
// Файл B2.java
package mypack;
import pack.A;
class B2{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Виведення на екран:

```
Hello
```

3) Використання повного імені

Якщо ви використовуєте повне ім'я, то буде доступний лише оголошений клас цього пакета. Тепер немає потреби в імпорті. Але вам потрібно використовувати повне ім'я кожного разу, коли ви отримуєте доступ до класу або інтерфейсу.

Зазвичай він використовується, коли два пакети мають однакову назву класу, наприклад, пакети java.util і java.sql містять клас Date.

Приклад пакета за імпортом повного імені

```
// Файл A.java
package pack;
public class A{
    public void msg(){
        System.out.println("Hello");
    }
}
// Файл B3.java
```

```
package mypack;
class B3{
    public static void main(String args[]){
        pack.A obj = new pack.A(); // використання повного імені
        obj.msg();
    }
}
```

Виведення на екран:

```
Hello
```

Примітка. Якщо ви імпортуєте пакет, підпакети не імпортуються.

Якщо ви імпортуєте пакет, будуть імпортовані всі класи та інтерфейс цього пакета, за винятком класів та інтерфейсів підпакетів. Отже, вам також потрібно імпортувати підпакет.

Примітка. Послідовність дій у програмі: пакет, імпорт, а потім клас.

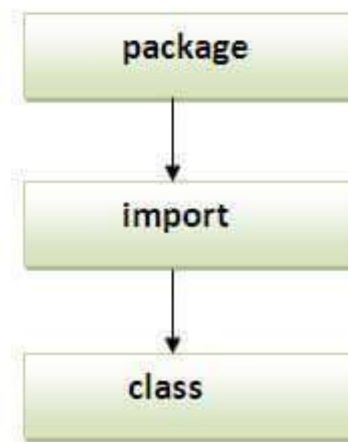

Рис. 19. Послідовність дій у програмі: пакет, імпорт, а потім клас

Підпакет у Java

Пакет всередині пакета називається *підпакетом*.

Давайте розглянемо приклад: Sun Microsystems визначила пакет з назвою java, який містить багато класів, таких як System, String, Reader, Writer, Socket тощо. Ці класи представляють певну групу, наприклад, Reader та Writer – для операцій введення/виведення, Socket та ServerSocket класи призначені для роботи в мережі тощо. Отже, Sun розбив пакет java на підпакети, такі як lang, net, io тощо, і помістив пов’язані класи введення/виведення в пакет io, класи Server і ServerSocket в пакети net тощо.

Стандартом визначення пакета є domain.company.package, наприклад, com.javatpoint.bean або org.sssit.dao.

Приклад підпакета

```
package com.javatpoint.core;
class Simple2{
    public static void main(String args[]){
        System.out.println("Hello subpackage");
    }
}
```

Скласти: javac -d . Simple2.java

Для запуску: java com.javatpoint.core.Simple2

Виведення на екран:

```
Hello subpackage
```

Як помістити файл класу в інший каталог або на диск?

Розглянемо, як помістити файл класу вихідного файлу A.java в папку classes диска C:. Наприклад:

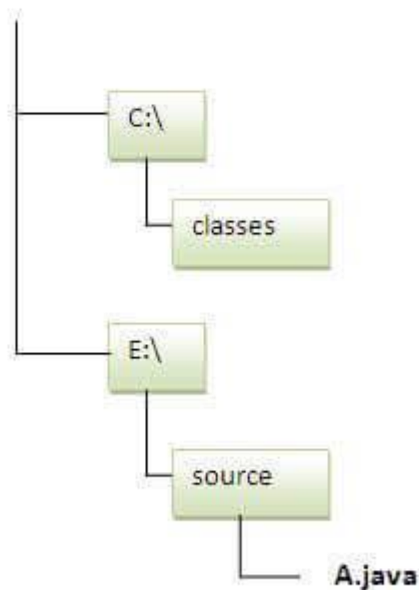

Рис. 20. Приклад структури папок

```
// Файл Simple.java
package mypack;
public class Simple3{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

Для компіляції:

e:\sources> javac -dc:\classes Simple3.java

Для запуску:

Щоб запустити цю програму з каталогу e:\source, вам потрібно встановити шлях до каталогу, в якому знаходиться файл класу.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple3

Інший спосіб запустити цю програму за допомогою перемикача -classpath java:

Перемикач -classpath можна використовувати з javac та java tool.

Щоб запустити цю програму з каталогу e:\source, ви можете використовувати перемикач -classpath java, який вказує, де шукати файл класу. Наприклад:

e:\sources> java -classpath c:\classes mypack.Simple3

Виведення на екран:

```
Welcome to package
```

Способи завантаження файлів класу або файлів jar

Існує два способи завантаження тимчасових та постійних файлів класів:

– тимчасовий (встановивши шлях до класу в командному рядку; перемикач -classpath);

– постійний (установивши шлях до класу в змінних середовища; створивши файл jar, який містить усі файли класів, і скопіювавши файл jar у папку jre/lib/ext).

Правило: у вихідному файлі Java може бути тільки один відкритий клас, і він повинен бути збережений під ім'ям загальнодоступного класу.

```
// зберегти як C.java, інакше помилка часу компіляції
class A{
}
class B{
}
public class C{
}
```

Як помістити два публічних класи в пакет?

Якщо ви хочете помістити два відкритих класи в пакет, створіть два вихідні файли Java, які містять один відкритий клас, але ім'я пакета залишиться незмінним. Наприклад:

// Файл A.java

```
package javatpoint;
public class A{
}
// Файл B.java
package javatpoint;
public class B{
}
```

Контрольні запитання

1. Що таке пакет у Java? Яке його призначення?
2. Які види пакетів використовуються в Java?
3. Які пакети називаються вбудованими? Наведіть приклади.
4. Які існують способи доступу до пакета?
5. Що таке підпакет? Наведіть приклади.
6. Як помістити файл класу в інший каталог або на диск?
7. Які є способи завантаження файлів класу або файлів jar?

ЛЕКЦІЯ 9. УСПАДКУВАННЯ В JAVA

1. Однорівневе успадкування
2. Багаторівневе успадкування
3. Ієрархічне успадкування

Успадкування в Java – це механізм, за якого один об'єкт набуває всіх властивостей та поведінки батьківського об'єкта. Це важлива частина ООП.

Ідея наслідування в Java полягає в тому, що ви можете створювати нові класи, побудовані на існуючих класах. При успадкуванні від існуючого класу можна повторно використовувати методи та поля батьківського класу. Крім того, ви також можете додати нові методи та поля у поточний клас.

Успадкування реалізує *відносини IS-A,* тобто відносини *батько-дитина*.

Навіщо використовувати успадкування в Java

– Для перевизначення методу (таким чином можна досягти поліморфізму під час виконання).

– Для повторного використання коду.

Терміни, що використовуються в успадкуванні

– *Клас:* Клас – це група об'єктів, які мають загальні властивості. Це шаблон або план, за яким створюються об'єкти.

– *Підклас/Дочірній клас:* Підклас – це клас, який успадковує інший клас. Його також називають похідним класом, розширеним класом або дочірнім класом.

– *Суперклас/Батьківський клас:* Суперклас – це клас, з якого підклас успадковує методи. Його також називають базовим або батьківським класом.

– *Повторне використання:* Як вказує назва, повторне використання – це механізм, який дозволяє вам повторно використовувати поля та методи існуючого класу під час створення нового класу. Ви можете використовувати ті самі поля та методи, які вже визначені у попередньому класі.

Синтаксис успадкування Java

```
class   Назва_підкласу extends Назва_суперкласу {
    // методи та поля
}
```

Ключове слово extends вказує на те, що ви створюєте новий клас, похідний від існуючого класу. Значення "розширюється" вказує на збільшенні функціональності.

У термінології Java успадкований клас називається батьківським або надкласом, а новий – дочірнім або підкласом.

Приклад успадкування Java

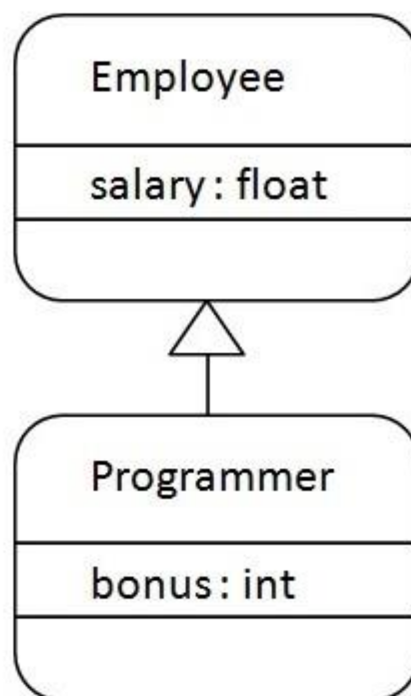

Рис. 21. Приклад спадкування Java

Як показано на рис. 21, програміст – це підклас, а співробітник – надклас. Відносини між цими двома класами – це «Працівник-програміст IS-A». Це означає, що програміст – це тип співробітника.

```
class Employee{
    float salary = 40000; // заробітна плата
}
class Programmer extends Employee{
    int bonus = 10000; // бонус
    public static void main(String args[]){
        Programmer p = new Programmer();
        // Друк заробітної плати програміста
        System.out.println("Programmer salary is:"+p.salary);
        // Друк бонусів програміста
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
```

Виведення на екран:

```
Programmer salary is:40000.0
Bonus of Programmer is:10000
```

У наведеному вище прикладі об'єкт Programmer може отримати доступ до поля власного класу, а також класу Employee, тобто повторного використання коду.

Види успадкування в Java

Виходячи з класу, у Java може бути три види успадкування: *одно-, багаторівневе* та *ієрархічне*.

У програмуванні Java *багаторазове* та *гібридне успадкування* підтримується лише через інтерфейс. Про інтерфейси ми дізнаємося пізніше.

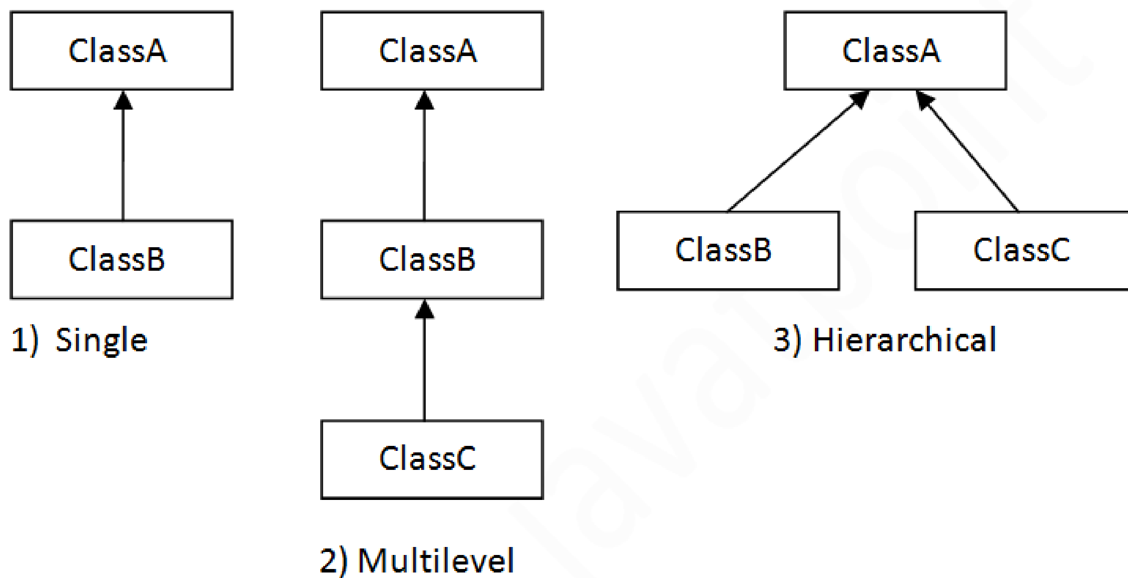

Рис. 22. Види успадкування в Java

Примітка: множинне успадкування не підтримується в Java через клас.

Коли один клас успадковує кілька класів, це називається *множинним успадкуванням*. Наприклад:

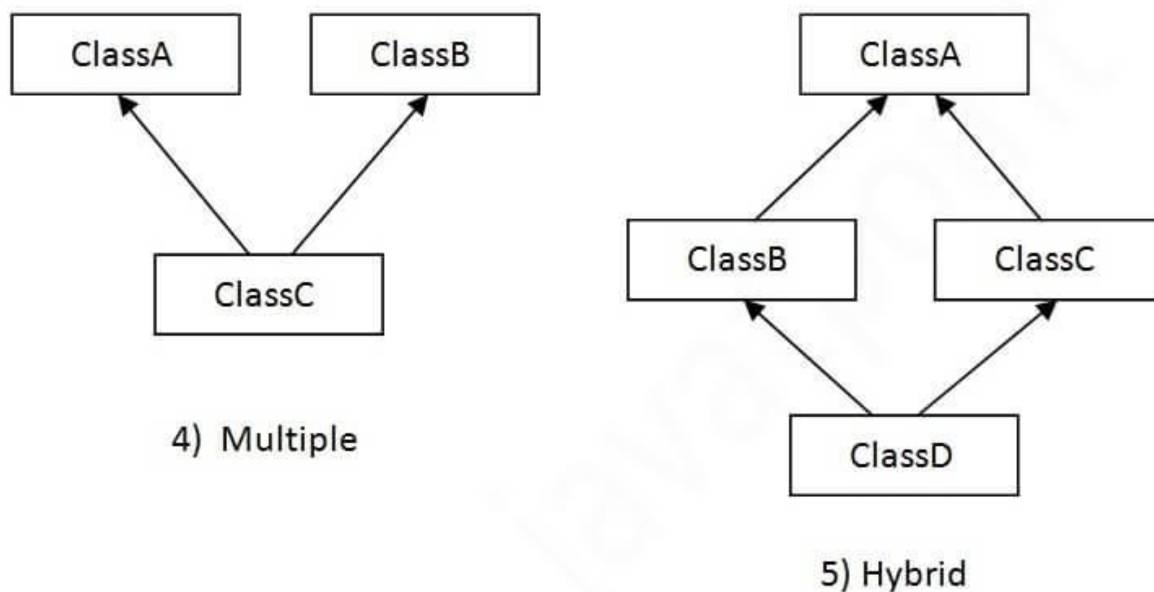

Рис. 23. Множинне успадкування в Java

Однорівневе успадкування

Коли клас успадковує інший клас, це називається *однорівневим успадкуванням*. У наведеному нижче прикладі клас Dog успадковує клас Animal, тому це однорівневе успадкування.

Файл: TestInheritance.java

```
class Animal{
	void eat(){
		System.out.println("eating..."); // їсть
	}
}
class Dog extends Animal{
	void bark(){
		System.out.println("barking..."); // гавкає
	}
}
class TestInheritance{
	public static void main(String args[]){
		Dog d = new Dog();
		d.bark();
		d.eat();
	}
}
```

Виведення на екран:

```
barking...
eating...
```

Багаторівневе успадкування

Коли існує ланцюг успадкування, це називається *багаторівневим успадкуванням*. Як ви можете бачити в наведеному нижче прикладі, клас BabyDog успадковує клас Dog, який у свою чергу успадковує клас Animal, тому це багаторівневе успадкування.

Файл: TestInheritance2.java

```
class Animal{
	void eat(){
		System.out.println("eating..."); // їсть
	}
}
class Dog extends Animal{
	void bark(){
		System.out.println("barking..."); // гавкає
	}
}
class BabyDog extends Dog{
```

```
	void weep(){
		System.out.println("weeping..."); // плаче
	}
}
class TestInheritance2{
	public static void main(String args[]){
		BabyDog d = new BabyDog();
		d.weep();
		d.bark();
		d.eat();
	}
}
```

Виведення на екран:

```
weeping...
barking...
eating...
```

Ієрархічне успадкування

Якщо два або більше класів успадковують один клас, це називається *ієрархічним успадкуванням*. У наведеному нижче прикладі класи Dog і Cat успадковують клас Animal, тому це ієрархічне успадкування.

Файл: TestInheritance3.java

```
class Animal{
	void eat(){
		System.out.println("eating..."); // їсть
	}
}
class Dog extends Animal{
	void bark(){
		System.out.println("barking..."); // гавкає
	}
}
class Cat extends Animal{
	void meow(){
		System.out.println("meowing..."); // нявкає
	}
}
class TestInheritance3{
```

```
	public static void main(String args[]){
		Cat c = new Cat();
		c.meow();
		c.eat();
	//c.bark();//C.T.Error
	}
}
```

Виведення на екран:

```
meowing...
eating...
```

Чому множинне успадкування не підтримується в Java?

Щоб зменшити складність та спростити мову, множинне успадкування не підтримується в Java.

Розглянемо сценарій, де A, B і C – це три класи. Клас C успадковує класи A і B. Якщо класи A та B мають один і той самий метод, і ви викликаєте його з об’єкта дочірнього класу, виникне неоднозначність для виклику методу класу A або B.

Оскільки помилка під час компіляції краще, ніж помилка під час виконання, Java відображає помилку часу компіляції, якщо ви успадковуєте 2 класи. Тому незалежно від того, чи є у вас той самий метод чи інший, буде помилка часу компіляції.

```
class A{
	void msg(){
		System.out.println("Hello");
	}
}
class B{
void msg(){
	System.out.println("Welcome");
}
}
class C extends A, B{ // припустимо, якби це було так

	public static void main(String args[]){
	  C obj = new C();
	  obj.msg(); // Тепер який метод msg() буде викликаний?
	}
}
```

Виведення на екран:

```
Compile Time Error
```

Контрольні запитання

1. Що таке спадкування в Java?
2. Навіщо використовувати спадкування в Java?
3. Що таке суперклас?
4. Що таке підклас?
5. Що розуміють під повторним використанням?
6. Який синтаксис спадкування в Java?
7. Які є види спадкування в Java?
8. Що таке однорівневе спадкування?
9. Що розуміють під багаторівневим спадкуванням?
10. Що таке ієрархічне спадкування? Наведіть приклади.
11. Чи підтримується множинне спадкування в Java?

ЛЕКЦІЯ 10. ВІДНОСИНИ МІЖ КЛАСАМИ В JAVA

План

1. IS-A відносини
2. HAS-A відносини
3. Асоціація
4. Агрегація
5. Композиція

Більшість класів у програмі пов’язані між собою. Давайте докладніше розглянемо, які бувають відносини між класами Java.

1. IS-A відносини

В ООП принцип IS-A ґрунтується на спадкуванні класів або реалізації інтерфейсів. Наприклад, якщо клас HeavyBox успадковує Box, ми говоримо, що HeavyBox є Box (HeavyBox IS-A Box). Або інший приклад – клас Lorry розширює клас Car. У цьому випадку Lorry IS-A Car.

Те саме стосується і реалізації інтерфейсів. Якщо клас Transport реалізує інтерфейс Moveable, то вони знаходяться у відношенні Transport IS-A Moveable.

2. HAS-A відносини

HAS-A відносини ґрунтується на використанні. Виділяють три варіанти відношення HAS-A: асоціація, агрегація та композиція.

Почнемо з асоціації. У цих відносинах об’єкти двох класів можуть посилатися один на одного. Наприклад, клас Horse HAS-A

Halter, якщо код класу Horse містить посилання на екземпляр класу Halter:

Асоціація

```
public class Halter {
}
public class Horse{
    private Halter halter;
}
```

Агрегація та композиція є окремими випадками асоціації. Агрегація – відносини, коли один об'єкт є частиною іншого. А композиція – ще тісніший зв'язок, коли об'єкт не тільки є частиною іншого об'єкта, а й взагалі не може належати іншому об'єкту. Різниця буде зрозуміла при розгляді цих відносин.

Агрегація

Об'єкт класу Halter створюється ззовні Horse та передається до конструктора для встановлення зв'язку. Якщо об'єкт класу Horse буде видалений, об'єкт класу Halter може й надалі використовуватись, якщо, звичайно, на нього залишиться посилання:

```
public class Horse {
    private Halter halter;
    public Horse(Halter halter) {
        this.halter = halter;
    }
}
```

Композиція

Тепер подивимося на реалізацію композиції. Об'єкт класу Halter створюється у конструкторі, що означає тісніший зв'язок між об'єктами. Об'єкт класу Halter не може існувати без об'єкта, що його створив Horse:

```
public class Horse {
    private Halter halter;
    public Horse() {
        this.halter = new Halter();
    }
}
```

Приклади агрегації в Java

Якщо клас має посилання на сутність, це називається *агрегація*. Агрегація реалізує відносини HAS-A.

Розглянемо ситуацію. Об’єкт “Співробітник” містить багато інформації, таку як ідентифікатор, ім’я, електронна адреса тощо. Також він містить ще один об’єкт із назвою адреси, яка містить особисту інформацію, таку як місто, штат, країна, поштовий індекс тощо, як наведено нижче.

```
class Employee{
        int id;
        String name;
        Address address; // Адреса – це клас
        ...
}
```

У такому випадку у працівника є довідкова адреса організації, тому відносини між адресою та співробітником є HAS-A.

Навіщо використовувати агрегацію?

Для повторного використання коду.

Приклад агрегації

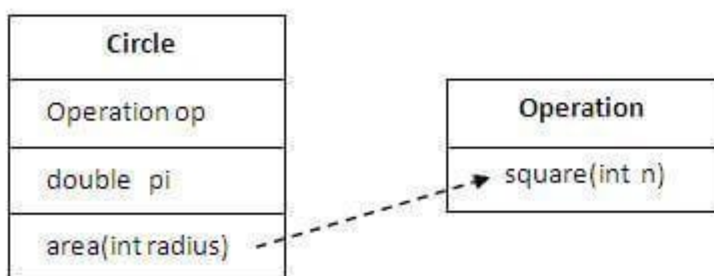

Рис. 24. Приклад агрегації

У цьому прикладі ми створили посилання на клас Operation у класі Circle.

```
class Operation {
      int square(int n){
            return n*n;
      }
}
class Circle {
      Operation op; // агрегація
      double pi=3.14;
      double area(int radius){
            op = new Operation();
            // повторне використання коду
            // (тобто делегує виклик методу)
```

```
        int rsquare = op.square(radius);
        return pi*rsquare;
    }
    public static void main(String args[]){
        Circle c = new Circle();
        double result=c.area(5);
        System.out.println(result);
    }
}
```

Виведення на екран:

```
78.5
```

Коли використовувати агрегацію?

– Повторне використання коду також найкраще досягається за допомогою агрегації, коли немає зв'язку "is-a".

– Успадкування слід використовувати лише у тому випадку, якщо відносини "is-a" зберігаються протягом усього життя об'єктів, що беруть участь; в іншому випадку агрегація – найкращий вибір.

Розуміння значущого прикладу агрегації

У цьому прикладі у працівника є об'єкт адреси, об'єкт адреси містить власну інформацію, таку як місто, штат, країна тощо.

Файл Address.java

```
class Address {
    String city, state, country;
    public Address(String city, String state, String country) {
            this.city = city; // місто
            this.state = state; // штат
            this.country = country; // країна
    }
}
public class Emp {
    int id;
    String name;
    Address address;
    public Emp(int id, String name, Address address) {
        this.id = id;
        this.name = name;
        this.address=address;
    }
    void display(){
```

```
            System.out.println(id+" "+name);
            System.out.println(address.city+" "+address.state+"
"+address.country);
        }
        public static void main(String[] args) {
            Address address1 = new Address("gzb", "UP", "india");
            Address address2 = new Address("gno", "UP", "india");
            Emp e = new Emp(111, "varun", address1);
            Emp e2 = new Emp(112, "arun", address2);
            e.display();
            e2.display();
        }
    }
```

Виведення на екран:

```
111 varun gzb UP india
112 arun gno UP india
```

Контрольні запитання

1. Що таке відношення IS-A між класами в Java? Наведіть приклади.
2. Що таке відношення HAS-A між класами в Java?
3. Що таке асоціація?
4. Що таке агрегація?
5. Що таке композиція?
6. Яка відмінність між агрегацією і композицією?

ЛЕКЦІЯ 11. ПЕРЕВАНТАЖЕННЯ МЕТОДУ В JAVA

План

1. Метод перевантаження: зміна кількості аргументів
2. Перевантаження методу: зміна типу даних аргументів
3. Метод перевантаження та просування типу

Якщо клас має кілька методів з однаковими іменами, але різними за параметрами, він відомий як *перевантаження методів.*

Якщо нам доводиться виконувати лише одну операцію, однакова назва методів збільшує читаність програми.

Припустимо, вам потрібно виконати додавання чисел, але може бути будь-яка кількість аргументів. Якщо ви напишете метод, такий як add(int, int) для двох параметрів, і add(int, int, int) для трьох параметрів,

тоді це вам, а також іншим програмістам, може бути важко зрозуміти поведінку методу, оскільки його назва не відрізняється.

Отже, ми виконуємо перевантаження методів, щоб швидко зрозуміти програму.

Перевага методу перевантаження

Перевантаження методів *збільшує читаність програми*.

Способи перевантаження методу

Існує два способи перевантажити метод у Java

1. Зміна кількість аргументів.
2. Зміна типу даних.

У Java перевантаження методу неможливе лише шляхом зміни типу повернення методу.

Перевантаження методу: зміна кількості аргументів

У цьому прикладі ми створили два методи: перший метод add() виконує додавання двох чисел, а другий метод додавання – додавання трьох чисел.

У цьому прикладі ми створюємо статичні методи щоб нам не потрібно було створювати екземпляр для виклику методів.

```
class Adder{
    static int add(int a, int b){
        return a + b;
    }
    static int add(int a, int b, int c){
        return a + b + c;
    }
}
class TestOverloading1{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(11,11,11));
    }
}
```

Виведення на екран:

```
22
33
```

Перевантаження методу: зміна типу даних аргументів

У цьому прикладі ми створили два методи, що відрізняються типом даних. Перший метод додавання отримує два цілі аргументи, а другий метод додавання отримує два дійсні аргументи.

```
class Adder{
    static int add(int a, int b){
        return a + b;
    }
    static double add(double a, double b){
        return a + b;
    }
}
class TestOverloading2{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(12.3,12.6));
    }
}
```

Виведення на екран:

```
22
24.9
```

Чому перевантаження методу неможливе, якщо змінити лише тип повернення методу?

У Java перевантаження методу неможливе шляхом зміни типу повернення методу лише через неоднозначність. Давайте подивимось, як може виникнути двозначність:

```
class Adder{
    static int add(int a, int b){
        return a + b;
    }
    static double add(int a, int b){
        return a + b;
    }
}
class TestOverloading3{
    public static void main(String[] args){
        // Тут, як Java може визначити, який метод sum()
        // слід викликати?
        System.out.println(Adder.add(11,11)); // двозначність
```

```
        }
    }
```

Виведення на екран:

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

Примітка: помилка часу компіляції краще, ніж помилка часу виконання. Отже, компілятор Java відображає помилку часу компілятора, якщо ви оголошуєте той самий метод з однаковими параметрами.

Чи можна перевантажити метод main()?

Так, методом перевантаження. Ви можете мати будь-яку кількість основних методів у класі шляхом перевантаження методів. Але JVM викликає метод main(), який отримує масив рядків лише як аргументи. Давайте розглянемо простий приклад:

```
class TestOverloading4{
    public static void main(String[] args){
        System.out.println("main with String[]");
    }
    public static void main(String args){
        System.out.println("main with String");
    }
    public static void main(){
        System.out.println("main without args");
    }
}
```

Виведення на екран:

```
main з String []
```

Метод перевантаження та просування типу

Один тип переходить до іншого неявно, якщо не знайдено відповідного типу даних. Давайте зрозуміємо концепцію за рисунком, наведеним нижче:

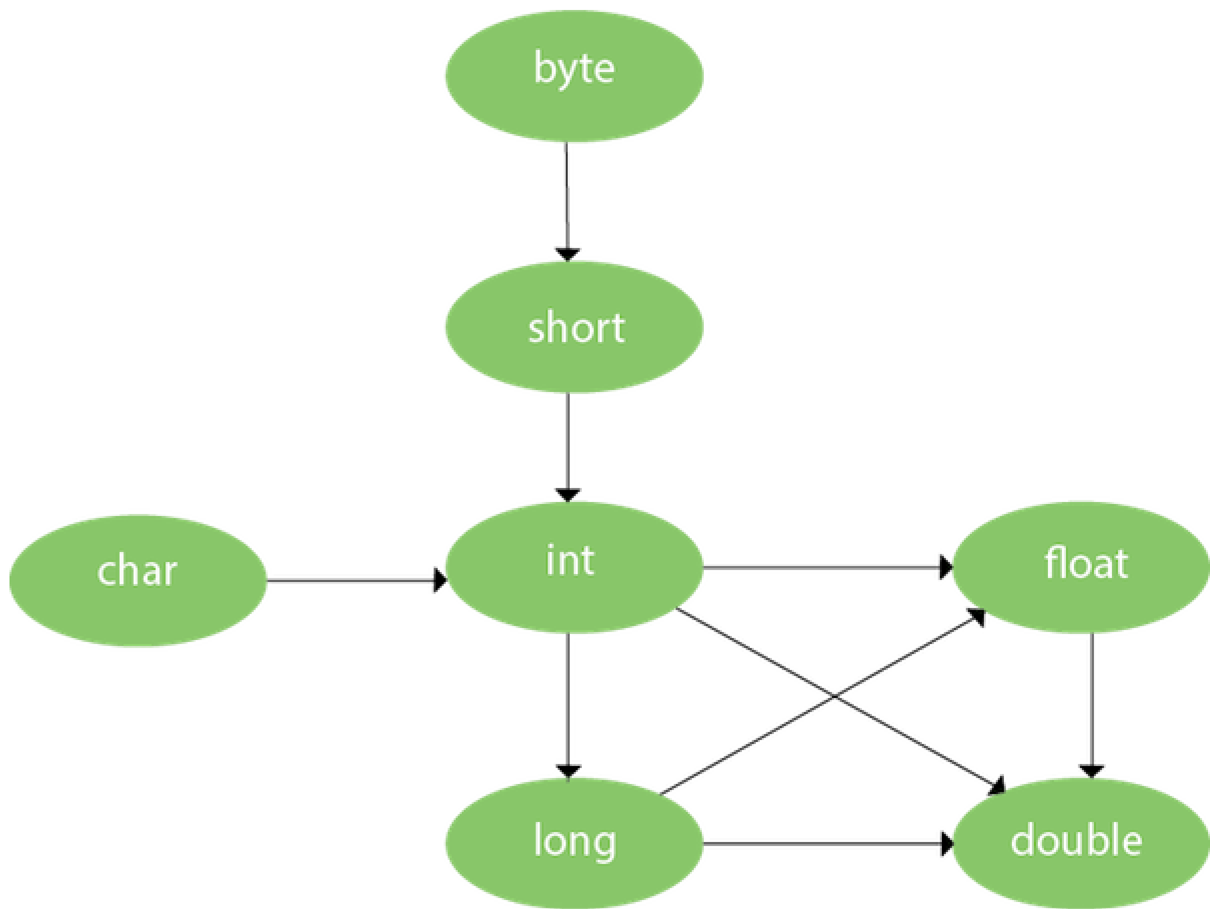

Рис. 25. Перетворення типів в Java

Як показано на наведеній вище діаграмі, байт можна перетворити на short, int, long, float або double. Короткий тип даних можна перетворити на int, long, float або double. Тип даних char можна перетворити на int, long, float або double тощо.

Приклад перевантаження методу за допомогою TypePromotion

```
class OverloadingCalculation1{
    void sum(int a, long b){
        System.out.println(a + b);
    }
    void sum(int a, int b, int c){
        System.out.println(a + b+ c);
    }
    public static void main(String args[]){
        OverloadingCalculation1 obj = new OverloadingCalculation1();
        // тепер другий літерал int буде підвищено до long
        obj.sum(20,20);
```

```
            obj.sum(20,20,20);
        }
    }
```

Виведення на екран:

```
40
60
```

Приклад перевантаження методу просуванням типу, якщо знайдено відповідність

Якщо в методі є відповідні аргументи типу, просування типу не виконується.

```
class OverloadingCalculation2{
    void sum(int a, int b){
        // викликано метод int arg
        System.out.println("int arg method invoked");
    }
    void sum(long a, long b){
        // викликано метод long arg
        System.out.println("long arg method invoked");
    }
    public static void main(String args[]){
    OverloadingCalculation2 obj = new OverloadingCalculation2();
    obj.sum(20,20); // тепер викликається метод int arg sum()
  }
}
```

Виведення на екран:

```
int arg method invoked
```

Приклад перевантаження методу просуванням типу у разі неясності

```
class OverloadingCalculation3{
    void sum(int a, long b){
        System.out.println("a method invoked");
    }
    void sum(long a, int b){
        System.out.println("b method invoked");
    }
    public static void main(String args[]){
    OverloadingCalculation3 obj = new OverloadingCalculation3();
    obj.sum(20,20); // тепер двозначність
  }
```

}

Виведення на екран:

```
Compile Time Error
```

Примітка. Один тип не демонструється неявно, наприклад double не може бути приведений до будь-якого типу неявно.

Контрольні запитання

1. Що таке перевантаження методів?
2. Які існують способи перевантаження методу? Наведіть приклади.
3. Що розуміють під перевизначенням методу в Java?
4. Яке призначення перевизначення методу?
5. Які існують правила заміни методів у Java?

ЛЕКЦІЯ 12. ПЕРЕВИЗНАЧЕННЯ МЕТОДУ В JAVA

План

1. Розуміння проблеми без перевизначення методу
2. Чи можна змінити статичний метод?
3. Перевантаження методу проти перевизначення методу

Якщо підклас (дочірній клас) має той самий метод, що й оголошений у батьківському класі, то це розуміють як *перевизначення методу в Java.*

Іншими словами, якщо підклас передбачає конкретну реалізацію методу, який був оголошений одним із батьківських класів, то це називається *перевизначенням методу.*

Використання перевизначення методу в Java

Перевизначення методу використовується для забезпечення конкретної реалізації методу, який уже передбачений його надкласом.

Метод перевизначення використовується для поліморфізму під час виконання.

Правила перевизначення методів у Java

1. Метод повинен мати таку ж назву, що і в батьківському класі.
2. Метод повинен мати той самий параметр, що і в батьківському класі.
3. Повинні існувати відносини IS-A (успадкування).

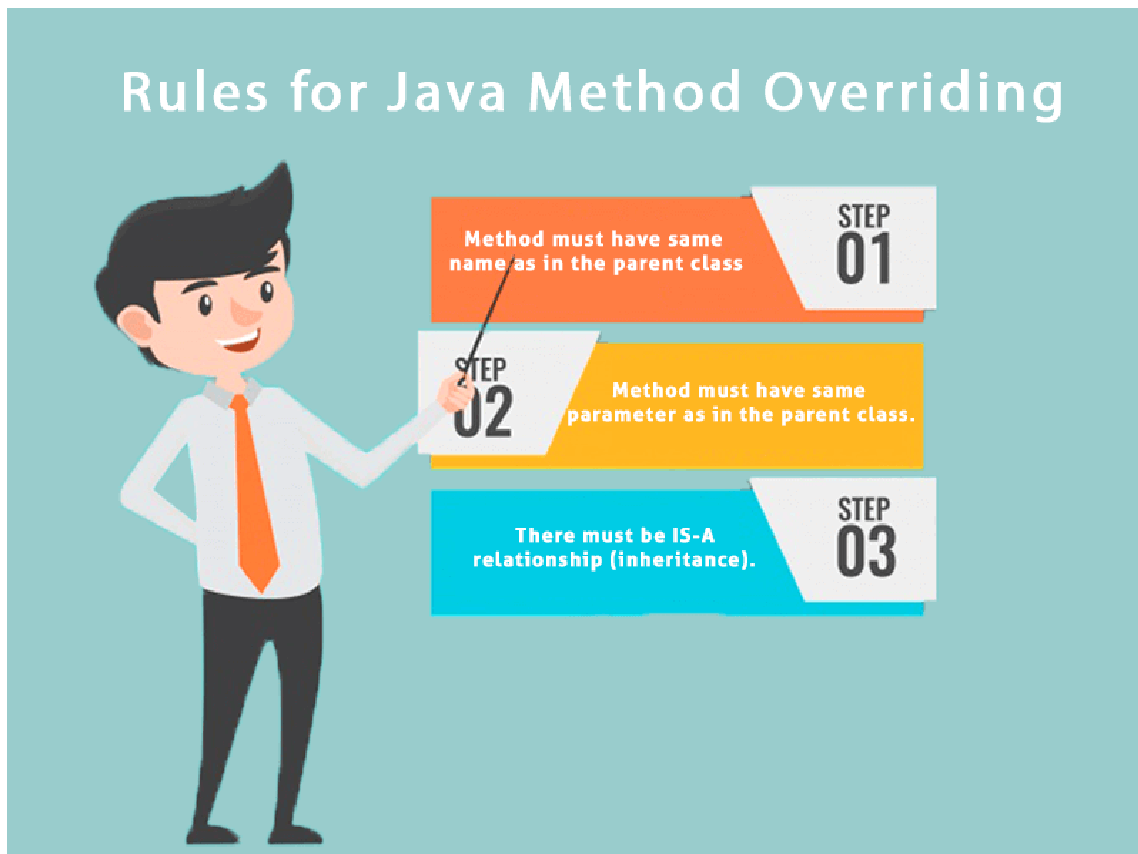

Рис. 26. Правила перевизначення методів у Java

Розуміння проблеми без заміни методу

Давайте розберемося з проблемою, з якою ми можемо зіткнутися у програмі, якщо не будемо використовувати заміну методу.

Розглянемо програму на Java, яка демонструє, чому нам потрібно перевизначити метод.

Тут ми викликаємо метод батьківського класу з об’єкта дочірнього класу.

```
// Створення батьківського класу
class Vehicle{
    void run(){
        // Виведення на друк "Автомобіль працює"
        System.out.println("Vehicle is running");
    }
}
// Створення дочірнього класу
class Bike extends Vehicle{
    public static void main(String args[]){
        // створення екземпляра дочірнього класу
        Bike obj = new Bike();
```

```
            // виклик методу з екземпляром дочірнього класу
            obj.run();
        }
    }
```

Виведення на екран:

```
Vehicle is running
```

Проблема в тому, що потрібно надати конкретну реалізацію методу run() у підкласі, тому ми використовуємо метод перевизначення методу.

Приклад перевизначення методу

У цьому прикладі ми визначили метод run у підкласі, який визначено у батьківському класі, але він має певну реалізацію. Назва і параметр методу однакові, і між класами існує зв'язок IS-A, тому існує перевизначення методу.

Програма Java для ілюстрації використання перевизначення методу в Java

```
// Створення батьківського класу
class Vehicle{
    // визначення методу
    void run(){
        System.out.println("Vehicle is running");
    }
}
// Створення дочірнього класу
class Bike2 extends Vehicle{
    // визначення того самого методу, що і в батьківському класі
    void run(){
        // Виведення на друк "Велосипед працює безпечно"
        System.out.println("Bike is running safely");
    }
    public static void main(String args[]){
        Bike2 obj = new Bike2(); // створення об'єкта
        obj.run(); // метод виклику
    }
}
```

Виведення на екран:

```
Bike is running safely
```

Реальний приклад перевизначення методу в Java

Розглянемо сценарій, коли банк – це клас, який забезпечує функціональність для отримання процентної ставки. Однак процентна ставка залежить від банків. Наприклад, банки SBI, ICICI та AXIS можуть надавати процентну ставку 8%, 7%та 9%.

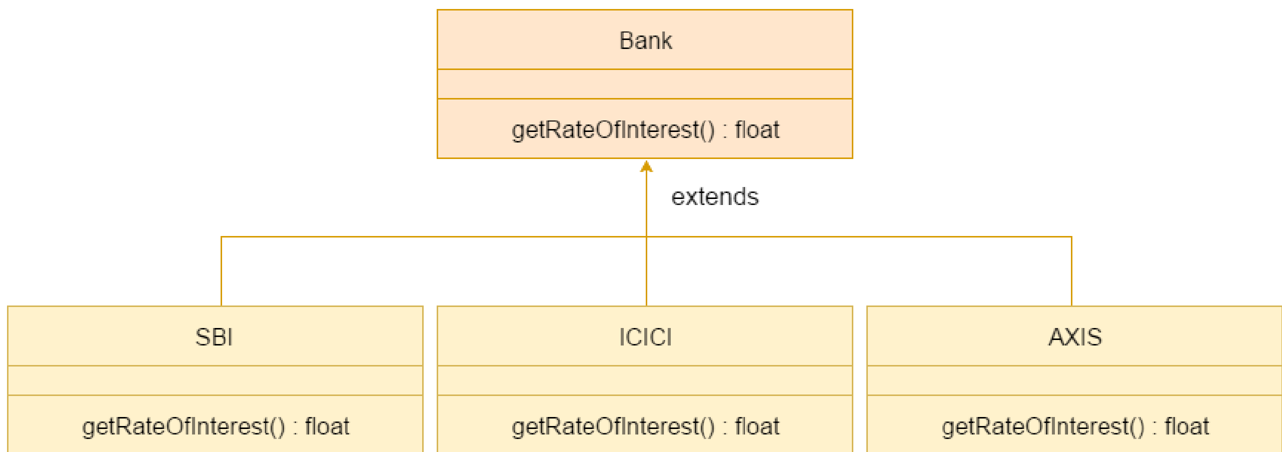

Рис. 27. Приклад перевизначення методу в Java

Примітка. Перевизначення методу Java в основному використовується в поліморфізмі середовища виконання, про що ми дізнаємось згодом.

Розглянемо програму, яка демонструє реальний сценарій перевизначення методу в Java, де три класи замінюють метод батьківського класу.

```
// Створення батьківського класу.
class Bank{
    int getRateOfInterest(){
        return 0;
    }
}
// Створення дочірніх класів.
class SBI extends Bank{
    int getRateOfInterest(){
        return 8;
    }
}
class ICICI extends Bank{
    int getRateOfInterest(){
        return 7;
    }
```

```
}
class AXIS extends Bank{
    int getRateOfInterest(){
        return 9;
    }
}
// Тестування класу для створення об'єктів та виклику методів
class Test2{
    public static void main(String args[]){
        SBI s=new SBI();
        ICICI i=new ICICI();
        AXIS a=new AXIS();
        System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
        System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
        System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
    }
}
```

Виведення на екран:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Чи можемо ми змінити статичний метод?

Ні, статичний метод не можна змінити. Це можна довести поліморфізмом під час виконання, ми дізнаємось про це пізніше.

Чому ми не можемо замінити статичний метод?

Це тому, що статичний метод пов'язаний з класом, тоді як метод екземпляра пов'язаний з об'єктом. Static належить до області класів, а екземпляр – до області кучі.

Чи можемо ми замінити основний метод Java?

Ні, тому що основний – це статичний метод.

Контрольні запитання

1. Що таке перевантаження методів?
2. Які існують способи перевантаження методу? Наведіть приклади.
3. Що розуміють під перевизначенням методу в Java?
4. Яке призначення перевизначення методу?

5. Які існують правила заміни методів у Java?

ЛЕКЦІЯ 13. ПОЛІМОРФІЗМ У JAVA

1. Поняття поліморфізму
2. Поліморфізм під час виконання в Java
3. Модернізація
4. Приклади поліморфізму під час виконання в Java

Поліморфізм у Java – це поняття, за допомогою якого ми можемо виконувати *одну дію різними способами*. Поліморфізм походить від двох грецьких слів: *poly* і *morphs*. Слово "poly" означає багато, а "morphs" означає форми. Отже, поліморфізм означає багато форм.

У Java існує два типи поліморфізму: поліморфізм під час компіляції та поліморфізм під час виконання. Ми можемо виконати поліморфізм у Java методом перевантаження методу та заміни методу.

Якщо ви перевантажуєте статичний метод у Java, це приклад поліморфізму часу компіляції. Тут ми зосередимось на поліморфізмі під час виконання в Java.

Поліморфізм під час виконання в Java

Поліморфізм середовища виконання або динамічна розсилка методів – це процес, у якому виклик перевизначеного методу вирішується під час виконання, а не під час компіляції.

У цьому процесі викликається перезаписаний метод через еталонну змінну суперкласу. Визначення методу для виклику ґрунтується на об’єкті, на який посилається посилальна змінна.

Давайте спочатку розберемося в оновленні перед поліморфізмом під час виконання.

Модернізація

Якщо посилальна змінна батьківського класу посилається на об’єкт дочірнього класу, це називається оновленням. Наприклад:

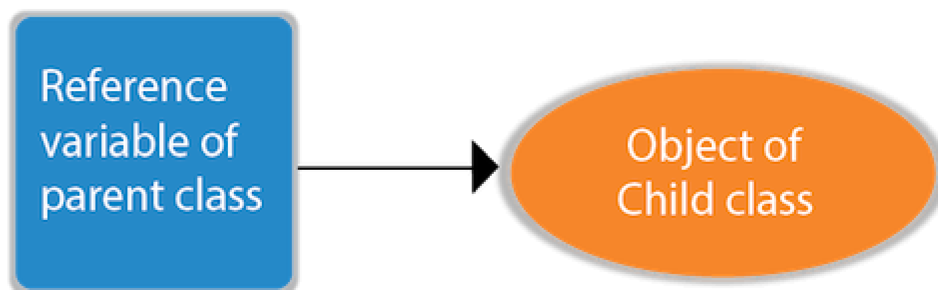

Рис. 28. Модернізація в Java

```
class A{}
class B extends A{}
A a = new B(); // оновлення
```

Для оновлення ми можемо використовувати еталонну змінну типу класу або типу інтерфейсу. Наприклад:

```
interface I{}
class A{}
class B extends A implements I{}
```

Тут відносини класу B будуть такими:

```
B IS-A A
B IS-A I
B IS-A Object
```

Оскільки Object є кореневим класом усіх класів на Java, тому ми можемо написати B IS-A Object.

Приклад поліморфізму під час виконання в Java

У цьому прикладі ми створюємо два класи Bike та Splendor. Клас Splendor розширює клас Bike і замінює його метод run(). Ми викликаємо метод run за допомогою посилальної змінної класу Parent. Оскільки він посилається на об'єкт підкласу, а метод підкласу замінює метод класу Parent, метод підкласу викликається під час виконання.

Оскільки виклик методу визначається компілятором JVM, він відомий як поліморфізм під час виконання.

```
class Splendor extends Bike{
    void run(){
        // Друк "безпечний біг на 60 км"
        System.out.println("running safely with 60km");
    }
    public static void main(String args[]){
```

```
        Bike b = new Splendor(); // оновлення
        b.run();
    }
}
```
Виведення на екран:

```
running safely with 60km
```

Приклад поліморфізму під час виконання в Java: банк

Розглянемо сценарій, коли Банк – це клас, який надає метод отримання процентної ставки. Однак процентна ставка може відрізнятися залежно від банків. Наприклад, банки SBI, ICICI та AXIS надають процентні ставки 8,4%, 7,3% та 9,7%.

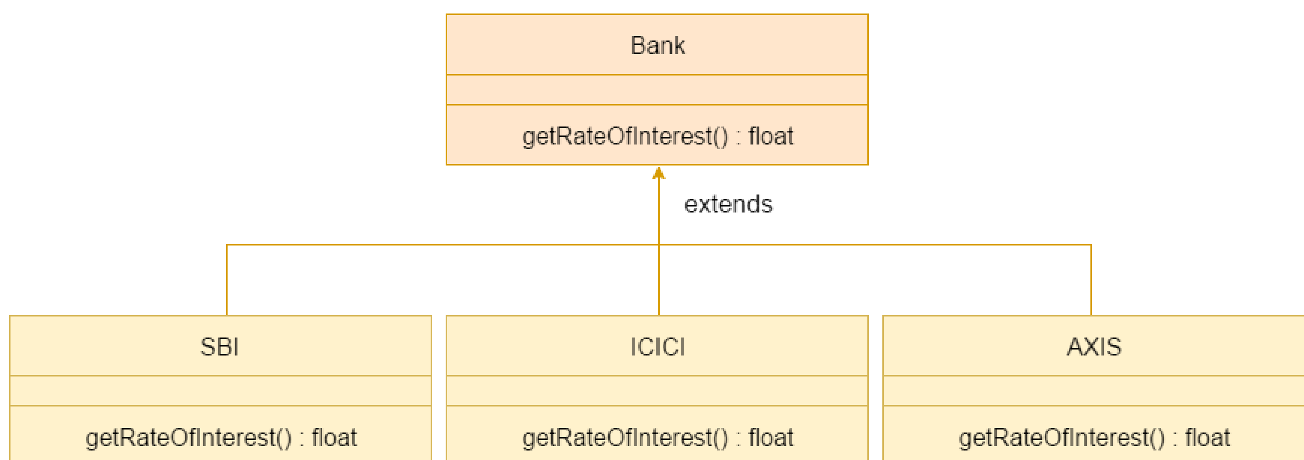

Рис. 29. Приклад поліморфізму в Java

Примітка: Цей приклад також наведено при розгляді перевантаження методу, але оновлення не було.

```
class Bank{
    float getRateOfInterest(){
        return 0;
    }
}
class SBI extends Bank{
    float getRateOfInterest(){
        return 8.4f;
    }
}
class ICICI extends Bank{
    float getRateOfInterest(){
        return 7.3f;
    }
```

```
}
class AXIS extends Bank{
    float getRateOfInterest(){
        return 9.7f;
    }
}
class TestPolymorphism{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("SBI Rate of Interest:
"+b.getRateOfInterest());
        b=new ICICI();
        System.out.println("ICICI Rate of Interest:
"+b.getRateOfInterest());
        b=new AXIS();
        System.out.println("AXIS Rate of Interest:
"+b.getRateOfInterest());
    }
}
```

Виведення на екран:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Приклад поліморфізму середовища виконання в Java: Shape

```
class Shape{
    void draw(){
        System.out.println("drawing...");
    }
}
class Rectangle extends Shape{
    void draw(){
        System.out.println("drawing rectangle...");
    }
}
class Circle extends Shape{
    void draw(){
        System.out.println("drawing circle...");
    }
```

```
}
class Triangle extends Shape{
    void draw(){
        System.out.println("drawing triangle...");
    }
}
class TestPolymorphism2{
    public static void main(String args[]){
        Shape s;
        s=new Rectangle();
        s.draw();
        s=new Circle();
        s.draw();
        s=new Triangle();
        s.draw();
    }
}
```

Виведення на екран:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Приклад поліморфізму середовища виконання в Java: Animal

```
class Animal{
    void eat(){
        System.out.println("eating...");
    }
}
class Dog extends Animal{
    void eat(){
        System.out.println("eating bread...");
    }
}
class Cat extends Animal{
    void eat(){
        System.out.println("eating rat...");
    }
}
class Lion extends Animal{
    void eat(){
```

```
            System.out.println("eating meat...");
        }
    }
    class TestPolymorphism3{
        public static void main(String[] args){
            Animal a;
            a=new Dog();
            a.eat();
            a=new Cat();
            a.eat();
            a=new Lion();
            a.eat();
        }
    }
```

Виведення на екран:

```
eating bread...
eating rat...
eating meat...
```

Поліморфізм середовища виконання в Java з полем даних

Перевизначається метод, а не члени даних, тому поліморфізм під час виконання не може бути досягнутий членами даних.

У наведеному нижче прикладі обидва класи мають обмеження швидкості для члена даних. Ми отримуємо доступ до члена даних за допомогою посилальної змінної класу Parent, яка посилається на об'єкт підкласу. Оскільки ми звертаємось до елемента даних, який не перевизначається, він завжди матиме доступ до члена даних класу Parent.

Правило: Поліморфізм під час виконання не може бути досягнутий за допомогою полів даних.

```
     class Bike{
     int speedlimit = 90;
    }
    class Honda3 extends Bike{
     int speedlimit = 150;

        public static void main(String args[]){
            Bike obj=new Honda3();
            System.out.println(obj.speedlimit); // 90
        }
```

```
}
```

Виведення на екран:

```
90
```

Поліморфізм середовища виконання в Java з багаторівневим успадкуванням

Давайте розглянемо простий приклад поліморфізму під час виконання з багаторівневим успадкуванням.

```
class Animal{
    void eat(){
        System.out.println("eating");
    }
}
class Dog extends Animal{
    void eat(){
        System.out.println("eating fruits");
    }
}
class BabyDog extends Dog{
    void eat(){
        System.out.println("drinking milk");
    }
    public static void main(String args[]){
        Animal a1,a2,a3;
        a1=new Animal();
        a2=new Dog();
        a3=new BabyDog();
        a1.eat();
        a2.eat();
        a3.eat();
    }
}
```

Виведення на екран:

```
eating
eating fruits
drinking Milk
```

Розглянемо ще один приклад поліморфізму під час виконання з багаторівневим успадкуванням.

```
class Animal{
    void eat(){
```

```
            System.out.println("animal is eating...");
        }
}
class Dog extends Animal{
        void eat(){
            System.out.println("dog is eating...");
        }
}
class BabyDog1 extends Dog{
        public static void main(String args[]){
            Animal a=new BabyDog1();
            a.eat();
        }
}
```

Виведення на екран:

```
dog is eating...
```

Оскільки BabyDog не замінює метод eat(), тому викликається метод eat() класу Dog.

Контрольні запитання

1. Що таке поліморфізм?
2. Навіщо використовувати поліморфізм у Java?
3. Що таке перевизначення методу?
4. Що таке перезавантаження методу?
5. Яка відмінність між перевизначенням і перезавантаженням методу? Наведіть приклади.

ЛЕКЦІЯ 14. АБСТРАКТНИЙ КЛАС У JAVA

План

1. Абстракція в Java
2. Абстрактний клас
3. Абстрактний метод
4. Абстрактний клас, що містить конструктор, поля та методи

Клас, оголошений за допомогою ключового слова *abstract*, називається *абстрактним*. Він може містити абстрактні та неабстрактні методи (метод з тілом).

Перш ніж вивчити абстрактний клас у Java, давайте спочатку розберемося в абстракції в Java.

Абстракція в Java

Абстракція – це процес приховування деталей реалізації та показ лише функціональності для користувача.

Цей спосіб показує користувачеві лише важливі речі і приховує внутрішні деталі, наприклад, надсилання SMS, де ви вводите текст і надсилаєте повідомлення. Ви не знаєте внутрішньої обробки доставки повідомлень.

Абстракція дозволяє зосередитися на тому, що є об'єктом замість того, що він робить.

Шляхи досягнення абстракції

Існує два способи досягнення абстракції в Java:

– нотація (від 0 до 100%);

– інтерфейс (100%).

Абстрактний клас

Абстрактний клас – це клас, який містить методи, що не мають реалізації. Абстрактний клас створюється з метою створення спільного інтерфейсу між різними реалізаціями класів, які будуть породжені від абстрактного класу. Абстрактний клас створюється для визначення деяких спільних рис класів які будуть визначати конкретну реалізацію в породжених від нього класах.

Особливості абстрактного класу

1. Абстрактний клас повинен бути оголошений за допомогою ключового слова *abstract*.

2. Він може мати абстрактні та неабстрактні методи.

3. Заборонено (немає сенсу) створювати об'єкт абстрактного класу.

4. Він може мати як конструктори, так і статичні методи.

5. Він може мати остаточні (*final*) методи, які змусять підклас не змінювати тіло методу.

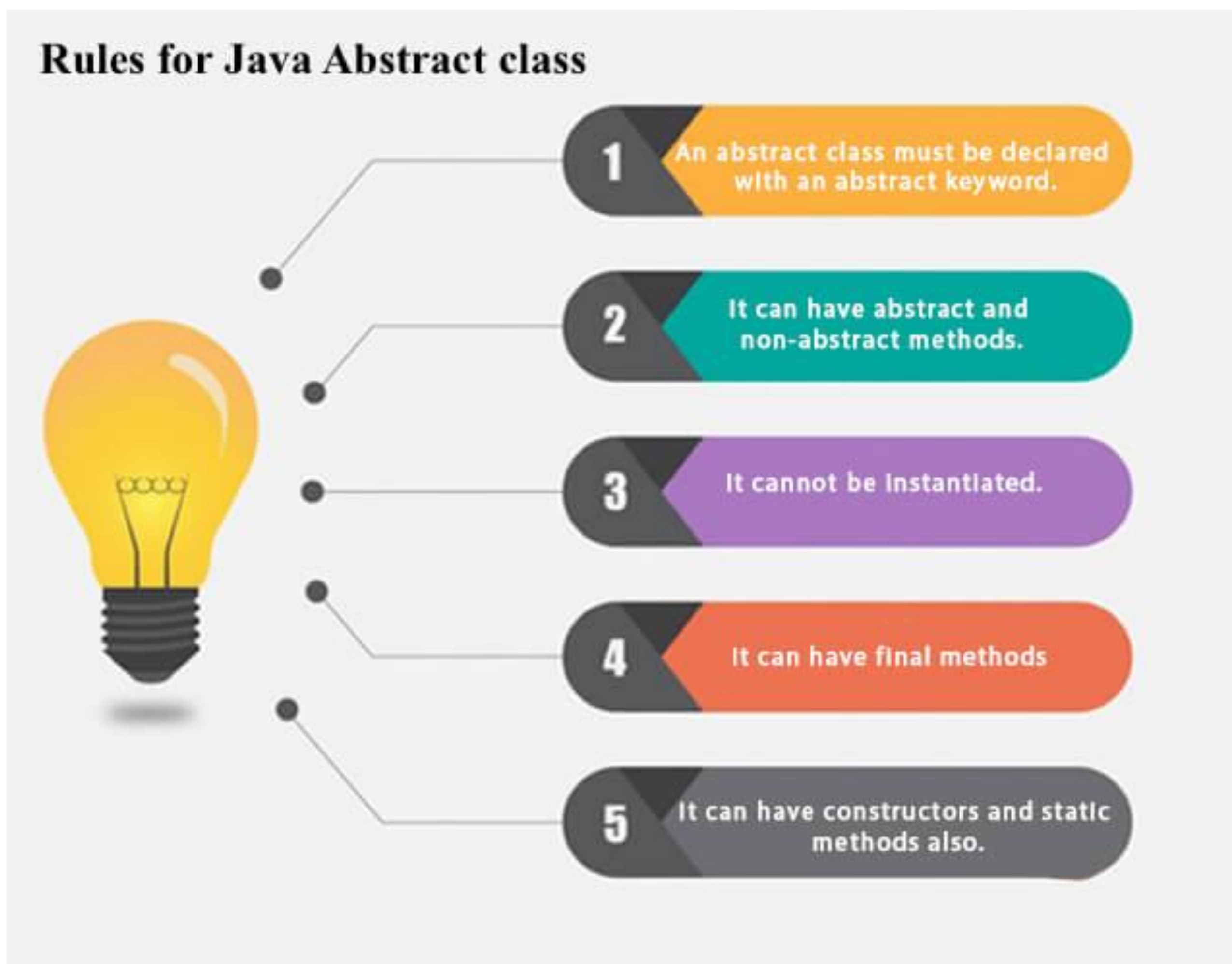

Рис. 30. Правила для абстрактного класу

Приклад абстрактного класу

abstract class A{}

Абстрактний метод

Абстрактний метод – це метод, реалізація якого в програмі не має ніякого змісту. Абстрактний метод – це тільки оголошення форми (інтерфейсу), а не реалізація. Як і у випадку з абстрактним класом, абстрактний метод починається з ключового слова *abstract*.

Якщо у класі оголошено абстрактний метод, то клас також вважається абстрактним. У цьому випадку перед іменем класу також ставиться ключове слово abstract.

Приклад абстрактного методу

abstract void printStatus(); // немає тіла методу та реалізації

Приклад абстрактного класу, який має абстрактний метод

У цьому прикладі Bike – це абстрактний клас, який містить лише один запуск абстрактного методу. Його реалізацію забезпечує клас Honda.

```
abstract class Bike{
    abstract void run();
}
class Honda4 extends Bike{
    void run(){
        // безпечно їздить
        System.out.println("running safely");
    }
    public static void main(String args[]){
        Bike obj = new Honda4();
        obj.run();
    }
}
```

Виведення на екран:

```
running safely
```

Розуміння реального сценарію абстрактного класу

У цьому прикладі *Shape* – це абстрактний клас, а його реалізація забезпечується класами *Rectangle* та *Circle*.

Здебільшого ми не знаємо про клас реалізації (який прихований для кінцевого користувача), а об'єкт класу реалізації надається фабричним *(factory)* методом.

Фабричний метод – це метод, який повертає екземпляр класу. Про фабричний метод ми дізнаємося пізніше.

У цьому прикладі, якщо ви створите екземпляр класу *Rectangle*, буде викликано метод draw() класу *Rectangle*.

Файл: TestAbstraction1.java

```
abstract class Shape{
    abstract void draw();
}
// У реальному сценарії реалізацію забезпечують інші класи,
// тобто вона невідома кінцевим користувачем
class Rectangle extends Shape{
    void draw(){
        // креслення прямокутника
        System.out.println("drawing rectangle");
    }
}
class Circle1 extends Shape{
    void draw(){
```

```
            // креслення кола
            System.out.println("drawing circle");
      }
}
// У реальному сценарії метод викликається програмістом
// або користувачем
class TestAbstraction1{
      public static void main(String args[]){
            // У реальному сценарії об'єкт надається через метод,
            // наприклад, метод getShape()
            Shape s = new Circle1();
            s.draw();
      }
}
```

Виведення на екран:

```
drawing circle
```

Ще один приклад абстрактного класу в Java

Файл: TestBank.java

```
abstract class Bank{
      abstract int getRateOfInterest();
}
class SBI extends Bank{
      int getRateOfInterest(){
            return 7;
      }
}
class PNB extends Bank{
      int getRateOfInterest(){
            return 8;
      }
}
class TestBank{
      public static void main(String args[]){
            Bank b;
            b=new SBI();
            // Процентна ставка
            System.out.println("Rate of Interest is:
"+b.getRateOfInterest()+" %");
            b=new PNB();
```

```
            // Процентна ставка
            System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
      }
}
```

Виведення на екран:

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Абстрактний клас, що містить конструктор, поля та методи

Абстрактний клас може мати поле, абстрактний метод, тіло методу (неабстрактний метод), конструктор і навіть метод main().

Файл: TestAbstraction2.java

```
// Приклад абстрактного класу, який має абстрактні
// та неабстрактні методи
abstract class Bike{
      Bike(){
            // велосипед створено
            System.out.println("bike is created");
      }
      abstract void run();
      void changeGear(){
            // передача змінена
            System.out.println("gear changed");
      }
}
// Створення дочірнього класу, який успадковує
// абстрактний клас
class Honda extends Bike{
      void run(){
            // безпечно працює..
            System.out.println("running safely..");
      }
}
// Створення класу TestAbstraction2, який викликає абстрактні
// та неабстрактні методи
class TestAbstraction2{
      public static void main(String args[]){
            Bike obj = new Honda();
```

```
        obj.run();
        obj.changeGear();
    }
}
```

Виведення на екран:

```
bike is created
running safely..
gear changed
```

Правило. Якщо в класі є абстрактний метод, цей клас має бути абстрактним.

```
class Bike12{
    abstract void run();
}
```

Виведення на екран:

```
compile time error
```

Правило. Якщо ви розширюєте абстрактний клас, який має абстрактний метод, ви повинні або надати реалізацію методу, або зробити цей клас абстрактним.

Ще один реальний сценарій абстрактного класу

Абстрактний клас також можна використовувати для забезпечення певної реалізації інтерфейсу. У такому випадку кінцевий користувач може не бути змушений перевизначати всі методи інтерфейсу.

Примітка. Якщо ви новачок у Java, спочатку вивчіть інтерфейс і пропустіть цей приклад.

```
interface A{
    void a();
    void b();
    void c();
    void d();
}
abstract class B implements A{
    public void c(){
        System.out.println("I am c");
    }
}
class M extends B{
    public void a(){
        System.out.println("I am a");
```

```
	}
	public void b(){
		System.out.println("I am b");
	}
	public void d(){
		System.out.println("I am d");
	}
}
class Test5{
	public static void main(String args[]){
		A a = new M();
		a.a();
		a.b();
		a.c();
		a.d();
	}
}
```

Виведення на екран:

```
I am a
I am b
I am c
I am d
```

Контрольні запитання

1. Що таке абстракція даних?
2. Що таке абстрактний клас? Наведіть приклад.
3. Що таке абстрактний метод?
4. Чи може мати абстрактний клас неабстрактні методи?
5. З якою метою використовують абстрактні класи і методи?

ЛЕКЦІЯ 15. ІНТЕРФЕЙС У JAVA

1. Поняття інтерфейсу
2. Множинне успадкування за допомогою інтерфейсу
3. Інтерфейс з тегами
4. Вкладений інтерфейс
5. Різниця між абстрактним класом та інтерфейсом

Поняття інтерфейсу в Java

Інтерфейс у Java є проєкт класу. Він має статичні константи та абстрактні методи.

Інтерфейс у Java є *механізмом досягнення абстракції*. В інтерфейсі Java можуть бути лише абстрактні методи, а не тіло методу. Він використовується для досягнення абстракції та множинного успадкування.

Іншими словами, можна сказати, що інтерфейс може мати абстрактні методи та змінні. Він не може мати тіло методу.

Інтерфейс Java також *представляє відносини IS-A*.

Його не можна створити так само, як абстрактний клас.

Починаючи з Java 8, ми можемо мати *стандартні та статичні методи* в інтерфейсі.

Починаючи з Java 9, ми можемо мати *приватні методи* в інтерфейсі.

Навіщо використовувати інтерфейс у Java?

Існують в основному три причини використовувати інтерфейс. Вони наведені нижче.

Використовується для досягнення абстракції.

За допомогою інтерфейсу ми можемо підтримувати функціональність множинного успадкування.

Його можна використовувати для досягнення вільного зчеплення.

Рис. 31. Причини використання інтерфейсу

Як оголосити інтерфейс?

Інтерфейс оголошується за допомогою ключового слова *interface*. Він забезпечує повну абстракцію; це означає те, що всі методи в інтерфейсі оголошені з порожнім тілом, і всі поля є загальнодоступними, статичними та фінальними за замовчуванням. Клас, який реалізує інтерфейс, повинен реалізувати всі методи, оголошені в інтерфейсі.

Синтаксис:

```
interface <ім'я_інтерфейсу>{
    // оголошуємо статичні поля
    // оголошуємо методи, які абстрактні
    // за замовчуванням
}
```

Покращення інтерфейсу в Java 8

Починаючи з Java 8, інтерфейс може мати стандартні та статичні методи, які будуть розглянуті далі.

Внутрішнє доповнення компілятором

Компілятор Java додає ключові слова *public* та *abstract* перед методами інтерфейсу. Крім того, він додає ключові слова *public*, *static* та *final* перед полями даних.

Іншими словами, поля інтерфейсу є загальнодоступними, статичними та остаточними за замовчуванням, а методи є відкритими та абстрактними.

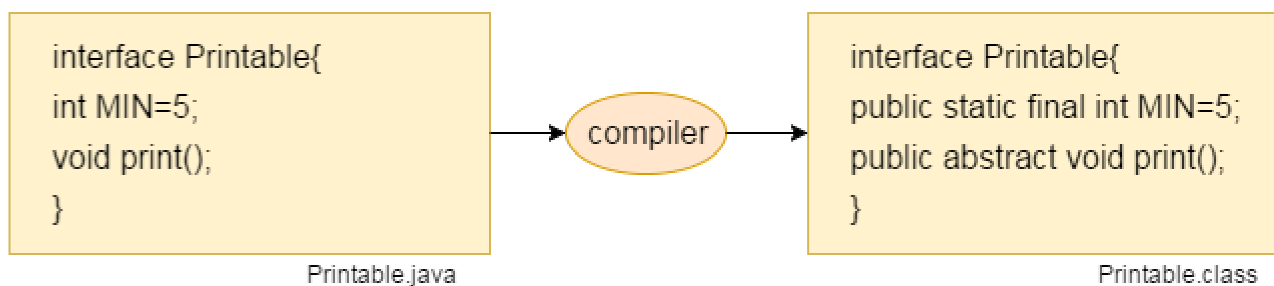

Рис. 32. Додавання компілятором Java ключових слів в інтерфейсі

Зв'язок між класами та інтерфейсами

Як показано нижче на рисунку, клас розширює інший клас, інтерфейс розширює інший інтерфейс, але *клас реалізує інтерфейс*.

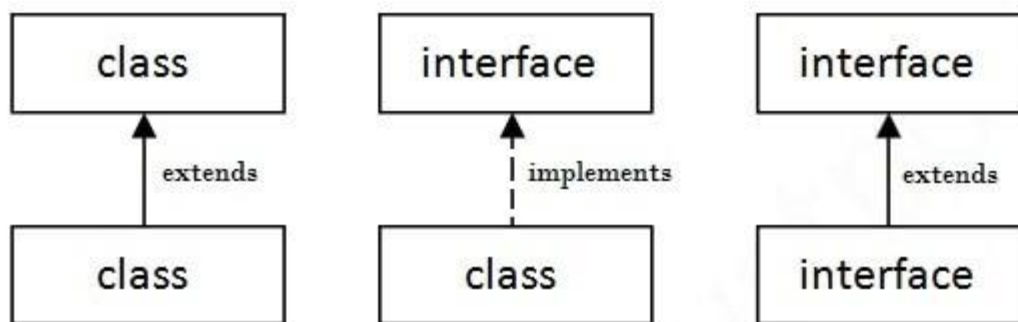

Рис. 33. Зв’язок між класами та інтерфейсами

Приклад інтерфейсу в Java

У цьому прикладі інтерфейс Printable має лише один метод, і його реалізація надається в класі A6.

```
interface printable{
    void print();
}
class A6 implements printable{
    public void print(){
        System.out.println("Hello");
    }
    public static void main(String args[]){
        A6 obj = new A6();
        obj.print();
    }
}
```

Виведення на екран:

```
Hello
```

Приклад інтерфейсу: Drawable

У цьому прикладі інтерфейс *Drawable* має лише один метод. Його реалізація забезпечується класами *Rectangle* і *Circle*. У реальному сценарії інтерфейс визначається одним користувачем, але його реалізацію забезпечують інші користувачі. Частина реалізації прихована користувачем, який використовує інтерфейс.

Файл: TestInterface1.java

```
// Оголошення інтерфейсу: першим користувачем
interface Drawable{
    void draw();
```

```
}
// Реалізація: другим користувачем
class Rectangle implements Drawable{
    public void draw(){
        System.out.println("drawing rectangle");
    }
}
class Circle implements Drawable{
    public void draw(){
        System.out.println("drawing circle");
    }
}
// Використання інтерфейсу: третім користувачем
class TestInterface1{
    public static void main(String args[]){
        // У реальному сценарії об’єкт надається методом,
        // наприклад getDrawable()
        Drawable d = new Circle();
        d.draw();
    }
}
```

Виведення на екран:

```
drawing circle
```

Приклад інтерфейсу: Bank

Давайте розглянемо ще один приклад класу, який забезпечує реалізацію інтерфейсу банку.

Файл: TestInterface2.java

```
interface Bank{
    float rateOfInterest();
}
class SBI implements Bank{
    public float rateOfInterest(){
        return 9.15f;
    }
}
class PNB implements Bank{
    public float rateOfInterest(){
        return 9.7f;
    }
```

```
}
class TestInterface2{
    public static void main(String[] args){
        Bank b = new SBI();
        // Процентна ставка
        System.out.println("ROI: "+b.rateOfInterest());
    }
}
```

Виведення на екран:

```
ROI: 9.15
```

Множинне успадкування за інтерфейсом у Java

Якщо клас реалізує декілька інтерфейсів або інтерфейс розширює декілька інтерфейсів, це називається *множинним успадкуванням*.

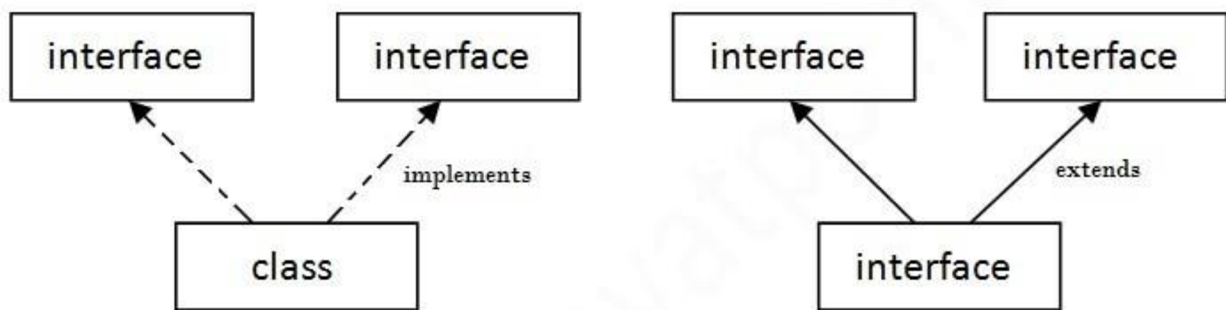

Рис. 34. Множинний інтерфейс у Java

```
interface Printable{
    void print();
}
interface Showable{
    void show();
}
class A7 implements Printable,Showable{
    public void print(){
        System.out.println("Hello");
    }
    public void show(){
        System.out.println("Welcome");
    }
    public static void main(String args[]){
```

```
        A7 obj = new A7();
        obj.print();
        obj.show();
    }
}
```

Виведення на екран:

```
Hello
Welcome
```

Множинне успадкування не підтримується через клас у Java, але це є можливим за допомогою інтерфейсу, чому?

Як ми пояснили в темі про спадкування, множинне успадкування не підтримується у випадку класу через двозначність. Однак воно підтримується у випадку інтерфейсу, оскільки немає двозначності. Це тому, що його реалізація забезпечується класом реалізації. Наприклад:

```
interface Printable{
    void print();
}
interface Showable{
    void print();
}
class TestInterface3 implements Printable, Showable{
    public void print(){
        System.out.println("Hello");
    }
    public static void main(String args[]){
        TestInterface3 obj = new TestInterface3();
        obj.print();
    }
}
```

Виведення на екран:

```
Hello
```

Як ви можете бачити у наведеному вище прикладі, інтерфейси *Printable* і *Showable* мають однакові методи, але його реалізація надається класом TestTnterface1, тому немає двозначності.

Успадкування інтерфейсу

Клас реалізує інтерфейс, але один інтерфейс розширює інший інтерфейс.

```
interface Printable{
    void print();
```

```
}
interface Showable extends Printable{
	void show();
}
class TestInterface4 implements Showable{
	public void print(){
		System.out.println("Hello");
	}
	public void show(){
		System.out.println("Welcome");
	}
	public static void main(String args[]){
		TestInterface4 obj = new TestInterface4();
		obj.print();
		obj.show();
	}
}
```

Виведення на екран:

```
Hello
Welcome
```

Метод за замовчуванням у Java 8 в інтерфейсі

Починаючи з Java 8, ми можемо мати тіло методу в інтерфейсі. Але нам потрібно зробити його методом за замовчуванням. Давайте розглянемо приклад:

Файл: TestInterfaceDefault.java

```
interface Drawable{
	void draw();
	default void msg(){
		System.out.println("default method");
	}
}
class Rectangle implements Drawable{
	public void draw(){
		System.out.println("drawing rectangle");
	}
}
class TestInterfaceDefault{
	public static void main(String args[]){
		Drawable d = new Rectangle();
```

```
        d.draw();
        d.msg();
    }
}
```
Виведення на екран:

```
drawing rectangle
default method
```

Статичний метод в інтерфейсі

Починаючи з Java 8, ми можемо мати статичний метод в інтерфейсі. Давайте розглянемо приклад:

Файл: TestInterfaceStatic.java

```
interface Drawable{
    void draw();
    static int cube(int x){
        return x*x*x;
    }
}
class Rectangle implements Drawable{
    public void draw(){
        System.out.println("drawing rectangle");
    }
}
class TestInterfaceStatic{
    public static void main(String args[]){
        Drawable d = new Rectangle();
        d.draw();
        System.out.println(Drawable.cube(3));
    }
}
```
Виведення на екран:

```
drawing rectangle
27
```

Інтерфейс з тегами

Інтерфейс, який не містить методів, називається *маркером* або *інтерфейсом з тегами*, наприклад, Serializable, Cloneable, Remote тощо. Вони використовуються для надання певної важливої інформації JVM, щоб JVM могла виконувати корисні операції.

```
// Як написати інтерфейс, який можна перетворити у серію?
public interface Serializable{
}
```

Вкладений інтерфейс у Java

Примітка: інтерфейс може мати інший інтерфейс, який називається *вкладеним інтерфейсом*. Ми докладно говорили про це у темі про вкладені класи. Наприклад:

```
interface printable{
  void print();
  interface MessagePrintable{
    void msg();
  }
}
```

Різниця між абстрактним класом та інтерфейсом

Абстрактний клас і інтерфейс використовуються для досягнення абстракції, де ми можемо оголошувати абстрактні методи. Абстрактний клас і інтерфейс не можуть бути створені.

Але є багато відмінностей між абстрактним класом та інтерфейсом, які наведені нижче.

Анотація класу	Інтерфейс
1) Абстрактний клас може мати абстрактні та неабстрактні методи.	Інтерфейс може мати тільки абстрактні методи. Починаючи з Java 8, він також може мати стандартні та статичні методи.
2) Абстрактний клас не підтримує множинну спадковість.	Інтерфейс підтримує множинне успадкування.
3) Абстрактний клас може мати фінальні, нефінальні, статичні та нестатичні змінні.	Інтерфейс має лише статичні та фінальні змінні.
4) Абстрактний клас може забезпечити реалізацію інтерфейсу.	Інтерфейс не може забезпечити реалізацію абстрактного класу.
5) Ключове слово abstract використовується для оголошення абстрактного класу.	Ключове слово interface використовується для оголошення інтерфейсу.

6) Абстрактний клас може розширити інший клас Java та реалізувати кілька інтерфейсів Java.	Інтерфейс може розширити тільки інший інтерфейс Java.
7) Абстрактний клас можна розширити за допомогою ключового слова “*extends*”.	Інтерфейс може бути реалізований з використанням ключових слів “*implements*”.
8) Абстрактний клас Java може мати елементи таких як приватний, захищений тощо.	Елементи інтерфейсу Java є загальнодоступними.

9) Приклад:
public abstract class Shape{
public abstract void draw();
}
	Приклад:
	public interface Drawable{
	void draw();
	}

Просто, абстрактний клас досягає часткової абстракції (від 0 до 100%), тоді як інтерфейс досягає повної абстракції (100%).

Приклад абстрактного класу та інтерфейсу в Java

Давайте розглянемо простий приклад, де ми використовуємо інтерфейс та абстрактний клас.

```
interface A{
    void a(); // за замовчуванням, загальнодоступний і абстрактний
    void b();
    void c();
    void d();
}
// Створення абстрактного класу, що забезпечує реалізацію
// одного методу інтерфейсу A
abstract class B implements A{
    public void c(){
        System.out.println("I am C");
    }
}
// Створюючи підклас абстрактного класу, тепер нам
// потрібно забезпечити реалізацію інших методів
class M extends B{
    public void a(){
        System.out.println("I am a");
    }
```

```
	public void b(){
		System.out.println("I am b");
	}
	public void d(){
		System.out.println("I am d");
	}
}
// Створення тестового класу, який викликає методи
// інтерфейсу A
class Test51{
	public static void main(String args[]){
		A a = new M();
		a.a();
		a.b();
		a.c();
		a.d();
	}
}
```

Виведення на екран:

```
I am a
I am b
I am C
I am d
```

Контрольні запитання

1. Що таке інтерфейс у Java?
2. Яке призначення інтерфейсу?
3. Яка відмінність між абстрактним класом й інтерфейсом?
4. Як отримати доступ до методів інтерфейсу?
5. Чи можна реалізувати множинне успадкування за допомогою інтерфейсу?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гаврилов А. В., Дегтярова О. А., Лезин И. А., Лезина И. В. Учебное пособие по языку Java. Самара: Изд-во Самарского государственного аэрокосмического университета, 2010. 175 с.

2. Спірінцева О. В., Литвинов О. А., Герасимов В. В. Java-технології та мобільні пристрої. Алгоритми і структури даних: навч. посіб. Д.: Вид-во ДНУ ім. О. Гончара, 2016. 140 с.

3. Java Tutorial. URL: https://www.w3schools.com/java/default.asp

4. Java. Классы. Объектно-ориентированное программирование. URL: https://metanit.com/java/tutorial/3.1.php

5. Підручник з Java. URL: https://www.javatpoint.com/java-tutorial

6. GDB online Debugger / Compiler. URL: https://www.onlinegdb.com/

7. Java – Учебники по программированию. URL: https://betacode.net/

8. Apache NetBeans. URL: https://netbeans.apache.org/download/index.html

9. Java Course. URL: http://java-course.ru/begin/introduce/

10. Java SE Downloads. URL: https://www.oracle.com/java/technologies/javase-downloads.html

11. Руководство JavaFX для начинающих – Hello JavaFX. URL: https://betacode.net/10623/javafx-tutorial-for-beginners

Навчальне видання

Муляр Вадим Петрович

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій