Лабораторна робота №3

Надідання класів і поліморфізм у мові Java

Мета лабораторної роботи:	1. Вивчити основні концепції відносин між класами в Java. Вивчити ставлення агрегування та успадкування. 2. Вивчити прийоми програмування з використанням відношення спадкування класів мовою Java.

Зміст роботи.

1. Теоретичні відомості

1.1 Спадкування класів у Java

Короткі теоретичні відомості про успадкування класів Java можна подивитися тут

1.2 Приклад реалізації спадкування класів

Нижче наведено приклад реалізації суперкласу Employee та підкласу Manager. Вони наочно демонструють принцип успадкування класів. Реалізовані у кожному класі методи

typeEmployee() і toString() демонструють поліморфізм.
Приклад реалізації суперкласу та підкласу

2. Завдання на лабораторну роботу

Варіант вибирається за формулою V = (N mod K) + 1, де N - Ваш номер за академічним журналом групи, K = 6, а mod - операція "залишок від поділу на".

Необхідно виконати 2 завдання. Завдання 1 - загальне всім варіантів. Завдання 2 вибирається згідно з Варіантом.

Умови Завдання1 та Завдання 2 можна переглянути тут.

3. Звіт

Повинен містити: постановку завдання, програмний код розв'язання, результат роботи написаної програми

Завдання 1 та 2

Для всіх завдань необхідно розробити тестову програму, де демонструється робота зі створеними класами та тестуються їх методи.

Завдання 1. (для всіх варіантів)

Створити базовий клас "Людина".
Кожен об'єкт класу повинен містити такі дані:

- ПІБ,
- рік народження,
- підлога,
- статичне поле, що містить кількість створених об'єктів (передбачити статичний блок ініціалізації для цього поля у класі)
-

Клас повинен містити такі методи:

- Конструктор для ініціалізації інформації;
- Методи доступу (accessors) та модифікуючі методи (mutators) для ВСІХ полів класу;
- Метод WhoIAm(), що виводить на консольний висновок назву класу (ідентифікатор, вигаданий Вами для іменування класу «Людина»);
- Метод toString(), що повертає у вигляді рядка інформацію про людину (поточний об'єкт) у довільному вигляді.
- Статичний метод HowMany(), що повертає кількість створених екземплярів класу.

Завдання 2.

Створити клас із зазначеними елементами згідно з варіантом, успадкувавши його від класу «Людина».

Крім зазначених у завданні кожного варіанта елементів, необхідно додатково реалізувати:

- статичне поле, що містить кількість створених об'єктів (передбачити статичний блок ініціалізації для цього поля у класі)
- Конструктор для ініціалізації інформації;
- Методи доступу (accessors) та модифікуючі методи (mutators) для ВСІХ полів класу;
- Метод WhoIAm(), що виводить на консольний висновок назву класу (ідентифікатор, придуманий Вами для іменування класу згідно з завданням варіанта)
- Метод toString(), що повертає у вигляді рядка інформацію про поточний об'єкт у довільному вигляді.
- Статичний метод HowMany(), що повертає кількість створених екземплярів класу (згідно з варіантом завдання).
- Методи введення та виведення інформації про об'єкт з консолі (можна використовувати акцесори та мутатори)

Варіант 1.

Створити похідний клас «Людина» клас Студент, який має додаткові дані:

- рік вступу,
- № залікової книжки,
- кількість дисциплін

- дисципліни, що вивчаються (динамічний масив),
- середній бал.

Клас повинен містити такі методи:

- додавання дисциплін,
- розрахунок середнього балу

Варіант 2

Створити похідний від класу «Людина» клас «Інженер», який має додаткові дані:
рік закінчення, ВНЗ,

- спеціальність,
- тип диплома,
- тип навчання,
- перекваліфікація (динамічний масив),
- місце роботи,
- заробітна плата,
- базовий оклад - статична константа.

Клас повинен містити такі методи:

- розрахунок заробітної плати (базовий оклад плюс надбавка залежно від кількості перекваліфікацій)
- розрахунок щорічного доходу (приймає кількість відпрацьованих місяців як параметр)
- додавання інформації про перекваліфікацію

Варіант 3.

Створити похідний від класу «Людина» клас «Користувач_бібліотеки», що містить такі дані:

- номер читацького квитка,
- дата видачі,
- перелік прочитаних книг (динамічний масив) — інформація про книгу зберігається у вигляді рядка з вибраним вмістом,
- щомісячний читацький внесок,
- статус користувача.

Клас повинен містити такі методи:

- розрахунок знижки (залежить від кількості прочитаних книг),
- розрахунок щоквартального читацького внеску,
- додавання інформації про прочитані книги.

Варіант 4.

Створити похідний від класу «Людина» клас «Мандрівник», що містить такі дані:

- назва туру,
- дата початку туру,
- дата закінчення туру,
- рівень обслуговування ("***", "****", "****, all inclusive", "*****", "*****, all inclusive", …),

- перелік відвіданих міст та географічних місць (динамічний масив) — інформацію зберігайте у форматі рядка у довільному вигляді,
- ціна відвідування, проживання та екскурсій по кожному об'єкту (динамічний масив – розмірність відповідає розмірності масиву відвіданих об'єктів),

Клас повинен містити такі методи:

- розрахунок вартості туру,
- розрахунок знижки (залежить від кількості відвіданих місць та рівня обслуговування),
- метод IsAllInclusive() – наявність опції «все включено» (визначити з поля рівня обслуговування),
- додавання інформації про відвідані місця.

Варіант 5.

Створити похідний від класу «Людина» клас «Космонавт», що містить такі дані:

- військове звання,
- кількість льотних годин,
- дати тренувальних польотів (динамічний масив),
- дати реальних польотів у космос (□) (динамічний масив),
- базовий посадовий оклад.

Клас повинен містити такі методи:

- розрахунок зарплати (посадовий оклад плюс надбавки по 1% за кожен тренувальний політ плюс надбавки по 500% за кожен реальний політ),

- IsHerro() – метод, який повертає, чи є космонавт героєм країни (в основному присвоюється за 1 реальний політ у космос, у деяких країнах (Росія) – за 2 реальні польоти),
- додавання інформації про тренувальні та реальні польоти.

Зауваження. При введенні даних зверніть увагу, що в державі Україна було здійснено 1 реальний політ 1 космонавтом.

Варіант 6

Створити похідний від класу «Людина» клас «Програміст», який містить такі дані:

- рівень (junior, middle, senior, team leader),
- кількість відпрацьованих годин за останні 11 місяців (динамічний масив),
- список мов та технологій, якими володіє програміст,
- базова оплата за годину роботи.

Клас повинен містити такі методи:

- розрахунок зарплати за кожен місяць (базова оплата * кількість годин на місяць) - результат - масив зарплат,
- розрахунок відпускних (зарплата за всі місяці + матеріальна допомога у розмірі базова оплата * 150 годин)
- розрахунок квартальної премії (за кожні три ПОВНИХ місяці - у розмірі Базова оплата

* Середня кількість годин за три місяці * Коефіцієнт. Коефіцієнт береться або 1, або 1.2 у разі знання C #, або 1.4 у разі знання Java - інформація береться зі списку технологій).

Приклад реалізації суперкласу та підкласу

```
public class Employee {// рядовий співробітник

private int tab_num;

public Employee(int id) {
tab_num = id;
}

public int getId() {
return tab_num;
}

public void typeEmployee() {
System.out.println("Стандартний співробітник компанії");
}

@Override
public String toString(){
return "Я рядовий співробітник з табельним номером" + Integer.toString
(tab_num);
}
}

public class Manager extends Employee {
private int idProject;

public Manager(int idc, int idp) {
super(idc); // виклик конструктора суперкласу
idProject = idp;
}

public int getIdProject() {
return idProject;
}

@Override
public void typeEmployee() {
System.out.println("Менеджер");
    }

@Override
public String toString(){
return "Я рядовий співробітник з табельним номером"
+Integer.toString(getId())
+" та керую проектом номер "
+ Integer.toString(idProject);
    }
}

public class Runner {  // test
public static void main(String[] args) {
Employee b1 = New Employee(7110);
Employee b2 = new Manager(9251, 31);
```

```
b1.typeEmployee();// виклик версії з класу Employee
b2.typeEmployee();// виклик версії з класу Manager
// А ось так не вийде!!
// System.out.println( b2.getIdProject());// помилка компіляції!
// -------------------
System.out.println( ((Manager) b2).getIdProject() );
Manager b3 = New Manager (9711, 35);
System.out.println(b3.getIdProject()); // 35
System.out.println(b3.getId()); // 9711
// Нижче неявно викликається toString()
System.out.println(b1);
System.out.println(b2);
System.out.println(b3);
}
}
```

СПАДЧИНА І ПОЛІМОРФІЗМ

успадкування

Відношення між класами, при якому характеристики одного класу (суперкласу) передаються іншому класу (підкласу) без їхнього повторного опису, називається успадкуванням.

Підклас успадковує змінні та методи суперкласу, використовуючи ключове слово extends. Клас може також реалізовувати будь-яке число інтерфейсів, використовуючи ключове слово – implements. Підклас має прямий доступ до всіх відкритих змінних та методів батьківського класу, начебто вони знаходяться у підкласі. Виняток становлять члени класу, помічені private (в усіх випадках) та «за замовчуванням» для підкласу в іншому пакеті. У будь-якому випадку (навіть якщо ключового слова extends немає) клас автоматично успадковує властивості суперкласу всіх класів – класу Object.

Множинне успадкування класів заборонено, хоча його аналог надає реалізація інтерфейсів, які є класами і містять опис набору методів, дозволяють задати поведінка об'єкта класу, реалізує ці інтерфейси. Наявність загальних методів, які мають бути реалізовані у різних класах, забезпечують їм подібну функціональність.

Підклас доповнює члени базового класу своїми змінними та методами, імена яких можуть частково співпадати з іменами членів суперкласу. Якщо імена методів збігаються, а параметри різняться, таке явище називається перевантаженням методів.

У підкласі можна оголосити (перевизначити) метод з тим самим ім'ям, списком параметрів і значенням, що повертається, що й у методу суперкласу.

Здатність посилання динамічно визначати версію перевизначеного методу в залежності від переданого посилання в повідомленні типу об'єкта називається поліморфізмом.

Поліморфізм є основою реалізації механізму динамічного чи «пізнього зв'язування».

У наступному прикладі перевизначуваний метод typeEmployee() знаходиться у двох класах Employee та Manager. Відповідно до принципу поліморфізму викликається метод, найближчий до поточного об'єкта.

```
/* Приклад # 1 : успадкування класу і перевизначення методу:
Employee.java: Manager.java: Runner.java */
package chapt04;

public class Employee {// рядовий співробітник
      private int id;
      public Employee(int idc) {
            super();/* за замовчуванням, необов'язковий
                        явний виклик конструктора
                        суперкласу */
            id = idc;
      }
      public int getId() {
            return id;
      }
      public void typeEmployee() {
            //...
            System.out.println("Працівник");
      }
}

package chapt04;
// співробітник із проектом, за який він відповідає

 public class Manager extends Employee {
      private int idProject;

      public Manager(int idc, int idp) {
        super(idc); /* виклик конструктора суперкласу
                        з параметром */
            idProject = idp;
      }
      public int getIdProject() {
            return idProject;
      }
      public void typeEmployee() {
```

```
            //…
            System.out.println("Менеджер");
      }
}
package chapt04;

public class Runner {
      public static void main(String[] args) {
            Employee b1 = New Employee(7110);
            Employee b2 = new Manager(9251, 31);
            b1.typeEmployee();// виклик версії з класу Employee
            b2.typeEmployee();// виклик версії з класу Manager
            // b2.getIdProject();// помилка
            компіляції!((Manager) b2).
            getIdProject(); Manager b3 = New
            Manager (9711, 35);
            System.out.println(b3.getIdProject());// 35
            System.out.println(b3.getId()); // 9711
      }
}
```

Об'єкт b1 створюється за допомогою виклику конструктора класу Employee, і, відповідно, виклику методу typeEmployee() викликається версія методу з класу Employee. При створенні об'єкта b2, посилання типу Employee ініціалізується об'єктом типу Manager. За такого способу ініціалізації посилання на суперклас отримує доступ до методів, перевизначених у підкласі.

При оголошенні полів, що збігаються за сигнатурою (ім'я, тип, область видимості) в суперкласі та підкласах їх значення не перевизначаються і ніяк не перетинаються, тобто існують в одному об'єкті незалежно один від одного. У цьому випадку завдання отримання необхідного значення певного поля, що належить класу в ланцюжку успадкування, лягає на програміста. Для доступу до полів поточного об'єкта можна використовувати покажчик this, для доступу до полів суперкласу – вказівник super. Інші можливості розглянуті в наступному прикладі:

/ приклад # 2: створення об'єкта підкласу та доступ до полів з однаковими іменами: Course.java: BaseCourse.java: Logic.java */*

```
package chapt04;

public class Course {
      public int id = 71;

      public Course() { System.out.println("конструктор
       класу Course"); id = getId();//!!!
       System.out.println("id="+id);
      }
      public int getId() {
            System.out.println("getId() класу Course");
            return id;
      }
}
package chapt04;

public class BaseCourse extends Course {
      public int id = 90;// так робити не слід!

      public BaseCourse() { System.out.println("конструктор
       класу BaseCourse"); System.out.println(" id=" +
       getId());
      }
      public int getId() {
          System.out.println("getId() класу BaseCourse");
          return id;
      }
}
package chapt04;
```

```
public classLogic {
      public static voidmain(String[] args) {
            Course objA = new BaseCourse();
            BaseCourse objB = new BaseCourse();
            System.out.println("objA: id=" + objA.id);
            System.out.println("objB: id=" + objB.id);
            Course objC = new Course();
      }
}
```

В результаті виконання цього коду послідовно буде виведено:

```
конструктор класу Course
getId() класу BaseCourse
 id=0
конструктор класу BaseCourse
getId() класу BaseCourse
 id=90
конструктор класу Course
getId() класу BaseCourse
 id=0
конструктор класу BaseCourse
getId() класу BaseCourse
 id=90objA:
id=0 objB:
id=90
конструктор класу Course
getId() класу Course
 id=71
```

Метод getId() міститься як у класі Course, так і в класі BaseCourse і є перевизначеним. Під час створення об'єкта класу BaseCourse одним із способів:

```
Course objA = new BaseCourse();
BaseCourse objB = new BaseCourse();
```

у будь-якому разі перед викликом конструктора BaseCourse() викликається конструктор класу Course. Але оскільки в обох випадках створюється об'єкт класу BaseCourse, то викликається метод getId(), оголошений у класі BaseCourse, який у свою чергу оперує полем id, ще не проініціалізованим для класу BaseCourse. Через війну id отримає значення за промовчанням, тобто. нуль.

Скориставшись перетворенням типів виду ((BaseCourse)objA).id або ((Course)objB).id, можна легко отримати доступ до поля id з відповідного класу.

Використання final

Не можна створити підклас для класу, оголошеного із специфікатором final:

```
// клас ConstCourse не може бути суперкласом
final classConstCourse {/*код*/}
// наступний клас неможливий
classBaseCourse extends ConstCourse {/*код*/}
```

Використання super і this

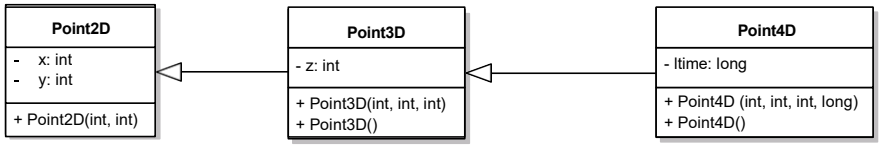
Ключове слово super використовується для виклику конструктора суперкласу та для доступу до члена суперкласу. Наприклад:

```
super(список_параметрів);/* виклик конструктора суперкласу
                              з передачею параметрів або без
неї*/super. id = 71; /* звернення до атрибуту
суперкласу */ super.getId(); // виклик методу
суперкласу
```

Друга форма super використовується для доступу з підкласу до змінної id суперкласу. Третя форма специфічна для Java і забезпечує виклик з підкласу перевизначеного методу суперкласу, причому якщо у суперкласі цей метод не визначений, то здійснюватиметься пошук по ланцюжку успадкування доти, доки метод не буде знайдено.

Кожен екземпляр класу має неявне посилання цього на себе, яке передається також і методам. Після цього метод «знає», який його об'єкт викликав. Замість звернення до атрибуту id у методах можна писати this.id, хоч і не обов'язково, оскільки записи id та this.id рівносильні.

Наступний код показує, як, використовуючи це, можна будувати одні конструктори на основі інших.



```
// Приклад # 3: this в конструкторі: Point2D.java, Point3D.java, Point4D.java
package chapt04;

public class Point2D {
	private int x, y;

	public Point2D(int x, int y) {
		this.x = x;//this використовується для присвоєння полям класу
		this.y = y;//x, y, значень параметрів конструктора x, y, z
	}
}
package chapt04;

public class Point3D extends Point2D {
	private int z;

	public Point3D(int x, int y, int z) {
		super(x, y);
		this.z = z;
	}
	public Point3D() {
	   this(-1,-1,-1);// виклик конструктора Point3D з параметрами
	}
}
package chapt04;

public class Point4D extends Point3D {
	private long time;

	public Point4D(int x, int y, int z, long time) {
		super(x, y, z);
		this. time = time;
	}
	public Point4D() {
		// за замовчуванням super();
	}
}
```

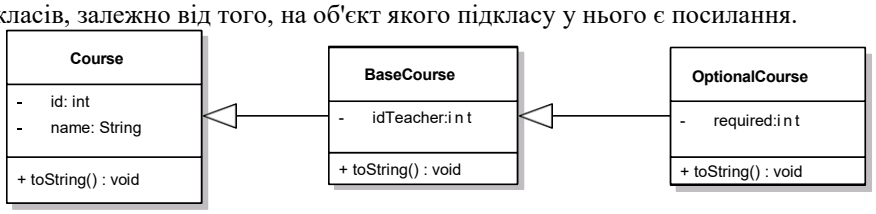
У класі Point3D другий конструктор для завершення ініціалізації об'єкта звертається до першого конструктору. Така конструкція застосовується у разі, коли клас потрібно додати конструктор за умовчанням з обов'язковим використанням вже існуючого конструктора.

Посилання this використовується в методі для уточнення того, про які саме змінних x, y і z йдеться в методі, а саме для доступу до змінних класу з методу, якщо в методі є локальні змінні з тим же ім'ям, що й у класу. Інструкція this() повинна бути єдиною в конструкторі, що викликає, і бути першою за рахунком виконуваною операцією.

Перевизначення методів та поліморфізм

Здатність Java робити вибір методу, виходячи з типу об'єкта під час виконання, називається пізнім зв'язуванням. При виклікі методу його пошук відбувається спочатку в даному класі, потім у суперкласі, поки метод не буде знайдений або досягнутий Object – суперклас для всіх класів.

Якщо два методи з однаковими іменами і значеннями, що повертаються знаходяться в одному класі, то списки їх параметрів повинні відрізнятися. Те саме стосується методів, успадкованих з суперкласу. Такі методи перевантажуються (overloading). При зверненні викликається метод, список параметрів якого збігається зі списком параметрів виклику. Якщо оголошення методу підкласу повністю, включаючи параметри, збігається з оголошенням методу суперкласу (що породжує класу), то метод підкласу перевизначає (overriding) метод суперкласу. Перевизначення методів є основою концепції динамічного зв'язування, що реалізує поліморфізм. Коли перевизначений метод викликається через посилання суперкласу, Java визначає, яку версію методу викликати, ґрунтуючись на типі об'єкта, на який є посилання. Отже, тип об'єкта визначає версію методу етапі виконання. У прикладі розглядається реалізація поліморфізму з урахуванням динамічного зв'язування. Так як супер клас містить методи, перевизначені підкласами, то об'єкт суперкласу буде викликати методи різних підкласів, залежно від того, на об'єкт якого підкласу у нього є посилання.



Мал. 4.1.Приклад реалізації поліморфізму

```
/* приклад # 4: динамічне зв'язування методів: Course.java: BaseCourse.java: OptionalCourse.java: DynDis-
```

```
patcher.java */
package chapt04;

public class Course {private
      int id; private
      String name;

      public Course(int i, String n) {
            id = i;
            name = n;
      }
      public String toString() {
            return "Назва:" + name + "(" + id + ")";
      }
}
package chapt04;

public class BaseCourse extends Course {
      private int idTeacher;

      public BaseCourse(int i, String n, int it) {
            super(i, n);
            idTeacher = it;
      }
      public String toString() {
            /* просто toString() не можна!
               метод буде викликати сам себе, що призведе
               до помилки під час виконання */
   return
      super.toString() + "виклад.(" + idTeacher + ")";
      }
}
package chapt04;

public class OptionalCourse extends BaseCourse {
      private boolean required;
      public OptionalCourse(int i, String n, int it,
                                          boolean r) {
            super(i, n, it);
            required = r;
      }
      public String toString() {
        return super.toString() + "required->" + required;
      }
}
package chapt04;

public class DynDispatcher{
      public void infoCourse(Course c) {
            System.out.println(c.toString());
            //System.out.println(c);//ідентично
      }
}
package chapt04;

public class Runner {
      public static void main(String[] args) {
            DynDispatcher d = new DynDispatcher();
            Course cc = new Course (7, "МА");
            d.infoCourse(cc);
            BaseCourse bc = new BaseCourse(71, "МП", 2531);
            d.infoCourse(bc);
            OptionalCourse oc =
                  new OptionalCourse(35, "ФА", 4128, true);
            d.infoCourse(oc);
      }
}
```

Результат:

```
Назва: МА(7)
Назва: МП(71) виклад.(2531)
Назва: ФА(35) виклад.(4128) required->true
```

Слід пам'ятати, що при виклику toString() звернення super завжди відбувається до найближчого суперкласу. Аналогічно при виклику super() у конструкторі звернення відбувається до відповідного конструктора безпосереднього суперкласу.

Основний висновок: Вибір версії перевизначеного методу проводиться на етапі виконання коду. Усі методи Java є віртуальними (ключове слово virtual, як і C++, немає).

Статичні методи можуть бути перевизначені в підкласі, але не можуть бути поліморфними, оскільки їх виклик не торкається об'єктів. Їх слід викликати лише з використанням імені класу.

Методи підставки

З п'ятої версії мови з'явилася можливість при перевизначенні методів вказувати інший тип значення, що повертається, в якості якого можна використовувати тільки типи, що знаходяться нижче в ієрархії наслідування, ніж вихідний тип.

```
/* приклад # 5: методи-підставки: CourseHelper.java:
BaseCourseHelper.java: RunnerCourse.java*/
package chapt04;

public class CourseHelper {
      public Course getCourse(){
             System.out.println("Course");
            return new Course();
      }
}
package chapt04;

public class BaseCourseHelper extends CourseHelper {
       public BaseCourse getCourse(){
              System.out.println("BaseCourse");
            return new BaseCourse();
      }
}
package chapt04;

public class RunnerCourse {
      public static void main(String[] args) {
            CourseHelper bch = new BaseCourseHelper();
            Course course = bch.getCourse();
            //BaseCourse course = bch.getCourse();//помилка компіляції
            System.out.println(bch.getCourse().id);
      }
}
```

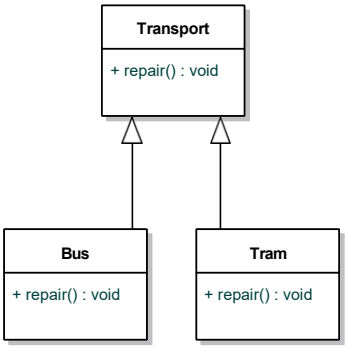
У цій ситуації при компіляції в підкласі BaseCourseHelper створюються два методи. При об-версії методу getCourse() версія методу визначається «раннім зв'язуванням» без використання поліморфізму, але під час виконання викликається метод-підставка. Звернення до поля проводиться за типом посилання, яке повертається методом getCourse(), тобто до поля класу Course.

Поліморфізм та розширюваність

В об'єктно-орієнтованому програмуванні застосування наслідування надає можливість розширення та доповнення програмного забезпечення, що має складну структуру з великою кількістю класів та методів. До завдань базового класу в цьому випадку входить визначення інтерфейсу (як способу взаємодії) для всіх спадкоємців.

У наступному прикладі приведення до базового типу відбувається у виразі:

```
Transport s1 = new Bus();
Transport s2 = new Tram();
```



Мал. 4.2.Приклад реалізації поліморфізму

Базовий клас Transport надає спільний інтерфейс своїх підкласів. Породжені класи **Bus**та Tram перекривають ці визначення для забезпечення унікальної поведінки.

```
/ * Приклад # 5 : поліморфізм: Transport.java: Bus.java: Tram.java:
RepairingCenter.java: Runner.java*/
package chapt04;
import java.util.Random;

class Transport {
      public void repair() {/* порожня реалізація */
      }
}
class Bus extends Transport {
      public void repair() {
            System.out.println("відремонтований АВТОБУС");
      }
}
class Tram extends Transport {
      public void repair() {
            System.out.println("відремонтований ТРАМВАЙ");
      }
}
class RepairingFactory {//шаблон Factory
       public Transport getClassFromFactory(int numMode) {
             switch(new Random().nextInt(numMode)) {
              case 0:
                    return new Bus();
              case 1:
                    return new Tram();
             default:
                 throw new IlegallegalArgumentException();
                  // assert false;
                  // return null;
                  /*
            * if((int)(Math.random() * numMode)==0) return new Bus(); else
            * return new Tram(); як альтернативний і не дуже вдалий
            * варіант. Чому?
                  */
            }

      }
}
public class Runner {
      public static void main(String[] args) {
            RepairingFactory rc = new RepairingFactory();
            Transport[] box = new Transport[15];

      for(int i = 0; i < box.length; i++)
            /* заповнення масиву одиницями перевіреного транспорту */
        box[i] = rc.getClassFromFactory(2);// 2 види транспорту
            for(Transport s : box)
               s.repair();// виклик поліморфного методу
      }
}
```

У процесі виконання програми буде випадковим чином сформовано масив з автобусів і трам-та інформація про їх ремонт буде виведена на консоль.

Клас RepairingFactory містить метод getClassFromFactory(int numMode), який повертає посилання на випадково вибраний об'єкт підкласу класу Transport щоразу, коли він викликається. Приведення до базового типу здійснюється оператором return, який повертає посилання Bus або Tram. Метод main() містить масив посилань Transport, заповнений за допомогою виклику getClassFromFactory(). На цьому етапі відомо, що є кілька посилань на об'єкти базового типу і нічого більше (не більше, ніж знає компілятор). Коли відбувається переміщення цим масивом, метод repair() викликається кожному за випадковим чином обраного об'єкта.

Якщо знадобиться в подальшому додати систему, наприклад, клас TrolleyBus, це вимагатиме лише перевизначення методу repair() і додавання одного рядка в код методу getClassFromFactory(), що робить систему легко расширяемой.

Статичні методи та поліморфізм

Перевизначення статичних методів класу немає практичного сенсу, оскільки звернення до статичного атрибуту чи методу здійснюється у вигляді завдання імені класу, якому вони належать. До статичних методів

принципи «пізнього зв'язування» не застосовуються. При використанні посилання для доступу до статичного члена компілятор при виборі методу або поля враховує тип посилання, а не тип присвоєного їй об'єкта.

```
/* Приклад # 6: поведінка статичного методу при «перевизначенні»: Runner.java */
package chapt04;

class Base {
      public static void assign() {
            System.out.println(
                  "метод assign() з Base");
      }
}
class Sub extends Base {
      public static void assign() {
            System.out.println(
                  "метод assign() із Sub");
      }
}
public class Runner {
      public static void main(String[] args) {
            Base ob1 = new Base();
            Base ob2 = New Sub ();
            Sub ob3 = New Sub();
            ob1.assign();//некоректний виклик статичного
            методу ob2.assign();// слід викликати
            Base.assign(); ob3.assign();
      }
}
```

В результаті виконання цього коду буде виведено:

```
метод assign() з Base
метод assign() з Base
метод assign() з Sub
```

При такому способі ініціалізації об'єктів ob1 і ob2 метод assign() буде викликаний з класу Base. Для об'єкта ob3 буде викликаний власний метод assign(), що з способу оголошення об'єкта. Якщо ж специфікатор static прибрати з оголошення методів, то виклики методів здійснюватимуться у відповідності до принципів поліморфізму.

Статичні методи завжди слід викликати через ім'я класу, в якому вони оголошені, а саме:

```
Base.assign();
Sub.assign();
```

Виклик статичних методів через об'єкт вважається нетиповим і порушує сенс статичного визначення.

Абстракція та абстрактні класи

Безліч предметів реального світу має деякий набір загальних характеристик і правил поведінки. Абстрактне поняття «Геометрична фігура» може містити опис геометричних параметрів та розташування центру ваги в системі координат, а також можливості визначення площі та периметра фігури. Однак у загальному випадку дати конкретну реалізацію наведених характеристик та функціональності неможливо через занадто загальне їх визначення. Для конкретного поняття, наприклад «Квадрат», дати опис лінійних розмірів та визначення площі та периметра не складає труднощів. Абстрагування поняття має надавати абстрактні характеристики предмета реального світу, а не його очікувану реалізацію. Грамотне виділення абстракцій дозволяє структурувати код програмної системи загалом і повторно використовувати абстрактні поняття для конкретних реалізацій щодо нових можливостей абстрактної сутності.

Абстрактні класи оголошуються з ключовим словом abstract і містять оголошення абстрактних методів, які не реалізовані у цих класах, а будуть реалізовані у підкласах. Об'єкти таких класів створити не можна, але можна створити об'єкти підкласів, які реалізують ці методи. При цьому допустимо оголошувати посилання на абстрактний клас, але ініціалізувати його можна лише об'єктом похідного від нього класу. Абстрактні класи можуть містити і повністю реалізовані методи, а також конструктори та поля даних.

За допомогою абстрактного класу оголошується контракт (вимоги до функціональності) для його підкласів. Прикладом може бути вже розглянутий вище абстрактний клас Number та його підкласи Byte, Float та інші. Клас Number оголошує контракт на реалізацію низки методів перетворення даних до значення конкретного базового типу, наприклад floatValue(). Можна припустити, що реалізація методу буде різною для кожного з класів-оболонок. Хоча об'єкт класу Number не можна створити, може отримати чисельне значення будь-якого базового типу. Однак у самого класу немає можливості перетворити це значення до конкретного базового типу.

```
/* приклад # 7: абстрактний клас і метод: AbstractManager.java */
package chapt04;

public abstract class AbstractManager {
      private int id;
      public AbstractManager(int id) {// конструктор
            this. id = id;
```

```
        }
        // абстрактний метод
        public abstract voidassignGroupToCourse(
                    intgroupId, String nameCourse);
}
```

/ приклад # 8: підклас абстрактного класу: CourseManager.java */*

```
packagechapt04;


// assignGroupToCourse() має бути реалізований у підкласі
public classCourseManager extends AbstractManager {
      public voidassignGroupToCourse(
                  intgroupId, String nameCourse) {
            //...
            System.out.println("група" + groupId
                  + "призначена на курс" + nameCourse);
      }
}
```

/ приклад # 9: оголошення об'єктів та виклик методів: Runner.java */*

```
packagechapt04;

public classRunner {
      public static voidmain(String[] args) {
            AbstractManager mng; // можна оголосити
            посилання
          // mng = newAbstractManager();не можна
            створити об'єкт! mng = new
            CourseManager();
            mng.assignGroupToCourse(10, "Алгебра");
      }
}
```

В результаті буде отримано:

```
Група 10 призначена на курс Алгебра
```

Посилання на абстрактний суперклас mng ініціалізується об'єктом підкласу, в якому реалізовано всі абстрактні методи суперкласу. За допомогою цього посилання можуть викликатися реалізовані методи абстрактного класу, якщо вони не перевизначені у підкласі.

Клас Object

На вершині ієрархії класів знаходиться клас Object, який суперкласом для всіх класів. Посилальна змінна типу Object може вказувати на об'єкт будь-якого іншого класу, будь-який масив, оскільки масиви реалізуються як класи. У класі Object визначено набір методів, що успадковується всіма класами:

protected Object clone() – створює та повертає копію об'єкта, що викликає;

boolean equals(Object ob) – призначений для перевизначення у підкласах із виконанням загальних угод про порівняння вмісту двох об'єктів;

Class<? extends Object> getClass() - Повертає об'єкттипу Class;

protected void finalize() - Викликається перед знищенням об'єкта автоматичним збирачем сміття (garbage collection);

int hashCode() - Повертає хеш-код об'єкта;

String toString() - Повертає подання об'єкта у вигляді рядка.

Методи notify(), notifyAll() та wait() будуть розглянуті у розділі «Потоки виконання».

Якщо при створенні класу передбачається перевірка логічної еквівалентності об'єктів, яка не виконана у суперкласі, слід перевизначити два методи: equals(Object ob) та hashCode(). Крім того, перевизначення цих методів необхідне, якщо логіка програми передбачає використання елементів у колекціях. Метод equals() при порівнянні двох об'єктів повертає істину, якщо вміст об'єктів еквівалентний, і брехня – інакше. При перевизначенні методу equals() повинні виконуватись угоди, передбачені специфікацією мови Java, а саме:

- рефлексивність - об'єкт дорівнює самому собі;
- симетричність – якщо**x.equals(y)** повертає значення**true**, то й**y.equals(x)** завжди повертає значення**true**;
- транзитивність – якщо метод**equals()** повертає значення**true**при порівнянні об'єктів**x**і**y**, а також**y**і**z**, то й при порівнянні**x**і**z**буде повернено значення**true**;
- несуперечність - при багаторазовому виклику методу для двох об'єктів, що не зазнали зміни за цей час, повертається значення завжди повинно бути однаковим;
- ненульове посилання у порівнянні з літералом**null**завжди повертає значення**false**.

При створенні інформаційних класів також рекомендується перевизначати методи hashCode() та

`toString()`, щоб адаптувати їх дії для створюваного типу.

Метод hashCode() перевизначений, як правило, у кожному класі і повертає число, що є унікальним ідентифікатором об'єкта, що в більшості випадків залежить тільки від значення об'єкта. Його слід перевизначати завжди, коли перевизначено метод equals(). Метод hashCode() повертає хеш-код об'єкта, обчислення якого керується такими угодами:

- під час роботи програми значення хеш-коду об'єкта не змінюється, якщо об'єкт не було змінено;
- всі однакові за змістом об'єкти одного типу повинні мати однакові хеш-коди;
- різні за змістом об'єкти одного типу можуть мати різні хеш-коди.

Один із способів створення правильного методу hashCode(), що гарантує виконання угод, наведено нижче у прикладі # 10.

Метод toString() слід перевизначати таким чином, щоб крім стандартної інформації про пакет (опціонально), в якому знаходиться клас, і самого імені класу (опціонально), він повертав значення полів об'єкта, що викликав цей метод (тобто всю корисну інформацію об'єкта), замість хеш-коду, як це робиться у класі Object. Метод toString() класу Object повертає рядок з описом об'єкта у вигляді:

```
getClass().getName() + '@' +
Integer.toHexString(hashCode())
```

Метод викликається автоматично, коли об'єкт виводиться методами println(), print() та деякими іншими.

```
/* приклад # 10: перевизначення методів equals(), hashCode, toString():
Student.java */
package chapt04;
public class Student
      {private int id;
      private String name;
      private int age;

      public Student(int id, String name, int age) {
            це.id = id;
            this.name = name;
            this.age = age;
      }
      public int getId() {
            return id;
      }
      public String getName() {
            return name;
      }
      public int getAge() {
            return age;
      }
      public boolean equals(Object obj) {
            if(this == obj)
                  return true;
            if (obj == null)
                  return false;
            if(obj  instanceof  Student)  {  //
                  Warning   Student   temp   =
                  (Student) obj; return this.id
                  == temp.id &&
                        name.equals(temp.name) &&
                        this.age == temp.age;
            } else
                  return false;
      }
      public int hashCode() {
            return (int) (31 * id + age
            + ((name == null) ? 0 : name.hashCode()));
      }
      public String toString() {
            return getClass().getName() + "@name" + name
                        + "id:" + id + "age:" + age;
      }
}
```

Вираз 31 * id + age гарантує різні результати обчислень при зміні місцями значень полів, а саме якщо id=1 та age=2, то в результаті буде отримано 33, якщо значення поміняти місцями, то 63. Такий підхід застосовується за наявності у класів полів базових типів.

Метод equals() перевизначається для класу Student таким чином, щоб переконатися в тому, що отриманий об'єкт є об'єктом типу Student або одним з його спадкоємців, а також порівняти вміст полів id, name і age відповідно у метод об'єкта і об'єкта, що викликає, що передається як параметр.

```
/*приклад # 11: клас студента факультету: SubStudent.java */
package chapt04;

public class SubStudent extends Student {
     private int idFaculty;
     public SubStudent (int id, String n, int a, int idf) {
            super(id, n, a);
            це.idFaculty = idf;
       }
}
```

```
/*Приклад # 12 : демонстрація роботи методу equals() при успадкування:
 StudentEq.java */
package chapt04;

public class StudentEq {
     public static void main(String[] args) {
            Student p1 = new Student(71, "Петрів", 19);
            Student p2 = new Student(71, "Петрів", 19);
            SubStudent p3 =
                 new SubStudent(71, "Петрів", 19, 5);
            System.out.println(p1.equals(p2));
            System.out.println(p1.equals(p3));
            System.out.println(p3.equals(p1));
     }
}
```

В результаті виконання цього коду буде виведено таке:

```
true
true
true
```

Перевизначений таким чином метод equals() дозволяє порівнювати об'єкти суперкласу з об'єктами підкласів, але тільки за полями, які є спільними. При наслідуванні з додаванням нових полів у підклас використання методу порівняння із суперкласу призводить до некоректних результатів.

Цю проблему можна легко вирішити, якщо замість рядка з позначкою //warning метод equals() класу Student підставити безпосередню перевірку на відповідність типів порівнюваних об'єктів з використанням об'єкта класу Class у вигляді:

```
if (getClass() == obj.getClass())
```

то в результаті буде виведено:

```
true
false
false
```

У той же час, така реалізація методу equals() повертатиме істину при порівнянні об'єктів класу SubStudent з однаковими значеннями полів, успадкованих від класу Student.

Клонування об'єктів

Об'єкти в методи передаються за посиланням, внаслідок чого метод передається посилання на об'єкт, що знаходиться поза методом. Тому якщо в методі змінити значення поля об'єкта, то ця зміна торкнеться вихідного об'єкта. Щоб уникнути такої ситуації для захисту зовнішнього об'єкта, слід створити клон (копію) об'єкта в методі. Клас Object містить protected-метод clone(), який здійснює побітове копіювання об'єкта похідного класу. Однак спочатку необхідно перевизначити метод clone() як public для забезпечення можливості дзвінка з іншого пакета. У перевизначеному методі слід викликати базову версію методу super.clone(), яка й виконує власне клонування. Щоб остаточно зробити об'єкт, що клонується, клас повинен реалізувати інтерфейс Cloneable. Інтерфейс Cloneable не містить методів відноситься до позначених (tagged) інтерфейсів, а його реалізація гарантує, що метод clone() класу Object поверне точну копію об'єкта, що викликав його, з відтворенням значень всіх його полів. В іншому випадку метод генерує виключення CloneNotSupportedException. Слід зазначити, що з використанням цього механізму об'єкт створюється без виклику конструктора. У мові C++ аналогічний механізм реалізований за допомогою конструктора копіювання.

```
/* приклад # 13: клас, що підтримує клонування: Student.java */
package chapt04;

public class Student implements Cloneable {/*включення
                                              інтерфейсу */
      private int id = 71;
      public int getId() {
            return id;
      }
      public void setId(int value) {
```

```
            id = value;
      }
      publicObject clone() {//перевизначення методу
            try{
                  return super.clone();//виклик базового методу
            } catch (CloneNotSupportedException e) {
                  throw newAssertionError("неможливо!");
            }
      }
}
```

/ приклад # 14: безпечна передача за посиланням: DemoSimpleClone.java */*

```
packagechapt04;
public class DemoSimpleClone {
      private static voidchangeId(Student p) {
            p = (Student)
            p.clone();//клонування
            p.setId(1000);
            System.out.println("->id = " + p.getId());
            }
      public static voidmain(String[] args) {
            Student ob = new Student();
            System.out.println("id = " + ob.getId());
            changeId(ob);
            System.out.println("id = " + ob.getId());
      }
}
```

В результаті буде виведено:

```
id = 71
-> id = 1000
id = 71
```

Якщо закоментувати виклик методу clone(), то буде виведено наступне:

```
id = 71
-> id = 1000
id = 1000
```

Таке рішення ефективно тільки у випадку, якщо поля об'єкта, що клонується, являють собою значення базових типів та їх оболонок або незмінних (immutable) об'єктних типів. Якщо ж поле типу, що клонується, є змінним об'єктним типом, то для коректного клонування потрібен інший підхід. Причина полягає в тому, що при створенні копії поля оригінал і копія є посиланням на той самий об'єкт. У цій ситуації слід також клонувати об'єкт поля класу.

/ приклад # 15: глибоке клонування: Student.java */*

```
packagechapt04;
importjava.util.ArrayList;

public class Student implements Cloneable {
      private intid = 71;
      privateArrayList <Mark> lm = новий ArrayList<Mark>();

       public intgetId() {
            returnid;
      }
      public voidsetId(int id) {
           це.id = id;
      }
      publicArrayList<Mark> getMark() {
           returnlm;
      }
      public voidsetMark(ArrayList<Mark> lm) {
           це.lm = lm;
       }
      publicObject clone() {
            try{
                  Student copy = (Student) super.clone ();
                  copy.lm = (ArrayList<Mark>)lm.clone();
                  return copy;
            } catch (CloneNotSupportedException e) {
                  throw newAssertionError(
```

```
                                    "відсутня Cloneable!");
            }
        }
}
```

Таке клонування можливе лише у випадку, якщо тип атрибута класу також реалізує інтерфейс Cloneable та перевизначає метод clone(). В іншому випадку, виклик методу неможливий, оскільки він просто недоступний. Отже, якщо клас має суперклас, то для реалізації механізму клонування поточного класу потрібна наявність коректної реалізації такого механізму в суперкласі. При цьому слід відмовитися від використання об'яв final для полів об'єктних типів через неможливість зміни їх значень при реалізації клонування.

"Складання сміття" та звільнення ресурсів

Так як об'єкти створюються динамічно за допомогою операції new, а знищуються автоматично, бажано знати механізм ліквідації об'єктів і спосіб звільнення пам'яті. Автоматичне звільнення пам'яті, яку об'єкт займає, виконується за допомогою механізму "складання сміття". Коли жодних посилань на об'єкт не існує, тобто всі посилання на нього вийшли з області видимості програми, передбачається, що об'єкт більше не потрібен, і пам'ять, зайнята об'єктом, може бути звільнена. "Складання сміття" відбувається нерегулярно під час виконання програми. Форсувати "складання сміття" неможливо, можна лише "рекомендувати" його виконати викликом методу System.gc() або Runtime.getRuntime().gc(), але віртуальна машина виконає очищення пам'яті тоді, коли сама вважає це зручним. Виклик методу System.runFinalization() призведе до запуску методу finalize() для об'єктів, що втратили всі посилання.

Іноді об'єкту потрібно виконувати деякі дії перед звільненням пам'яті. Наприклад, звільнити зовнішні ресурси. Для обробки таких ситуацій можуть застосовуватися два способи: конструкція try-finally і механізм finalization. Конструкція try-finally є кращою, абсолютно надійною і буде розглянута у дев'ятому розділі. Запуск механізму finalization визначається алгоритмом складання сміття і до його безпосереднього виконання може пройти скільки завгодно багато часу. Через все це поведінка методу finalize() може вплинути на коректну роботу програми, особливо за зміни JVM. Якщо є можливість звільнити ресурси чи виконати інші подібні дії без залучення цього механізму, краще без нього обійтися. Віртуальна машина викликає цей метод завжди, коли вона збирається знищити об'єкт цього класу. Всередині методу finalize(), який викликається безпосередньо перед звільненням пам'яті, слід визначити дії, які мають бути виконані до знищення об'єкта.

Метод finalize() має таку сигнатуру:

```
protected voidfinalize(){
    // код завершення
}
```

Ключове слово protected забороняє доступ до finalize() коду, визначеного за межами цього класу. Метод finalize() викликається лише перед «складанням сміття», а не тоді, коли об'єкт виходить з області видимості, тобто заздалегідь неможливо визначити, коли finalize() буде виконано, і недоступний об'єкт може займати пам'ять досить довго. У принципі, цей метод може бути взагалі не виконаний! Не можна в додатку довіряти такому методу критичні за часом дії зі звільнення ресурсів.

```
/* приклад # 16: клас Manager з підтримкою finalization: Manager.java */
packagechapt04;

classManager {
      private intid;

      publicManager(int value) {
            id = value;
      }
      protected voidfinalize() throws Throwable {
          try{
          //звільнення ресурсів
       System.out.println("об'єкт буде видалений, id=" + id);
          } finally {
              super.finalize();
          }
    }
}
packagechapt04;

public classFinalizeDemo {
     public static voidmain(String[] args) {
          Manager d1 = new Manager(1);
          d1 = null;
          Manager d2 = New Manager(2);
          Object d3 = d2;                    //1
          // Object d3 = new Manager (3);     //2
          d2 = d1;
          System.gc();// прохання виконати "складання сміття"
```

```
        }
}
```

В результаті виконання цього коду перед викликом методу System.gc() без посилання залишиться лише один об'єкт.

```
об'єкт буде видалено, id=1
```

Якщо закоментувати рядок 1 і зняти коментар з рядка 2, перед виконанням gc() посилання втратять вже два об'єкти.

```
об'єкт буде видалено, id=1
об'єкт буде видалено, id=2
```

Якщо не викликати метод finalize() суперкласу, то він не буде викликаний автоматично. Ще одна небезпека полягає в тому, що якщо при виконанні даного методу виникне виняткова ситуація, то вона буде проігнорована і додаток продовжуватиме виконуватися, що також становить небезпеку для його коректної роботи.