МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чернігівський національний технологічний університет

ОСНОВИ ПРОГРАМУВАННЯ НА JAVA

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторного практикуму та самостійної роботи з дисципліни
«Об'єктно орієнтоване програмування»
для студентів спеціальності
6.050102 – "Комп'ютерна інженерія"

ЗАТВЕРДЖЕНО
на засіданні кафедри
інформаційних та комп'ютерних систем
протокол № 1 від 29.08.18

Чернігів ЧНТУ 2018

Основи програмування на Java. Методичні вказівки до лабораторного практикуму та самостійної роботи з дисципліни «Об’єктно орієнтоване програмування» для студентів спеціальності 128 – “Комп’ютерна інженерія”. /Укл.: Бивойно П. Г., Бивойно Т. П. – Чернігів: ЧНТУ, 2018. – 108 с.

Укладачі: Бивойно Павло Георгійович, канд. техн. наук, доцент;
Бивойно Тарас Павлович, старший викладач.

Відповідальний за випуск: С. В. Зайцев, доктор. техн. наук, зав. кафедри інформаційних комп'ютерних систем Чернігівського національного технологічного університету

Рецензент: В. А. Бичко, канд. ф-м. наук, доцент кафедри інформаційних та комп'ютерних систем Чернігівського національного технологічного університету

ЗМІСТ

Вступ

Лабораторні роботи є сполучною ланкою між лекційними заняттями та самостійною роботою студентів, в межах якої виконується розрахунково-графічна робота. В процесі виконання лабораторних робіт експериментально перевіряються ключові питання курсу, набуваються практичні навички побудови та налагодження програм, перевіряється ступінь засвоєння основних положень предмета. Під час лабораторних занять студенти знайомляться з типовими рішеннями деяких задач з використанням об’єктно орієнтованого підходу, що допомагає в отриманні досвіду об'єктно-орієнтованого програмування.

Лабораторні роботи виконуються на персональних комп'ютерах в системі програмування Eclipse. Передбачається, що студенти знайомі з основами програмування на мові С.

Студент зобов'язаний до лабораторного заняття ознайомитися з методичними вказівки до лабораторної роботи і відповідними теоретичними положеннями з конспекту лекцій. Під час лабораторного заняття студент починає виконувати лабораторну роботу, консультуючись у викладача з питань, що виникли. До більшості робіт студенту пропонується завдання до самостійної роботи, яку студент виконує вдома. Обсяг виконаної роботи може бути різним, залежно від того, на яку оцінку претендує студент. Коли робота закінчена, студент повинен захистити її. Захист полягає у відповідях на питання по темі лабораторної роботи і внесенні деяких змін у розроблений проект, в присутності викладача.

По кожній роботі студент повинен оформити звіт. У звіті не наводяться теоретичні положення. Мають бути тільки результати роботи, що відповідають підрозділу «Порядок виконання роботи». Звіти оформляються за допомогою текстового редактора Word на папері формату А4, у відповідності з вимогами стандартів на оформлення технічної документації. Звіт по роботі є розділом підсумкового документа. В кінці семестру звіти зшиваються в єдиний підсумковий документ з титульним листом, підписуються у керівника, після чого студент отримує допуск до іспиту.

За лабораторну роботу студент може отримати до 100 балів, з урахуванням своєчасності та якості виконання всіх складових роботи. Складовими є: звіт, результат виконання завдання для самостійної роботи і відповіді на контрольні питання. Оцінки, отримані за лабораторні роботи, враховуються при виставленні підсумкової оцінки. Для отримання допуску до іспиту всі роботи повинні бути виконані і кожна з них оцінена не менше ніж в 60 балів.

1 ЛАБОРАТОРНА РОБОТА № 1. ЗНАЙОМСТВО З СЕРЕДОВИЩЕМ ECLIPSE ТА ПРОГРАМОЮ НА МОВІ “JAVA”

Мета роботи:

- Отримати первинні навички роботи з середовищем Eclipse.
- Отримати первинні уявлення про клас та об’єкт.
- Ознайомитися зі структурою runtime бібліотеки Java.
- Ознайомитися з деякими класами мови Java.
- Отримати уявлення про програму на мові Java.
- Створити простіші програми, використовуючи класи Java.

1.1 ПОРЯДОК ВИКОНАННЯ РОБОТИ

1.1.1 *Створення проекту у інтегрованому середовищі розробки «Eclipse»*

При первісному запуску Eclipse на екрані з'являється вікно привітання Wellcome, але для роботи нам потрібно робоче вікно середовища розробки (перспектива Java). Це вікно можна відкрити за допомогою функції головного меню Window → NewWindow або вибору WorkBench на вікні-заставці.

Eclipse може після запуску також запропонувати відкрити або створити робочу область (папку, де будуть зберігатися ваші пакети та файли).

У цьому випадку створіть або відкрийте свою робочу область на диску, рисунок 1.1.

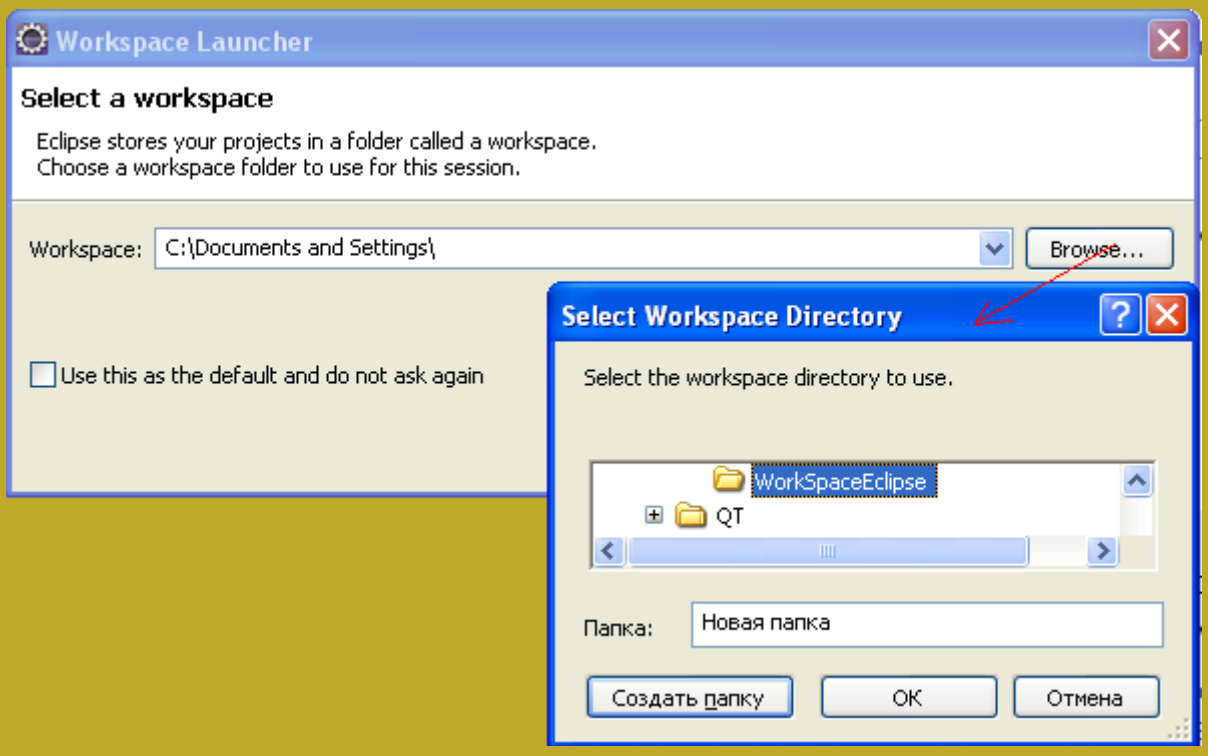

Рисунок 1.1 – Створення робочої області для проектів

Швидко створити проект можна за допомогою майстра створення проектів Java. Для цього необхідно виконати наступні дії:

– у головному меню виберіть File → New → JavaProject. Відкриється майстер створення проекту, рисунок 1.2;

– введіть ім’я проекту (проект – це папка у робочій області, а тому вимоги до імен проектів ті самі що й до назв папок);

– решта налаштувань має бути такою, як на рисунку 1.2;

– після цього можна натискати кнопку Finish;

– оскільки ми створюємо проект Java, то з ним має бути пов'язана Java-перспектива. Якщо Eclipse пропонує це зробити, то відмовлятися не слід.

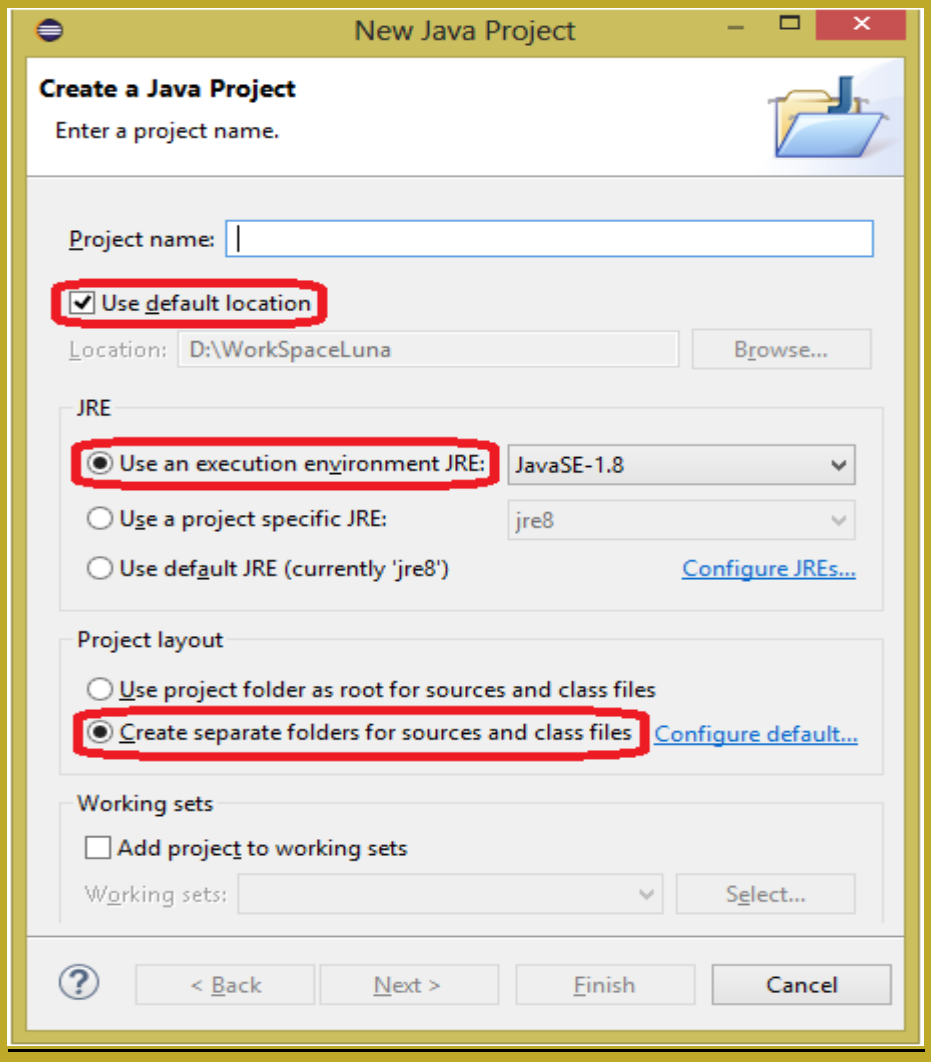

Рисунок 1.2 – Вікно майстра створення проекту Java

Проект створено. У закладці Package Explorer має з'явитися ім'я проекту (наприклад myFirstProject), рисунок 1.3.

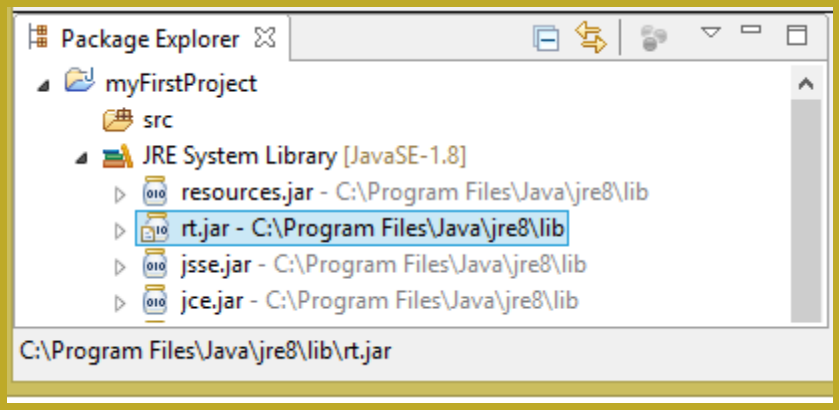

Рисунок 1.3 – Закладка «Package Explorer»

Зверніть увагу, до проекту підключена системна бібліотека JRE System Library (Java Runtime Environment), яка містить java-архіви з байт-кодами класів Java. Основні класи Java знаходяться в архіві rt.jar. Тут можна переглянути вихідні тексти класів Java. Але це можливо тільки у випадку, якщо вказано шлях до файлу src.zip, що зазвичай знаходиться у папці jdk. Тому jdk слід скачувати теж.

1.1.2 *Створення Java класу*

Після створення нового проекту слід створити пакет, бо кожний клас Java має бути у пакеті. Це можна зробити за допомогою функції головного меню File → New → Package, яка відкриває відповідний діалог, рисунок 1.4.

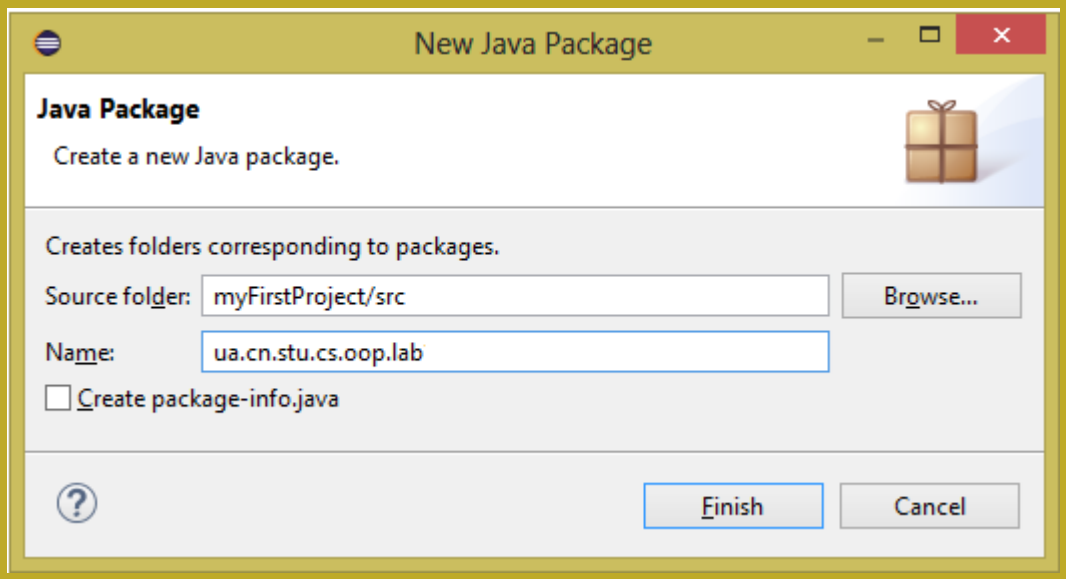

Рисунок 1.4 – Діалог створення пакету

Ім’я пакету може бути будь яким, але для унікальності класів рекомендують створювати імена пакетів на зразок internet-адрес, тільки у зворотному напрямку, рисунок 1.4. Але враховуючи, що наші класи не будуть виходити за межі лабораторного практикуму, можна для пакету вибирати і прості імена. Слід додержуватися також правила, що імена пакетів починають з маленької букви.

Створивши пакет, переходимо до створення класу. Його можна створити за допомогою діалогу, що з'являється після вибору функції головного меню File → New → Class, рисунок 1.5.

У діалозі перш за все треба визначитися з назвою класу. Назва класу може бути будь якою, але бажано, щоб назва була пов'язана з призначенням класу. Надалі це спрощує роботу з ним. Слід також дотримуватися правила, що назви класів починаються з великої літери.

Далі, в діалозі можна визначитися з суперкласом та інтерфейсами, але нам зараз це не потрібно.

Але доцільно замовити створення шаблону абстрактного статичного методу main().

Можна також замовити генерацію заготовок для коментарів, якщо Ви будете наповнювати ці заготовки.

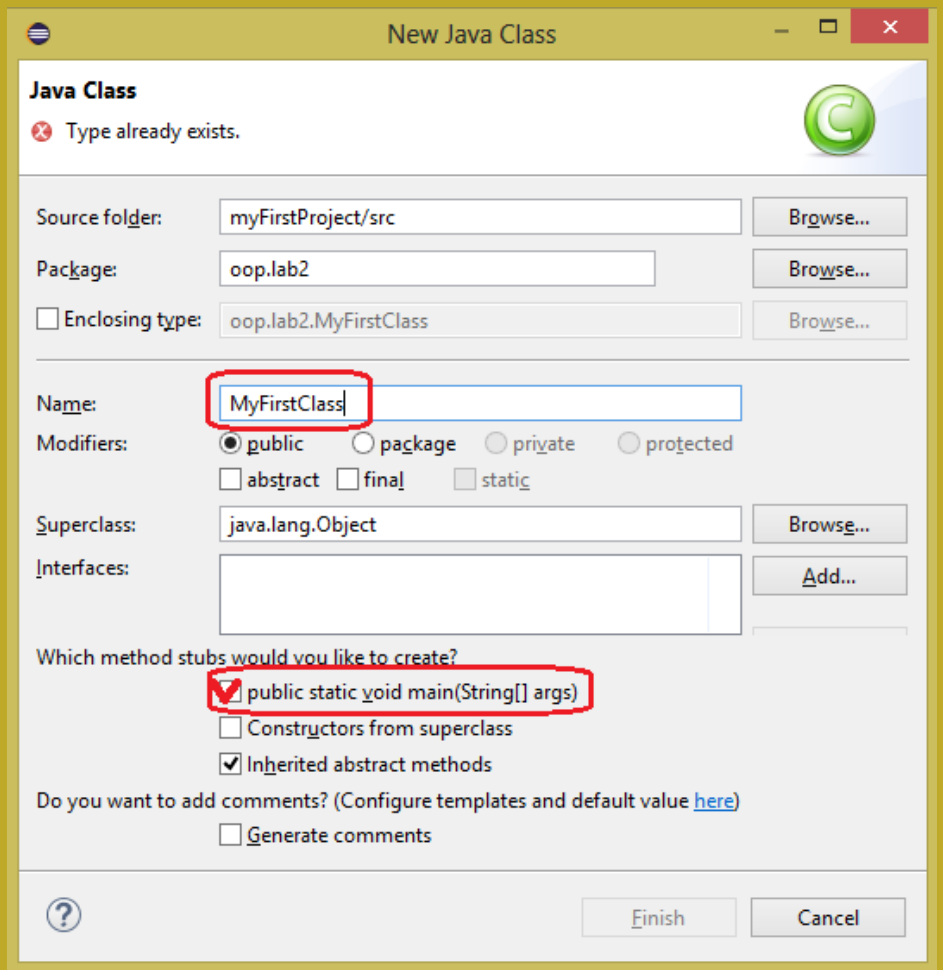

Рисунок 1.5 – Діалог створення класу

Після закриття діалогу з'являється закладка з вікном редактора коду Java для створеного класу, рисунок 1.6.

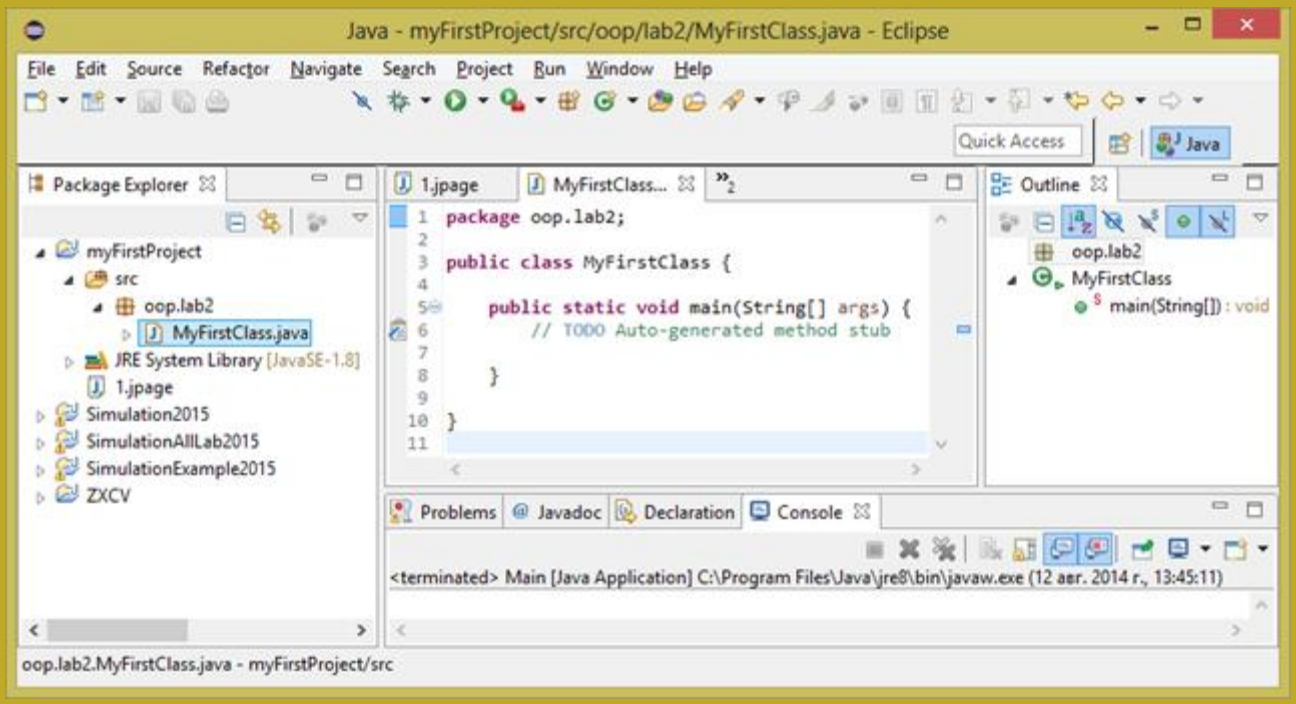

Рисунок 1.6 – Вікно Eclipse із закладкою редактора коду

Окрім редактора коду, ліворуч у вікні Eclipse ми бачимо панель з браузером пакетів (Package Explorer). Праворуч розташована дуже корисна панель Outline, яка автоматично синхронізована з відкритим класом і забезпечує зручну навігацію по елементам класу. Звикайте користуватися цією панеллю!

Нижня частина вікна Eclipse містить панель консолі (Console). На даній лабораторній роботі ми будемо спілкуватися з програмою через це вікно.

Текст класу, який було створено автоматично містить інформацію про пакет. Далі йде заголовок класу і його тіло, обмежене фігурними дужками.

У тілі класу знаходиться заготовка статичного методу main, з тілом у фігурних дужках. Саме тіло містить тільки коментар, який починається словом TODO. Але це не звичайний коментар. Ткий коментар нагадує програмісту, що тут потрібно щось зробити. Для кожного такого коментарю на правій крайці вікна створюється маленький блакитний прямокутник, який нагадує про незавершену роботу над текстом.

У нашому класі цей коментар можна видалити і замість нього писати код, який ми хочемо виконати.

1.1.3 *Створення і запуск найпростішої програми*

Напишіть замість коментаря TODO у тілі методу main такий код:

```
System.out.println("Hello, Word!");
System.out.println("Програму склав тут має бути ваше прізвище ");
```

Після цього, перш за все слід спробувати зрозуміти, що тут написано.

System – це клас з пакету java.lang, що доступний у будь якій програмі на Java. У класі System зберігається об’єкт out з публічним доступом. Цей об’єкт є

об’єктом класу PrintStream. У класі PrintStream є метод println(String), що приймає в якості параметру рядок символів.

Тому у відповідь на питання – «Що тут написано?» можна відповісти так: – «Об’єкту out, що належіть класу System, надсилається повідомлення println з параметром "Hello, Word!"».

Зверніть увагу, після імені об’єкту нема дужок, а метод завжди має дужки, навіть якщо нема параметрів.

Тепер, за допомогою контекстного меню виберіть функцію RunAs → JavaApplication.

Програма буде виконана і на консолі має з’явитися повідомлення Hello, Word!. А у натупному рядку буде повідомлено, хто її склав.

Результат у вигляді копії екрану з програмою та результатом на консолі слід зафіксувати у звіті.

1.1.4 *Дослідження класів System та PrintStream*

Відкрийте на закладці Package Explorer бібліотеку JRE System Library, а у ній файл rt.jar. Далі знайдіть пакет java.lang і в ньому клас System. Можна також натиснути клавішу Ctrl та навести вказівник миші на назву потрібного класу і після появи підкреслення клацнути лівою клавішею.

Далі можна знайомитися з класом.

На рисунку 1.7 показано вікно з текстом класу. На панелі ліворуч частина переліку класів rt бібліотеки. Праворуч панель Outline, через яку зручно переглядати перелік складових класу. На центральній панелі розміщується частина тексту класу відповідно до вибору на панелі Outline. Майте на увазі, коментар до вибраного елементу знаходиться ПЕРЕД текстом цього елементу. Коментарі зручніше читати через вікна коментарів, що з’являються після наведення курсору на назву елементу.

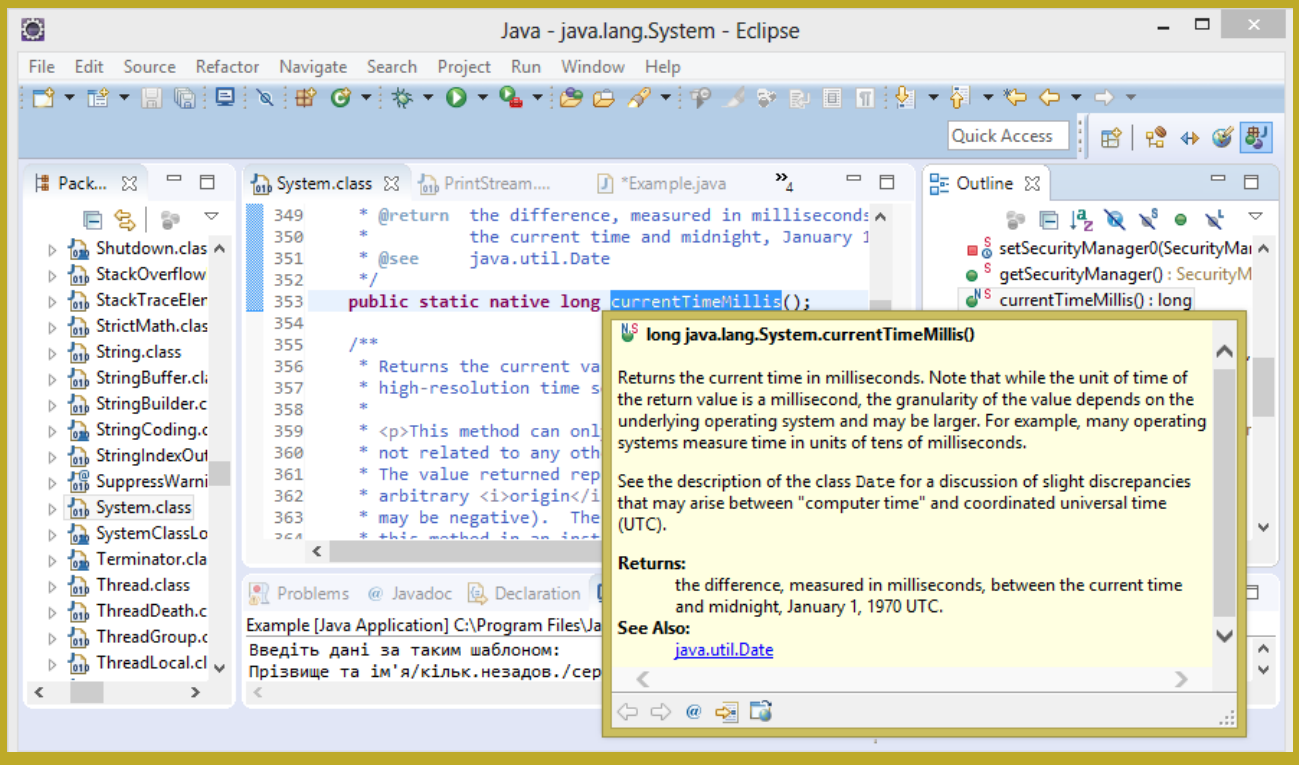

Рисунок 1.7 – Вікно Eclipse із текстом класу і коментарем

Переконайтеся, що об’єкт out належить класу PrintStream.

Знайдіть у класі System декілька методів, що можуть для Вас бути зрозумілими.

Перейдіть до класу PrintStream. З’ясуйте, чи можна виводити дані на консоль так як у С, за допомогою printf з рядком формату?

Результати виконання цього пункту мають бути зафіксовані у звіті.

1.1.5 *Дослідження класів StringBuilder та Scanner*

Об’єкти класу String представляють собою незмінні рядки символів.

Клас StringBuilder використовується для формування динамічного рядка символів.

Об’єкти класу Scanner використовуються для введення інформації з консолі.

Створімо у нашому проекті новий клас без методу main. Далі в тілі класу напишіть «main» та натисніть клавіші Ctrl+Пробіл. Як результат має з’явитися вікно вибору шаблонів автогенерації коду. Виберіть метод main і отримаєте його заготовку у тілі класу.

Далі наберіть код, що наведено нижче. Набираючи імена класів, друкуйте тільки декілька перших літер, потім натискайте Ctrl+Пробіл. Це вас убереже від помилок у назв класів та дозволить Eclipse автоматично добавляти інструкції імпорту.

Друкуючи текст зверніть увагу на те, що об’єкти sb (стрінгбуфер) та sc (сканер) створюються за допомогою інструкції new, що застосовується до конструктора класу. Конструктор завжди має таку саму назву як клас і після назви йде пара круглих дужок, між якими можуть бути вказані параметри.

Якщо після new ввести пробіл а потім Ctrl+Пробіл, можна викликати автогенерацію потрібного конструктора.

```
StringBuilder sb = new StringBuilder("Введіть дані за таким шаблоном:\n");
sb.append("Прізвище та ім'я/кільк.незадов./сердній бал/\n");
sb.append("Наприклад:\nКіт Базіліо/2/2,4/");
// Виводимо вміст стрінгбуферу
System.out.println(sb);
// Далі програма очикує на введення даних з консолі
Scanner sc = new Scanner(System.in);
sc.useDelimiter("/");
String name = sc.next();
int n = sc.nextInt();
float b = sc.nextFloat();
sc.close();
// Виведення отриманих даних
System.out.println(name+ ": двійок " + n + "; ср.бал=" + b);
```

Зверніть увагу на останній рядок. Там у дужках створюється об’єкт класу String із даних різних типів з використанням операцій +. Це така невластива для ООП особливість класу String.

Виконайте програму. Введіть своє ім’я і свої результати за семестр. Зафіксуйте результат у звіті у вигляді копії екрану.

Перегляньте класи String, StringBuilder, Scanner. Знайдіть у цих класах додатково по декілька корисних методів і наведіть їх у звіті.

1.1.6 *Дослідження класів Math та Arrays*

Клас Math містить статичні методи, що реалізують найбільш поширені математичні функції, а також деякі константи, наприклад π. Викликаються статичні методи безпосередньо через звернення до класу.

Клас Arrays містить статичні методи, що спрощують роботу з масивами.

Створімо у нашому проекті новий клас з методом main. Далі в тілі методу «main» введіть наведений нижче код. Не забувайте про автогенерацію коду. Наприклад, якщо після набору syso натиснути клавіші Ctrl+Пробіл, то отримаємо System.out.println(). Якщо одразу після for натиснути Ctrl+Пробіл, отримаємо для вибору перелік варіантів циклу for.

Робота з кодом спрощується, коли код відформатований. В Eclipse це завдання вирішується автоматично. Для форматування слід викликати функцію будь якого меню Source → Format.

```
// Вводимо довжину масиву
Scanner scan = new Scanner(System.in);
System.out.println("Введіть довжину масиву");
int size = scan.nextInt();
scan.close();

// Формуємо  масив
double[] arr = new double[size];

// Заносимо випадкові числа
for (int i = 0; i < arr.length; i++) {
      arr[i] = Math.random()*100;
}

// Сортуємо масив
Arrays.sort(arr);

// Виводимо масив за допомогою foreach
System.out.println("Ось числа масиву:");
for (double d : arr) {
      System.out.printf("%6.2f", d);
}
System.out.println();
```

Виконайте створену програму для масиву довжиною, що дорівнює значенню останньої цифри залікової книжки, якщо вона більше 5, інакше додайте ще 10. Копію екрану з програмою та результатом наведіть у звіті.

Познайомтесь з можливостями класів Math та Arrays. Дайте у звіті короткий огляд методів цих класів.

1.2 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Створити програму для обчислення стипендії студенту на підставі інформації, введеної з консолі. Вводити слід ім’я студента, статус студента(бюджет/контракт) та масив результатів сесії у балах ECTS, які

потрібно перевести у національну систему (2,3,4,5).

1.3 ВИМОГИ ДО ЗВІТУ

– Назва роботи.
– Мета роботи.
– Результати виконання роботи за пунктами з копіями екранів, що містять код та результати виконання, а також результати самостійного дослідження класів у вигляді короткого огляду по кожному пункту роботи
– Текст програми, що було створено самостійно з результатами.
– Висновки.

1.4 КОНТРОЛЬНІ ПИТАННЯ

1. Як спілкуються між собою об’єкти?
2. Особливості статичних методів.
3. Як створити об’єкт?
4. Що являє собою програми на Java.
5. Структура rt бібліотеки Java.
6. Короткий огляд одного з класів, розглянутих під час роботи.
7. Приклади автогенерації коду.
8. Форматування коду.
9. Дослідити клас за вказівкою викладача, (наприклад, Stack, Vector, LocalDateTime, Random, BigDecimal …) та написати простий приклад використання класу.

2 ЛАБОРАТОРНА РОБОТА № 2. СТВОРЕННЯ КЛАСІВ JAVA

Мета роботи:

– Докладно познайомитися із структурою класу Java.
– Продовжити знайомство із засобами автогенерації коду класу.
– Створити програму з використанням класів Java.

2.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Китайське деревообробне підприємство виготовляє брус з різних сортів деревини і відправляє цю продукцію по річці Амур баржами з Сибіру до Китаю.

Власники підприємства пропонують нам налагодити облік продукції, а також контроль завантаження барж за вагою, щоб уникнути перевантаження барж, враховуючи, що різні види деревини мають суттєво різну питому вагу.

Після аналізу завдання було виділено такі сутності:

– деревина, що характеризується назвою, питомою вагу та унікальним числовим ідентифікатором;

– довідник, що містить інформацію про різні види деревини;

– брус, що характеризується трьома геометричними розмірами та матеріалом, з якого виготовлений;

– сховище даних про оброблені бруси.

Потрібно створити програму, що забезпечить накопичення даних про бруси, формування довідника деревини та підрахунок сумарної накопиченої ваги накопичених брусів.

У подальшому можуть знадобитися і інші операції.

2.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Створіть новий проект для лабораторної роботи та у ньому три пакети – lab2.model, lab2.store, lab2.test.

У пакеті model будемо зберігати класи та інтерфейси для сутностей проекту. Поки що такими сутностями є деревина та брус.

У пакеті store будемо розміщувати класи, що будуть відповідати за збереження інформації.

Пакет test буде містити класи, що забезпечують спілкування з користувачем та тестування.

2.2.1 *Створення класу lab2.model.Wood*

Цей клас реалізує сутність «деревина».

Створіть у пакеті lab2.model новий клас Wood. Необхідності у методі main для класу нема.

Визначимо приватні поля для його характеристик:

```
public class Wood {

	private int id;
	private String name;
	private float density;
}
```

Далі, за допомогою функції меню Source → Generate constructor using fields створіть конструктор класу, використовуючи усі три параметри.

```
public Wood(int id, String name, float density) {
	this.id = id;
	this.name = name;
	this.density = density;
}
```

Префікс **this.** перед іменами полів у конструкторі вказує на те, що поле належить об'єктові даного класу, який має умовну назву this. Це дозволяє розрізняти поля і параметри конструктора, що мають однакові назви.

Після цього, за допомогою функції меню Source → Generate getter and setter створімо публічні методи get для полів класу. Методи set створювати не будемо. Це унеможливіть зміну параметрів вже створеного об'єкту.

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

public String getName() {
	return name;
}

public float getDensity() {
	return density;
}
```

Нарешті створімо метод toString() для об'єктів класу. Цей метод використовується для виведення зрозумілої інформації про об'єкт. Його можна також згенерувати автоматично шляхом виклику функції Source → Generate toString. За бажанням його можна потім відредагувати.

```
@Override
public String toString() {
	return "Wood [id=" + id + ", name=" + name +
			", density=" + density + "]";
}
```

2.2.2 *Створення класу lab2.store.WoodDirectory*

Цей клас визначає властивості та методи каталогу деревини.

Створюємо у пакеті lab2.store клас WoodDirectory.

Визначимо у класі масив об'єктів типу Wood та лічильник кількості об'єктів у каталозі.

```
private Wood[] arr = new Wood[3];
private int count = 0;
```

Конструктор у класі створювати не будемо, щоб показати, що клас може бути і без конструктора. Натомість, додаймо до тіла класу блок ініціалізації, у якому одразу проініціалізуємо масив, що спростить тестування проекту.

```
{
    arr[0] = new Wood(1, "Модрина", 1.1f);
    arr[1] = new Wood(2, "Ялина", 0.9f);
    arr[2] = new Wood(3, "Сосна", 0.7f);
    count = 3;
}
```

Код цього блоку буде виконуватися у разі виклику конструктора за замовчуванням, під час створення об'єкту.

Далі створюємо публічні гетери до полів. Метод getCount() тут не наводимо, а метод getArr() переробимо таким чином, щоб він повертав копію масиву без пустих елементів:

```
public Wood[] getArr() {
    return Arrays.copyOf(arr, count);
}
```

Вибирати деревину з каталогу будемо за допомогою id. Для цього будемо використовувати наступний метод:

```
public Wood get(int id){
    for (int i = 0; i < count; i++) {
        if(arr[i].getId() == id)
            return arr[i];
    }
    return null;
}
```

Поповнювати каталог будемо за допомогою наступного методу:

```
public boolean add(Wood newWood) {
    // Перевірка на унікальність id
    if(get(newWood.getId()) != null)
        return false;
    // Запобігаємо переповненню масиву
    if(arr.length == count)
        arr = Arrays.copyOf(arr, count +count/2);
    // Додаємо новий елемент
    arr[count++] = newWood;
    return true;
}
```

Для того, щоб можна було переглянути вміст каталогу деревини створімо метод toString, але без допомоги автогенерації коду.

У цьому методі за допомогою циклу for накопичимо інформацію про усі види деревини у об'єкті типу StringBilder, після чого повернемо цю інформацію у вигляді одного об'єкту типу String:

```
public String toString() {
   StringBuilder sb = new StringBuilder("Каталог деревини:\n");
   for (int i = 0; i < count; i++) {
      sb.append(arr[i]).append("\n");
   }
   return sb .toString();
}
```

Після цього, за бажанням, у пакеті test можна створити клас, наприклад, TestApp з методом main для тестування і перевірити працездатність створених класів.

Текст відповідного методу main наведено нижче, його навіть можна скопіювати.

```
public static void main(String[] args) {
   // Створюємо каталог деревини
   WoodDirectory wd = new WoodDirectory();
   // Перевіряємо наявність елементу з id = 3
   System.out.println(wd.get(3)+"\n");
   // Створюємо новий вид деревини
   Wood w = new Wood (4, "Дуб", 1f);
   // Перевіряємо метод додавання
   if(wd.add(w))
      System.out.println(wd);
   // Спроба додати з існуючим id
   if(!wd.add(w))
        System.out.println(w + " вже існує\n");
}
```

2.2.3 *Створення класу lab2.model.Timber*

Цей клас реалізує сутность «брус».

Створімо у пакеті lab2.model клас Timber.

Визначимо приватні поля класу, що визначають його розміри та матеріал, з якого він зроблений:

```
private Wood wood;
private float length;
private float height;
private float width;
```

Далі слід згенерувати конструктор з параметрами.

Після цього можна також зробити публічні гетери для полів класу.

Визначимо також методи для визначення об’єму бруса та його ваги:

```
public float volume(){
     return length * height * width;
}

public float weight(){
     return wood.getDensity() * volume();
}
```

Зверніть увагу, ми не передаємо до методу volume параметри бруса бо цей метод викликається для об’єкту, а кожен об’єкт знає свої параметри.

Окрім того можна сгенерувати метод toString, який буде виводити, наприклад, назву деревини та вагу бруса.

2.2.4 *Створення класу lab2.store.ProductStore*

Цей клас призначений для збереження інформації про вироби з дерева, поки що це бруси.

Клас частково схожий на клас WoodDirectory. Різниця полягає у типах даних та відсутності блоку ініціалізації. Окрім того, не потрібен метод доступу через id та у методі add не потрібна перевірка на унікальність і цьому методу не потрібно щось повертати.

Метод toString має повертати перелік брусів.

Реалізуйте цей клас самостійно, беручи за зразок клас WoodDirectory і враховуючи попередні зауваження.

2.2.5 *Тестування створених класів*

Для тестування класів слід створити у пакеті lab2.test допоміжний клас, наприклад TestApp, з методом main. Можна переробити вже існуючий, якщо його було створено. Визначимо у класі поля для каталогу деревини та сховища для брусів.

```
// Каталог деревини
private WoodDirectory wd = new WoodDirectory();
// Каталог для брусів
private ProductStore ps = new ProductStore();
```

Метод main реалізуємо за стандартною схемою створення java додатків.

```
public static void main(String[] args) {
	TestApp app = new TestApp();
	app.startApp();
}
```

У цьому методі створюється об’єкт даного класу, після цього через цей об’єкт викликається метод, з якого починається робота додатку. У нашому випадку це метод startApp(). Внаслідок того, що метод ще не створено, Eclipse повідомить про помилку, яку він може допомогти виправити.

Для виправлення помилки слід навести курсор на імя’я методу, що підкреслено червоною лінією. Як результат, з’являється діалог, у якому вибираємо «create method».

Після появи заготовки для методу наповнюємо її кодом.

```
private void startApp() {
    //Наповнюємо сховище брусів
    ps.add(new Timber(wd.get(0), 5f, 0.5f, 0.4f));
    ps.add(new Timber(wd.get(1), 10f, 0.5f, 0.4f));
    //Друкуємо перелік продуктів
    System.out.println(wd);
    System.out.println(ps);
    //Обчислюємо вагу продуктів
    System.out.printf("Загальна вага: %1.3f", calcWeight());
}
```

У цьому методі ми використовуємо метод calcWeight(), який ще не створено. Тому слід сформувати заготовку цього методу і після цього наповнити її кодом:

```
private  float calcWeight() {
        float fullWeight = 0;
        for (Timber timber : ps.getArr()) {
                fullWeight+=timber.weight();
        }
        return fullWeight;
}
```

Після цього можна перевірити роботу додатку.

2.2.6 *Діаграма класів*

Наостанок побудуємо діаграму класів. Діаграма, яку наведено на рисунку 2.1, побудовано засобами Enterprise Architect. Можна скопіювати її і доопрацювати у редакторі зображень, але бажано навчитися користуватися якимось засобом для побудови UML діаграм.

Діаграма наочно показує вміст класів та зв'язки між ними.

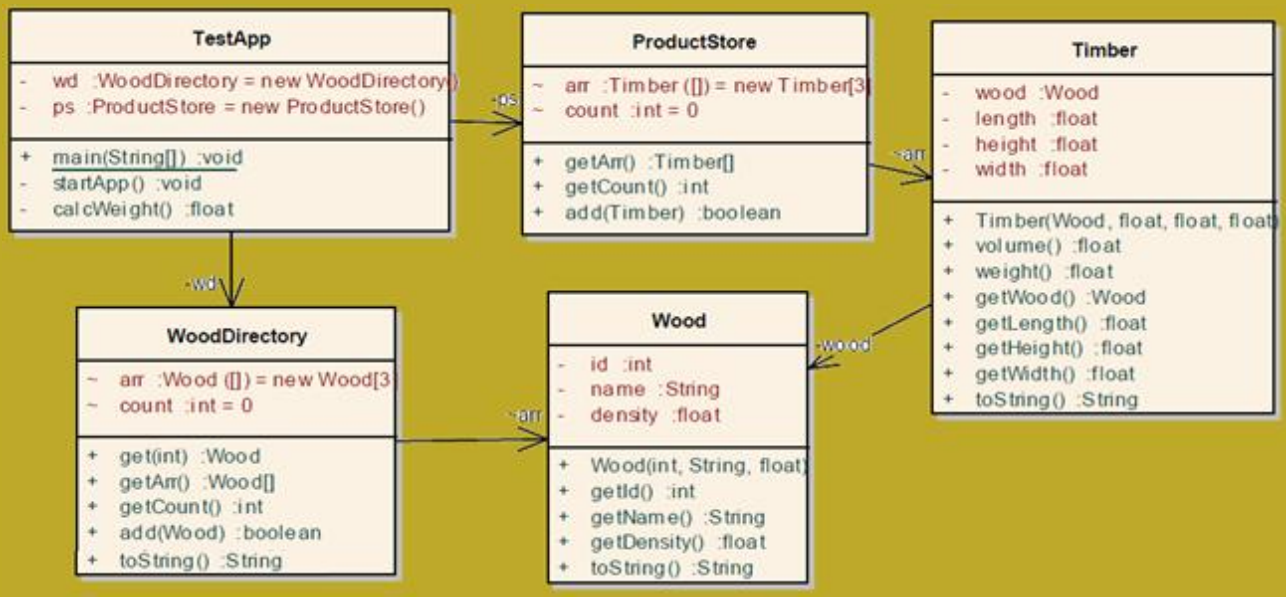

Рисунок 2.1 – Діаграма класів додатку

2.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Створити клас TestByConsole таким чином, щоб він забезпечив введення інформацію з консолі.

Меню програми має містити такі пункти:

1. Додати деревину
2. Додати брус
3. Підрахувати загальну вагу
4. Завершити роботу

Пункти меню мають виводитися у безкінечному циклі. Для вибору потрібного пункту використовувати switch або else if.

Пункти 1 – 3 реалізувати у вигляді окремих методів.

2.4 ВИМОГИ ДО ЗВІТУ

– Назва роботи.
– Мета роботи.
– Тексти класів, що було створено.
– Результати тестування додатку із копіями консолі.
– Текст класу, що був доопрацьований самостійно.
– Діаграма класів доопрацьованого додатку.
– Результати тестування доопрацьованого додатку.
– Висновки.

2.5 КОНТРОЛЬНІ ПИТАННЯ

1. Які засоби автогенерації коду використовувалися під час виконання роботи?
2. Які засоби виправлення помилок використовувалися.
3. Що являє собою програми на Java.
4. Що входить до складу класу?
5. Що таке блок ініціалізації і як він працює?
6. Як створити об’єкт?
7. Як створити масив?
8. Для чого використовувався клас Arrays у роботі?
9. Огляд одного з класів, розглянутих під час роботи, з докладними аналізом кожного рядка.
10. Зобразити від руки діаграму створених класів.

3 ЛАБОРАТОРНА РОБОТА № 3. СПАДКУВАННЯ ТА ПОЛІМОРФІЗМ JAVA

Мета роботи:

– Ознайомитися з реалізацією спадкування в Java.

– Ознайомитися зі шляхами реалізації поліморфізму в Java.

– Створити додаток з використанням спадкування і реалізацією поліморфізму.

3.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Припустимо, що деревообробне підприємство, для якого створювалося застосування на попередній лабораторній роботі, почало виготовляти також циліндричні бруси, і в подальшому, можливо, ще й інших форм.

Окрім того, з'ясувалося, що є великий попит на залишки деревини, що подрібнюються без сортування на види деревини та завантажуються у мішки.

Необхідно терміново доробити створений додаток. Інформація про продукцію, що відправляєтеся, має накопичуватися у одному сховищі.

3.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Закрийте усі класи з попередньої роботи, щоб уникнути плутанини.

Створіть новий проект для лабораторної роботи. Скопіюйте до теки src створеного проекту пакети з попередньої роботи. Після цього за допомогою рефакторингу поміняйте назви пакетів у відповідності з номером лабораторної роботи.

3.2.1 *Реалізація обліку виробів різної форми*

3.2.1.1 **Створення класу lab3.model.AbstractForm**

До цього класу ми винесемо поля і методи, що не залежать від форми виробу. А класи, що відповідають виробам конкретної форми, будуть успадковувати цей клас.

Для створення класу скористаймося технологією «рефакторинг».

На закладці «Package Explorer» виберіть клас Timber і активізуйте для нього контекстне меню. У меню виберіть функцію Refactor → Extract Superclass.

Як результат отримаємо діалог, що зображено на рисунку 3.1.

У діалозі вводимо ім'я суперкласу AbstractForm і ставимо маркери для полів та методів, які потрібно перенести до суперкласу.

У нашому випадку поле wood можна використовувати для виробів будь якої форми. Тому вибираємо це поле і метод доступу до нього. Окрім того, вибираємо метод розрахунку ваги weight(). Цей метод є універсальним для будь якої форми, тому що він використовує метод volume() для розрахунку об'єму виробу. А от метод volume() вже залежить від форми конкретного виробу.

Далі натискаємо кнопку «Finish».

Eclipse попереджує про помилку, що пов’язана з відсутністю методу volume() у новому класі. Але все одно ще раз натискаємо кнопку «Finish».

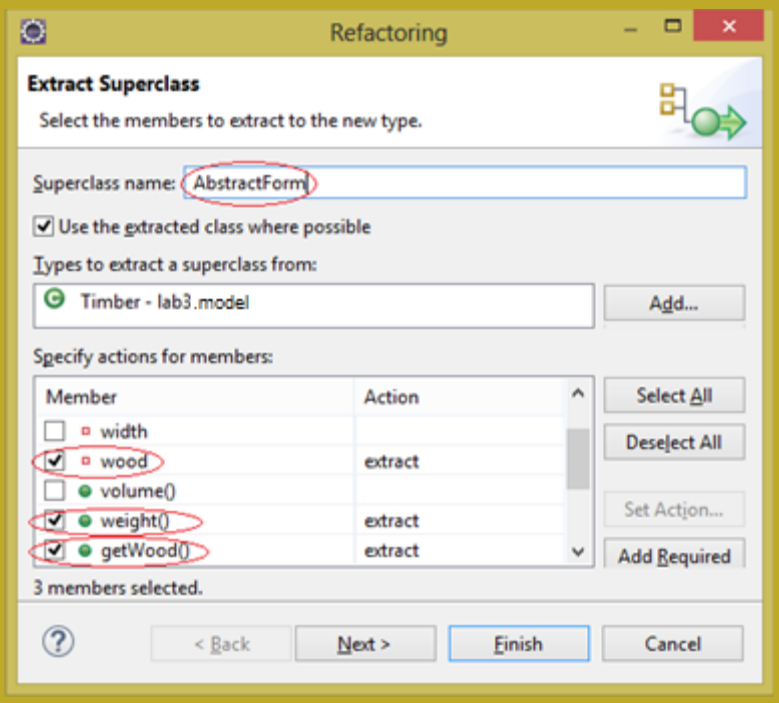

Рисунок 3.1 – Діалог виділення суперкласу із класу

Як результат отримуємо новий клас, але з помилкою.

Далі відкриваємо новий клас для редагування.

Як вже було сказано, Eclipse повідомляє про те, що у методі weight() використовується невідомий метод volume(). Ми знаємо, що такий метод реалізовано у класі Timber, який став спадкоємцем даного класу. Але для того, щоб сповістити про це компілятор, необхідно у даному класі визначити метод volume(), як абстрактний, тобто тільки заголовок. Це схоже на прототип функції в мові С.

```
public abstract float volume();
```

Але внаслідок появи абстрактного методу у класі необхідно і клас оголосити абстрактним. Це найпростіше зробити через діалог виправлення помилок.

```
public abstract class AbstractForm { ...
```

Зверніть увагу також на специфікатор доступу для поля wood, що змінився з private на protected. Це дозволить користуватися цим полем і спадкоємцям класу.

3.2.1.2 **Аналіз змін у класах Timber, ProductStore, TestApp**

Внаслідок операції рефакторингу було не тільки створено новий клас, але й відбулися зміни у класах Timber, ProductStore, TestApp.

Перегляньте клас Timber і порівняйте його код з кодом із попередньої роботи. Результати мають бути зафіксовані у звіті.

Перегляньте клас ProductStore. У ньому теж відбулися певні зміни, але цих змін недостатньо для того, щоб це сховище зберігало вироби різних типів.

Для цього слід змінити тип масиву arr з Timber на AbsractForm.

```
private AbstractForm[] arr = new AbstractForm[3];
```

Зверніть увагу, тут ми використовуємо абстрактний тип для визначення типу даних, що будуть зберігатися у масиві. Завдяки цьому у масиві можна буде зберігати об'єкти будь яких класів, що успадковують клас AbstractForm. Можна сказати, що масив arr став поліморфним, тобто здатним приймати дані різних типів.

У класі ProductStore слід також змінити тип параметру, що передається до методу add.

```
public void add(AbstractForm newProduct) {
```

Ця заміна дозволяє використовувати метод для передачі до сховища об'єкти будь яких класів, що успадковують клас AbstractForm.

Деякі зміни відбулися і у класі TestApp. Зафіксуйте ці зміни у звіті. Але доопрацювання цей клас не потребує.

Після внесених змін слід протестувати роботу додатку. Додаток має працювати так само, як і у попередній роботі.

3.2.1.3 **Створення класу lab3.model.Cylinder**

Цей клас буде реалізовувати сутність круглий брус.

Для спрощення роботи спочатку позначте пакет, у якому буде створюватися клас. Далі скористаймося майстром створення класів, що викликається через меню File → New → Class.

У діалозі, що з'явився, рисунок 3.2, слід ввести ім'я нового класу, а також вибрати проект та пакет, якщо перед цим пакет не було виділено.

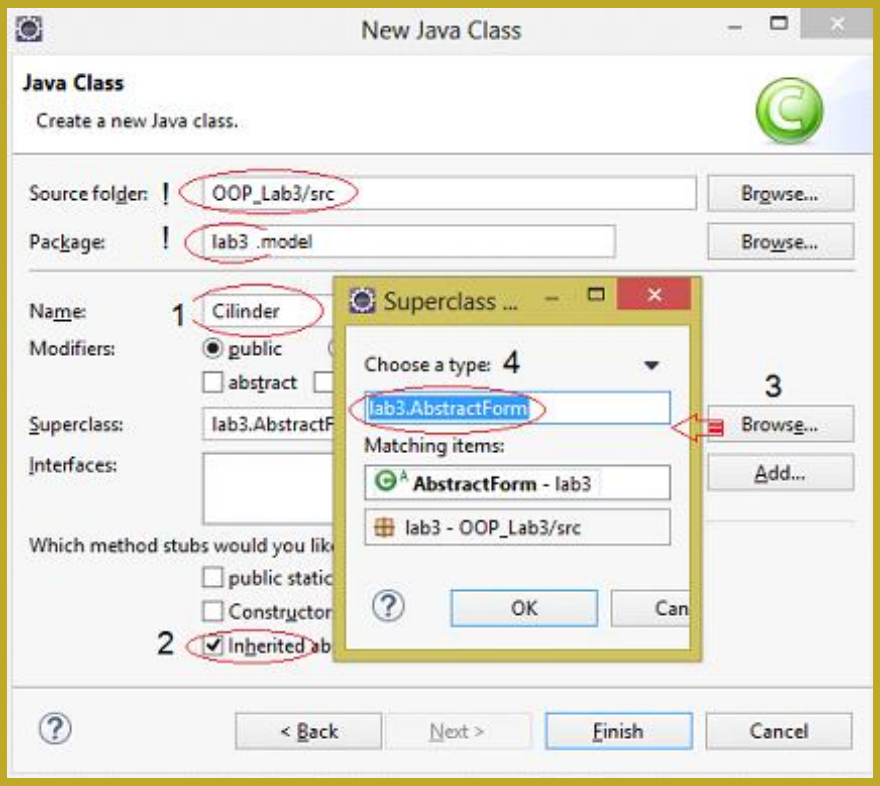

Рисунок 3.2 – Діалог створення класу з суперкласом

Далі слід включити перемикач inherited abstract methods, що забезпечує створення у класі заготовок для реалізації абстрактних методів оголошених у суперкласі. У нашому випадку це буде метод volume().

Після цього слід скористатися кнопкою Browse, щоб вибрати суперклас. Суперкласом буде клас AbstractForm.

Далі натискаємо Finish і отримуємо заготовку класу Cylinder.

```
package lab3;

public class Cilinder extends AbstractForm {

	@Override
	public float volume() {
		// TODO Auto-generated method stub
		return 0;
	}

}
```

Далі у класі слід визначити поля, що визначають довжину та діаметр бруса.

```
private float length;
private float diameter;
```

Маючи ці змінні, можна завершити метод volume. Константу π можна отримати у класі Math, але вона має тип double, тому результат доведеться привести до типу float, але це можна зробити за допомогою Eclipse, вибравши у меню виправлення помилки «Add cast to 'float'»

Далі слід додати до класу конструктор з параметрами. Його можна згенерувати за допомогою меню Source, але параметр wood доведеться додати, бо майстер створення конструктора з полями не бачить суперкласу. Можна також скопіювати конструктор з класу Timber та корегувати його.

Далі можна згенерувати гетери для полів і метод toString.

3.2.1.4 **Тестування додатку**

Для тестування змін можна скористатися класом TestApp, у якому до сховища додати ще декілька циліндрів. Зверніть увагу, у методі calcWeight() так само викликається метод weight(), але для бруса і циліндра він буде виконуватися по різному, бо використовує різні реалізації методу volume(). Тому метод weight() можна назвати узагальненим, а методи volume() – поліморфними, бо вони мають різні реалізації. Поліморфізм дає можливість створювати узагальнені методи, і це одна з переваг ООП.

3.2.1.5 **Діаграма класів**

Діаграма класів зображена на рисунку 3.3. У порівнянні з попередньою тут з'явилося два нових класи AbstractForm та Cylinder.

Зверніть також увагу на зображення зв’язку між суперкласом та його спадкоємцями.

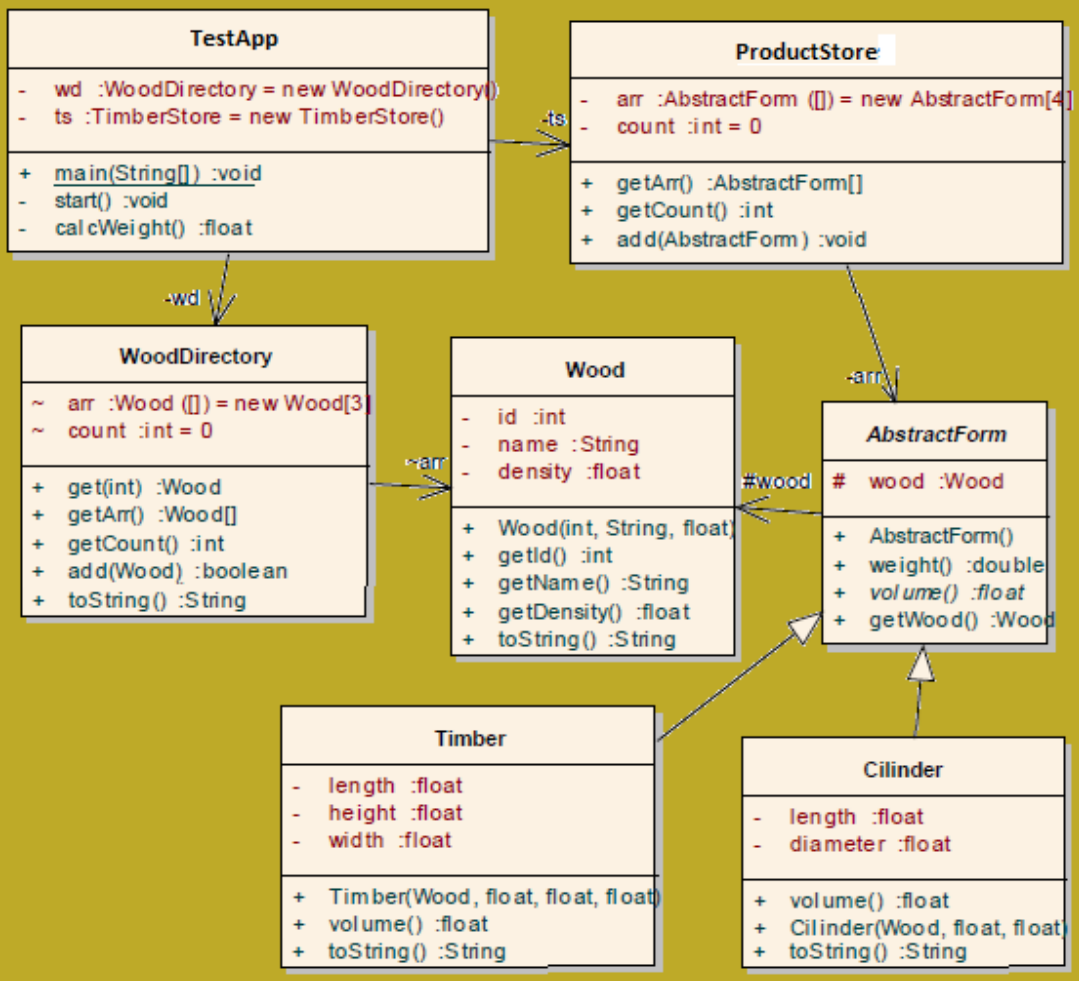

Рисунок 3.3 – Діаграма класів з AbstractForm

3.2.2 *Реалізація обліку залишків виробництва*

Мішки із залишками виробництва не вписуються до ієрархії класів AbstractForm, бо заповнюються вони будь якою деревиною і форма мішків не визначена. Мішки зважують, і вага є їх єдиною характеристикою.

Але інформацію про мішки треба теж зберігати у тому ж сховищі, що й бруси, і потрібно враховувати їх вагу при завантаженні.

3.2.2.1 **Створення інтерфейсу lab3.model.IWeight**

Єдине, що зв’язує мішки та бруси різної форми – це їх вага, і саме цей параметр є важливим для контролю завантаженості барж. Тому ми введемо новий тип даних – «такий, що має вагу» і назвемо його IWeight. Тип визначає сигнатуру методу weight(), що має повертати вагу об’єкту, але не містить його реалізації. Реалізація методу має бути прописана у класах, що успадковують IWeight. Цей тип також не потребує наявності у об’єктів яких небудь параметрів.

Такі типи, що не визначають властивості об’єктів і містять тільки опис абстрактної поведінки об’єктів у вигляді сигнатур методів, називають

інтерфейсами. Перевага інтерфейсу над абстрактним класом полягає у тому, що інтерфейсів можна успадковувати скільки завгодно, а клас можна успадкувати тільки один.

Потрібний нам інтерфейс IWeight можна створити із класу AbstractForm шляхом рефакторінгу.

Для цього слід виділити клас AbstractForm і викликати контекстне меню Refactor → Extract Interface. Далі у діалозі, що з'явився, визначити методи для інтерфейсу, рисунок 3.4, і натиснути OK.

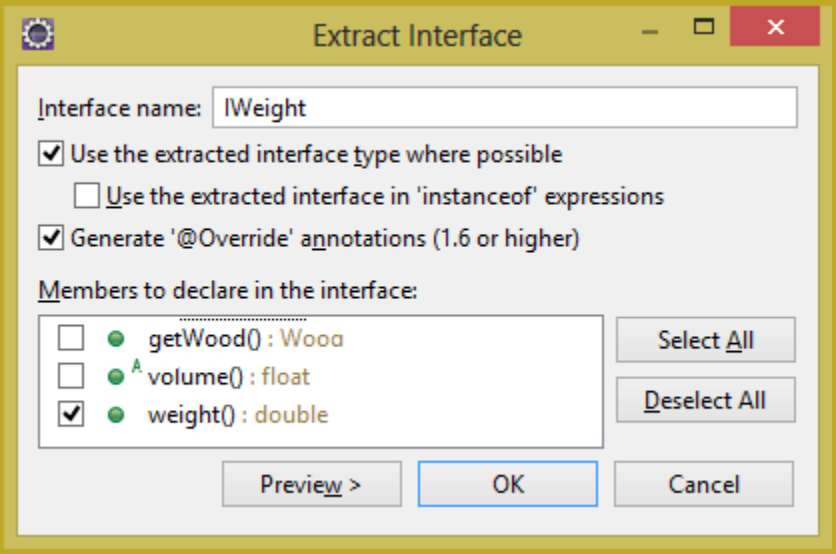

Рисунок 3.4 – Діалог виділення інтерфейсу із класу

Як результат виконаних операції ми отримуємо інтерфейс IWeight та зміни у класах AbstractForm, ProductStore, TestApp.

Перегляньте інтерфейс IWeight та зафіксуйте його у звіті.

3.2.2.2 **Аналіз змін у класах AbstractForm, ProductStore, TestApp**

Перегляньте класи AbstractForm та TestApp і знайдіть зміни у коді. Перегляньте клас ProductStore. У ньому теж відбулися певні зміни, але цих змін недостатньо для того, щоб це сховище зберігало новий тип виробів.

Треба змінити тип масиву arr з AbsractForm на IWeight.

Слід також змінити тип параметру, що передається до методу add на IWeight.

Ця заміна дозволяє використовувати метод для передачі до сховища об'єкти будь яких класів, що реалізують інтерфейс IWeight.

Після внесених змін слід протестувати роботу додатку. Додаток має працювати так само, як і раніше.

3.2.2.3 **Створення класу lab3.model.Waste**

Цей клас буде реалізовувати сутність мішок з відходами обробки деревини.

Для спрощення роботи спочатку позначте пакет, у якому буде створюватися клас. Далі скористаймося майстром створення класів, що викликається через меню File → New → Class.

У діалозі, що з’явився, рисунок 3.5, слід ввести ім’я нового класу, а також вибрати проект та пакет, якщо перед цим пакет не було виділено.

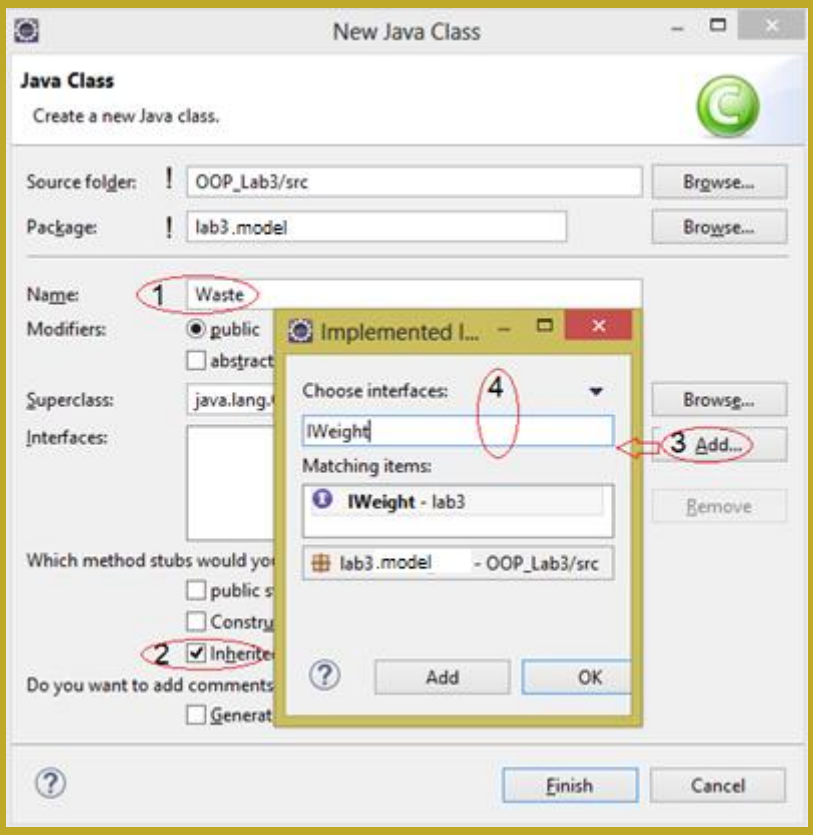

Рисунок 3.5 – Діалог створення класу з інтерфейсом

Далі слід включити перемикач inherited abstract methods, що забезпечує створення у класі заготовок для реалізації абстрактних методів оголошених у суперкласі. У нашому випадку це буде метод weight().

Після цього слід скористатися кнопкою Add, щоб вибрати інтерфейс. Це має бути IWeight.

Після вибору інтерфейсу натискаємо Finish і отримуємо заготовку класу Waste.

Далі у класі слід визначити поле weight, що визначає вагу мішка

Маючи цю змінну, можна завершити метод weight(), який має просто повертати значення поля weight.

Далі слід додати до класу конструктор з одним параметром. Його можна згенерувати за допомогою меню Source.

На завершення слід згенерувати метод toString.

3.2.2.4 **Тестування додатку з інтерфейсом IWeight**

Для тестування змін можна скористатися класом TestApp, у якому до сховища додати ще мішків з відходами.

Тепер методі weight(), що викликається з calcWeight(), використовуються вже три поліморфні реалізації методу volume(). Використання інтерфейсу IWeight для різнорідних сутностей дозволило розширити можливості узагальненого методу weight().

3.2.2.5 **Діаграма класів з інтерфейсом IWeight**

Діаграма класів зображена на рисунку 3.6. У порівнянні з попередньою тут збільшилася кількість ієрархічних відносин.

Зверніть увагу на різницю у зображенні спадкування від класу та від інтерфейсу.

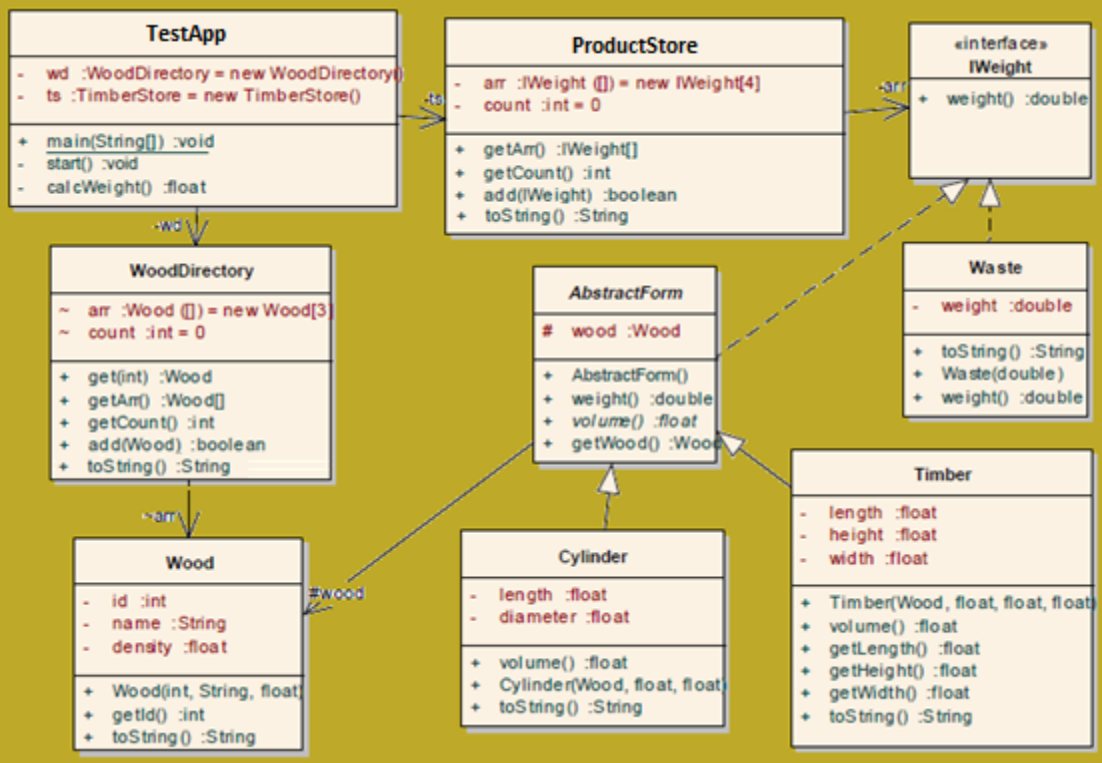

Рисунок 3.6 – Діаграма класів з інтерфейсом IWeight

3.2.3 *Створення ієрархії класів для сховищ даних*

Якщо порівняти класи ProductStore та WoodDirectory, то можна побачити багато схожих, а то й однакових членів. Тому можна спробувати винести однакові елементи до суперкласу. Різницю у типах даних можна ліквідувати, використовуючи тип даних Object для об’єктів, що зберігаються.

Виділення суперкласу шляхом рефакторингу у цьому випадку не буде зручним внаслідок різниці у типах. Тому ієрархію класів зробимо вручну.

3.2.3.1 **Створення суперклассу lab3.store.AbstractStore**

Створімо у пакеті lab3.store копію класу ProductStore з назвою AbstractStore.

Після цього змінюємо тип IWeight для членів класу і у методах на Object. Це можна зробити викликавши діалог пошук/заміна клавішами Ctrl+F.

Окрім того поміняємо специфікатори доступу полів та методу add на protected. У такому разі поля і метод будуть доступними й нащадкам. Але з пакету test він буде недоступний. А у нащадків методи add зробимо публічними.

У методі toString вилучимо параметр конструктора класу StringBuilder.

Після цих змін клас може виглядати так:

```
public class AbstractStore {
	protected Object[] arr = new Object[3];
	protected int count = 0;

	public Object[] getArr() {
		return Arrays.copyOf(arr, count);
	}
	public int getCount() {
		return count;
	}
	protected void add(Object newObject) {
		// Запобігаємо переповненню масиву
		if(arr.length == count)
			arr = Arrays.copyOf(arr, count +count/2);
		// Додаємо новий елемент
		arr[count++] = newObject;
	}
	public String toString(){
		StringBuilder sb = new StringBuilder();
		for (int i = 0; i < count; i++)
			sb.append(arr[i]).append("\n");
		return sb.toString();
	}
}
```

3.2.3.2 **Спрощення класу lab3.store.ProductStore**

Додайте зміни до заголовку класу, які будуть вказувати на те, що клас ProductStore успадковує клас AbstractStore.

```
public class ProductStore extends AbstractStore{
```

Тепер видалимо поля arr, count, методи getCount(), getArr().

Метод add() змінимо, використовуючи аналогічний метод суперкласу, що дозволить контролювати тип даних, що додаються до сховища:

```
public void add(IWeight newProduct) {
   super.add(newProduct);
}
```

А метод toString перетворимо у такий:

```
public String toString() {
   StringBuilder sb = new StringBuilder("Вміст сховища продуктів:\n");
   sb.append(super.toString());
   return sb.toString();
}
```

Префікс super перед викликом методів add() та toString() означає, що тут потрібно викликати такий самий метод із суперкласу.

3.2.3.3 **Спрощення класу lab3.store.WoodDirectory**

Так само, як і у попередньому класі слід успадкувати клас WoodDirectory від AbstractStore. Далі слід видалити поля arr, count, методи getCount(), getArr() . Методи get() та add() змінимо, щоб пристосувати їх до конкретного типу:

```
public Wood get(int id){
     for (int i = 0; i < count; i++) {
          Wood wood = (Wood) arr[i];
          if(wood.getId() == id)
               return wood;
     }
     return null;
}

public boolean add(Wood newWood) {
     // Перевірка на унікальність id
     if(get(newWood.getId()) != null)
          return false;
     super.add(newWood);
     return true;
}
```

Метод toString теж змінимо використовуючи метод суперкласу так само, як і у класі ProductStore.

3.2.3.4 **Тестування додатку з суперкласом AbstractStore**

Для тестування змін можна скористатися класом TestApp, але метод calcWeight() доведеться корегувати, виходячи з того, що метод getArr() повертає об’єкти типу Object:

```
private  float calcWeight() {
      float fullWeight = 0;
      for (Object timber : ps.getArr()) {
            fullWeight+=((IWeight)timber) .weight();
      }
      return fullWeight;
}
```

3.2.4 *Діаграма класів з суперкласом AbstractStore*

Діаграма класів сховищ з класом AbstractStore наведена на рисунку 3.7.

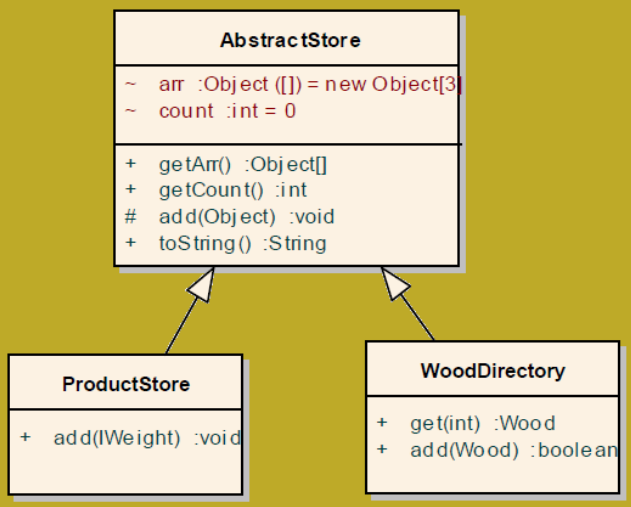

Рисунок 3.7 – Діаграма класів сховищ

3.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Доопрацювати консольний додаток таким чином, щоб він працював з усіма видами продуктів і новою ієрархією класів.

3.4 ВИМОГИ ДО ЗВІТУ

– Назва роботи.
– Мета роботи.
– Тексти класів, що були створені під час виконання лабораторної роботи.
– Тексти класів, що були створені для реалізації завдання на самостійну роботу.
– Діаграма класів з урахуванням завдання на самостійну роботу.
– Результати тестування застосування у вигляді копії консолі.
– Висновки.
–

3.5 КОНТРОЛЬНІ ПИТАННЯ

1. Спадкування та його реалізація в Java.
2. Поліморфізм та його реалізація в Java.
3. Покажіть на діаграмі класів поліморфні та узагальнені методі.
4. У чому різниця між звичайним класом, абстрактним класом та інтерфейсом.
5. Що таке приведення типів і як воно реалізується.
6. Для чого і як використовувався рефакторинг.

4 ЛАБОРАТОРНА РОБОТА № 4. ІНКАПСУЛЯЦІЯ ТА ВИНЯТКИ В JAVA

Мета роботи:

– Ознайомитися з реалізацією інкапсуляції в Java.
– Ознайомитися з механізмами формування та обробки виключних ситуацій.
– Створити додаток з формуванням винятків та їх обробкою.

4.1 ЗАВДАННЯ НА РОБОТУ

За допомогою додатку, який було створено на останній лабораторній роботі, деревообробна компанія почала контролювати завантаженість барж, щоб забезпечити безпечність перевезень.

Та одного разу, під час негоди, одна з барж перекинулась. Аналіз причин показав, що баржа була перевантажена внаслідок того, що недосвідчений оператор вводив вагу мішків у центнерах, хоча треба було у тонах.

Компанія звернулася з проханням забезпечити у розробленому додатку контроль введення даних.

4.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Закрийте усі класи з попередньої роботи, щоб уникнути плутанини.

Створіть новий проект для лабораторної роботи. Скопіюйте до теки src створеного проекту пакети з попередньої роботи. Після цього за допомогою рефакторингу поміняйте назви пакетів у відповідності з номером лабораторної роботи.

4.2.1 *Аналіз задачі*

Контроль правильності налаштування властивостей об’єкта містить у собі дві складові. Перша – це виявлення помилки (виключної ситуації), друга – це організація реакції на помилку.

Існує також декілька шляхів вирішення цієї проблеми. Але найбільш прогресивний полягає у тому, що виявлення виключної ситуації відбувається на рівні моделі, а реакція на виключну ситуацію реалізується на рівні шару подання (інтерфейсу користувача)

ООП дозволяє достатньо просто реалізувати перевірки на рівні моделі. Принцип інкапсуляції передбачає, що доступ до полів, які визначають властивості об’єктів, можливий тільки через методи або конструктори. Тому, якщо дотримуватися цього принципу, то можна контролювати значення властивостей об’єктів у методах або конструкторах під час їх виклику.

Для того щоб у разі виявлення виключної ситуацію можна було б передати інформацію про неї до шару подання у Java використовуються об’єкти класу Exception, або його підкласів.

У нашому додатку поле weight було приватним, тобто інкапсульованим.

Його значення можна було встановити тільки через конструктор, але введене значення там не контролювалося. Тому потрібно додати до конструктору класу Waste перевірку значення поля weight і в разі виявлення помилки сформувати об’єкт типу Exception.

4.2.2 *Реалізація контролю значення поля weight*

Відкрийте конструктор класу Waste і реалізуйте перевірку параметру weight перед присвоєнням його значення полю weight.

Хай, наприклад воно має примати значення в межах від 0.020 до 0.100. В разі виявлення помилки сформуйте об’єкт типу Exception і передайте його на подальшу обробку за допомогою інструкції **throw**.

```
if(weight < 0.020 || weight > 0.100)
   throw new Exception(weight +" is not correct!\n" +
       "Must be from 0.020 to 0.100");
```

Як видно з наведеного коду, об’єкт класу Exception може містити в собі повідомлення про помилку.

Після внесення таких змін Eclipse повідомить про помилку. У меню виправлення цієї помилки слід вибрати пункт add throws declaration. Внаслідок цього до заголовку конструктора буде добавлена фраза «throws Exception» (кидає виняток)

```
public Waste(float weight) throws Exception {
   if(weight < 0.020 || weight > 0.100)
      throw new Exception(weight +" is not correct!\n" +
          "Must be from 0.020 to 0.100");
   this.weight = weight;
}
```

4.2.3 *Реалізація реакції на викид винятку*

Реакцію на викид винятку запрограмуємо у методі start класу TestApp. Після доопрацювання конструктора класу Waste у цьому методі має з’явитися помилка. Eclipse вимагає розібратися з виключною ситуацією, що може виникнути під час використання конструктора класу Waste. Помилку можна виправити вибравши із меню варіантів виправлення помилок функцію “Surround with try/catch”.

У розділі catch, що відповідає за обробку виключної ситуації, ми запрограмуємо виведення віконця з повідомленням про помилку.

Виправлений фрагмент коду може виглядати так:

```
try {
   ps.add(new Waste(50f));
} catch (Exception e) {
   JOptionPane.showMessageDialog(null, e.getMessage(),
         "Введення продуктів", JOptionPane.ERROR_MESSAGE);
}
```

У разі, якщо після виклику конструктора виникне виключна ситуація, керування програмою буде передано у блок catch разом з об’єктом «e», який

несе с собі інформацію про виключну ситуацію. На екрані з'явиться діалогове вікно з повідомленням про помилку. Після його закриття програма продовжить роботу, але продукт з хибним параметром прийнято не буде.

4.2.4 *Доопрацювання конструкторів інших класів*

Додайте до конструкторів класів Timber, Cylinder контроль даних, що передаються через параметри конструкторів, та доопрацюйте клас TestApp.

4.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Доопрацюйте створений раніше додаток забезпечивши контроль даних, що вводяться через консоль. У разі помилки програма має видати запит на повторне введення даних

4.4 ВИМОГИ ДО ЗВІТУ

- Назва роботи.
- Мета роботи.
- Тексти методів, що були доопрацьовані підчас виконання роботи.
- Тексти методів, що були доопрацьовані для реалізації завдання на самостійну роботу.
- Результати тестування застосування у вигляді копії консолі і екрану.
- Висновки

4.5 КОНТРОЛЬНІ ПИТАННЯ

1. Інкапсуляція та її реалізація в Java.
2. Специфікатори доступу
3. Формування виключних ситуацій.
4. Обробка виключних ситуацій.

5 ЛАБОРАТОРНА РОБОТА № 5.
ПОТОКИ ВВЕДЕННЯ-ВИВЕДЕННЯ І СЕРІАЛІЗАЦІЯ ОБ'ЄКТІВ

Мета роботи:

- Ознайомитися с потоками введення-виведення в Java.
- Ознайомитися с інтерфейсом Serializable і специфікатором transient.
- Придбати навички роботи з потоками введення-виведення.
- Забезпечити збереження інформації у файлу та її відновлення у створюваному проекті.

5.1 ЗАВДАННЯ НА РОБОТУ

Після аварії, що трапилася, деревообробна компанія поставила завдання перед розробниками додатку фіксувати у текстовому файлі усі операції по введенню даних з фіксацією дати та прізвища оператора перед початком роботи.

5.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Закрийте усі класи з попередньої роботи, щоб уникнути плутанини.

Створіть новий проект для лабораторної роботи. Скопіюйте до теки src створеного проекту пакети з попередньої роботи. Після цього за допомогою рефакторингу поміняйте назви пакетів у відповідності з номером лабораторної роботи.

5.2.1 *Знайомство з класом JFileChooser*

Об'єкти типу JFileChooser використовуються для відображення списку файлів і вибору користувачем файлу або папки.

Створімо новий клас у пакеті test, наприклад TestFile, з методом main. До методу main додаймо такий код.

```
JFileChooser dialog = new JFileChooser();
dialog.showOpenDialog(null);
File f = dialog.getSelectedFile();
if(f != null){
   System.out.println(f.getName());
   System.out.println(f.getAbsolutePath());
}
```

Після запуску програми на екрані з'являється діалог для відкриття файлу і за його допомогою можна вибрати файл.

Після вибору файлу, або групи файлів, діалог повертає об'єкт класу File, але це не файл, назва класу оманлива. Об'єкти класу File – це ніби то посередники, які забезпечують доступ до імені вибраного файлу.

Два останні рядки коду показують, як можна отримати повне та скорочене ім'я файлу.

Діалог виглядає дещо застарілим, але він може підлаштовуватися під операційне оточення. Для цього слід додати таке налаштування програми перед зверненням до діалогу, або на початку роботи програми:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

Цей вираз вимагає конструкції try/catch, зробіть це.

Код і результат виконання зафіксуйте у звіті.

5.2.1.1 **Зміна налаштувань об’єкту JFileChooser**

Діалог можна налаштовувати, починаючи від того, чи дозволити обирати кілька файлів чи ні, та закінчуючи своїми назвами для різних елементів в ньому.

Нижче наведено приклад налаштувань діалогу, та отримання групи вибраних файлів. Протестуйте цей код, створивши новий клас, та зафіксуйте код та результати у звіті.

```
JFileChooser dialog = new JFileChooser();
dialog.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
dialog.setDialogTitle("Виберіть потрібні файли та папки");
dialog.setApproveButtonText("Відкрити");
dialog.setMultiSelectionEnabled(true);
dialog.showOpenDialog(null);
File[] ff = dialog.getSelectedFiles();
if (ff != null) {
     for(File f : ff)
          System.out.println(f.getAbsolutePath());
}
```

Надалі у заголовку діалогу налаштовуйте своє ім’я і групу, щоб звіт можна було ідентифікувати.

5.2.1.2 **Налаштування фільтру для об’єкту JFileChooser**

В компоненті можна задавати фільтр, який визначає які файли відображати користувачеві. Для створення фільтра тут використовується об'єкт анонімного класу, що реалізує інтерфейс javax.swing.filechooser.FileFilter.

Для створення фільтру додайте перед відкриттям діалогу початкового варіанту у класі TestFile такий рядок коду, після new має бути ще пробіл:

```
dialog.setFileFilter(new );
```

На помилку не звертайте поки що уваги. Для її виправлення розташуйте курсор після new з пробілом і натисніть Ctrl+Пробіл. Має з’явитися запрошення не генерацію варіантів коду. Виберіть FileFilter() Anonymous Inner Type … Після цього Eclipse згенерує тіло анонімного класу із заготовками двох методів, які слід реалізувати таким чином:

```
dialog.setFileFilter(new FileFilter() {
    @Override
    public String getDescription() {
       return "файли типу .txt";
    }
    @Override
    public boolean accept(File f) {
       if (f != null){
          // Відображати усі папки та файли .txt
          return f.isDirectory() || f.toString().endsWith(".txt");
       }
    return false;
    }
});
```

Перевірте роботу фільтру і код та результат зафіксуйте у звіті.

5.2.2 *Виведення рядків символів до текстового файлу*

Для виведення у текстовий файл можна використовувати об'єкт класу BufferedWriter, передавши йому як параметр об'єкт класу FileWriter, налаштований на обраний файл.

Нижче наведено приклад, де у текстовий файл виводиться послідовність випадкових чисел. Додайте цей код до методу main після відкриття файлу f, що дозволить створити новий текстовий файл з випадковими числами.

```
// Формування текстового файлу з числами
try {
     BufferedWriter writer = new BufferedWriter(new FileWriter(f));
     for (int i = 0; i < 10; i++) {
          double x = Math.random();
          String s = String.valueOf(x);
          writer.write(s);
          writer.newLine();
     }
     writer.close();
} catch (Exception e) {
     e.printStackTrace();
}
```

Додайте до коду виведення вашого імені та групи на початку, або наприкінці файлу. Скорегований код додайте до звіту.

5.2.3 *Читання рядків із текстового файлу*

Для зчитування рядків текстового файлу у об’єкти типу String, можна використовувати об'єкт класу BufferedReader, що створюється як обгортка для об’єкту типу FileReader.

Додайте до методу main у новому класі, після відкриття файлу f рядки коду, що забезпечать виведення змісту текстового файлу на консоль.

```
BufferedReader reader = null;
if (f != null) {
     try {
          reader = new BufferedReader(new FileReader(f));
          String s;
          while ((s = reader.readLine()) != null){
               System.out.println(s);
          }
          reader.close();
     } catch (Exception e) {
          e.printStackTrace();
     }
}
```

Протестуйте роботу доопрацьованої програми.

Збережіть у звіті метод main, та результати з консолі.

5.2.4 *Збереження об'єктів у файл*

Процес перетворення об'єкту у потік байтів з подальшим його збереженням називають серіалізацію(serialization).

Зовні, цей процес мало чим відрізняється від запису рядка у текстовий файл. Просто цю роботу виконують об'єкти інших класів, а саме – використовуються об'єкт класу FileOutputStream, та як оболонка для нього, об'єкт класу ObjectOutputStream.

Створіть новий клас з методом main, наприклад, TestStoreObject.

До методу main додайте такий код:

```
// Створення довідника деревини
WoodDirectory wd = new WoodDirectory();
// Додаємо новий вид деревини
wd.add(new Wood(4,"Дуб", 1f));
//Збереження каталогу у файлі
File f = new File("wd.object");
try {
     FileOutputStream fos = new FileOutputStream(f);
     ObjectOutputStream oos = new ObjectOutputStream(fos);
     oos.writeObject(wd);
     oos.close();
} catch (Exception e) {
     e.printStackTrace();
}
```

Спроба протестувати цей код має призвести до помилки, навіть якщо компілятор пропустить програму. Справа у тому, що серіалізувати можна тільки об'єкти, що реалізують інтерфейс java.io.Serializable. Цей інтерфейс є прикладом інтерфейсів-маркерів. Він не вимагає реалізації будь яких методів, а просто вказує на те, що об'єкт можна серіалізувати.

Тому для виправлення помилки необхідно додати цей інтерфейс до класів AbstractStore та Wood.

Зробіть це і протестуйте програму. Після цього з меню проекту виберіть функцію Refresh. У папці проекту має з'явитися файл wd.object.

5.2.5 *Відновлення об’єкту з файлу*

Процес перетворення потоку байтів у об’єкт називають десеріалізацією (deserialization).

Цей процес схожий на процес серіалізації. Але роботу виконують вже об’єкти інших класів, а саме – клас FileInputStream, та класу ObjectInputStream.

Створіть у пакеті test новий клас з методом main, наприклад, TestRestoreObject.

До методу main додайте такий код:

```
WoodDirectory wd = null;
// Відновлення довідника з файлу
File f = new File("wd.object");
try {
     FileInputStream fis = new FileInputStream(f);
     ObjectInputStream ois = new ObjectInputStream(fis);
     wd = (WoodDirectory) ois.readObject();
     ois.close();
} catch (Exception e) {
     e.printStackTrace();
}


// Виведення довідника на консоль
if (wd != null){
     for(Object w : wd.getArr())
          System.out.println(w.toString());
}
```

5.2.6 *Використання модифікатора transient*

Модифікатор transient використовується у ситуаціях, коли збереження деякого поля об’єкту небажано, наприклад, якщо у полі зберігається деяка конфіденційна інформація, наприклад пароль. Навіть якщо інформація у об'єкті описана як закрита (private), то секретні дані можна витягти з файлу або з перехопленого мережевого пакету.

Для того, щоб перевірити вплив цього модифікатора на процес збереження даних, задайте цей модифікатор для поля density класу Wood. Після цього збережіть довідник і відновіть його, використовуючи вже створені програми.

Результат зафіксуйте у звіті.

5.3 Завдання на самостійну роботу

Доопрацювати додаток, створений на попередніх лабораторних роботах, Додавши до нього реалізацію таких функцій:

1) Серіалізація каталогу деревини
2) Серіалізація переліку відправлених продуктів
3) Десеріалізація каталогу деревини
4) Десеріалізація переліку відправлених продуктів

5) Виведення до текстового файлу переліку відправлених продуктів

6) Протоколювання дій по введенню продуктів у текстовому файлі. Повинні протоколюватися такі події:

– запуск додатка із записом дати, часу та реквізитів студента;
– додавання продуктів.

Вирішити останнє завдання можна, виконавши таку послідовність дій:

– створіть у головному класі з методом main поле типу BufferedWriter для потоку виводу тексту;
– у цьому ж класі створіть метод writeString(String), який буде виводити рядок тексту у створений потік і обробляти виняткову ситуацію;
– у методі start створіть об'єкт класу BufferedWriter, використовуючи комбінацію конструкторів new BufferedWriter (new FileWriter ("Log.txt"));
– у цьому ж методі виведіть початкову інформацію для протоколу;
– у цьому ж методі, після закриття циклу введення інформації закрийте потік виводу тексту, використовуючи метод close().
– після цього доцільно перевірити, чи працює ваш логер;
– далі, у методах додавання продуктів використовуйте метод writeString(String) для виводу відповідних повідомлень.

5.4 Зміст звіту

– Назва роботи.
– Мета роботи.
– Тексти методів роботи з файлами та відповідні копії екрану та консолі.
– Тексти методів серіалізації і десеріалізації.
– Код створення протоколу роботи з додатком.
– Протокол роботи з додатком.
– Висновки

5.5 Контрольні питання

– Робота з діалогом вибору файлів.
– Виведення інформації до текстового файлу.
– Введення інформації з текстового файлу.
– Серіалізація і десеріалізація об'єктів.
– Інтерфейс Serializable.
– Модифікатор transient.
– Опис механізму формування протоколу роботи з додатком.

6 ЛАБОРАТОРНА РОБОТА № 6
ІТЕРАТОРИ

Мета роботи:
- Ознайомитися з інтерфейсами Iterable та Iterator.
- Ознайомитися з поняттям «внутрішній клас» в Java.
- Ознайомитися з прикладом реалізації інтерфейсу Iterator.
- Самостійно реалізувати ітератор типу ListIterator.
- Створити програму з використанням ітераторів.

6.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

У цій роботі ми повинні реалізувати інтерфейс Iterable для нашого узагальненого сховища даних AbstractStore та створити ітератори для об’єктів цих класів. Ітератори дозволять послідовно обробляти об'єкти, що знаходяться у сховищі, у масиві arr.

Звичайно, це завдання можна було б вирішити, витягнувши із сховища копію масиву arr. Але вирішивши поставлену задачу ми отримаємо навички створення ітераторів. Окрім того за допомогою створеного ітератора ми отримаємо безпосередній доступ до даних сховища, а не до копії масиву arr.

6.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Закрийте усі класи з попередньої роботи, щоб уникнути плутанини.

Створіть новий проект для лабораторної роботи. Скопіюйте до теки src створеного проекту пакети з попередньої роботи. Після цього за допомогою рефакторингу поміняйте назви пакетів у відповідності з номером лабораторної роботи.

6.2.1 *Підключення інтерфейсу Iterable до класу AbstractStore*

Реалізація інтерфейсу Iterable у класі AbstractStore дозволить використовувати цикл for each, а також ітератор типу Iterator, безпосередньо для об’єктів класів ProductStore та WoodDirectory.

Відкриймо клас lab6.store.AbstractStore і додаймо до заголовку класу інструкцію, що вимагає реалізації інтерфейсу Itreable.

```
public class AbstractStore implements Iterable< Object>{
```

Після таких змін має з’явитися повідомлення про помилку. Для виправлення помилки слід викликати меню автоматичного виправлення помилок, зупинивши курсор на підкресленому слові. У меню, що з’явилося слід вибрати «Add unimplemented methods».

Як результат, у класі має з’явитися метод iterator(), наявності якого вимагає інтерфейс Iterable. Про місце його знаходження повідомляє синя цяточка праворуч панелі редактора.

6.2.2 *Реалізація методу iterator()*

Знайдіть заготовку методу iterator() у класі. Як бачимо, він має повертати об’єкт типу Iterator. Але Iterator – це інтерфейс, тому для створення об’єкту нам доведеться створити клас, що реалізує цей інтерфейс.

Припустимо, що такий клас вже є і має назву StoreIterator. У такому разі реалізація методу itrerator() може бути такою:

```
@Override
public Iterator<Object> iterator() {
     return new StoreIterator();
}
```

Але ж класу StoreIterator поки що нема, тому знову отримуємо повідомлення про помилку.

6.2.3 *Реалізація класу StoreIterator*

Клас StoreIterator почнемо також створювати шляхом автоматичного виправлення помилки. У меню виправлення помилки виберемо функцію “Create class StoreIterator”. Як наслідок отримуємо вже знайомий нам діалог, рисунок 6.1, де майже все вже прописано. Єдине, що залишається зробити, це увімкнути перемикач «Enclosing type» та вибрати радіоклавішу «private». Ці налаштування означають, що наш клас буде створено, як внутрішній приватний клас, у класі AbstractStore.

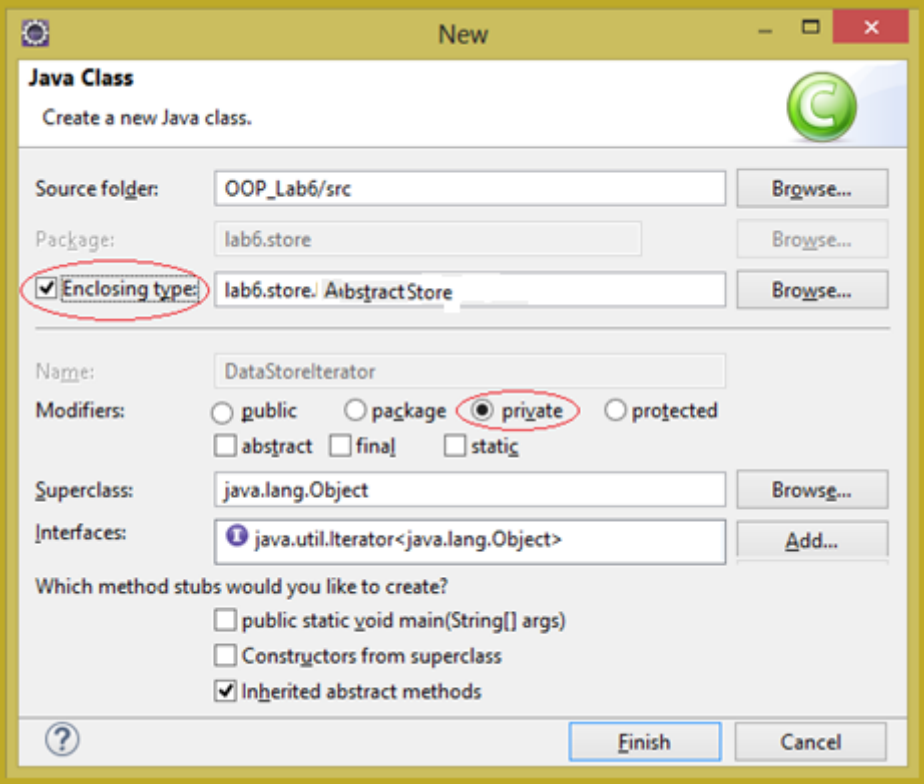

Рисунок 6.1 – Діалог створення класу StoreIterator

Натиснемо кнопку Finish і отримаємо заготовку внутрішнього класу StoreIterator у класі AbstractStore.

Залишається тільки вилучити непотрібне оголошення імпорту, яке Eclipse згенерував про всякий випадок, і повідомлень про помилки не буде.

Цей клас ми зробили внутрішнім не тільки заради демонстрації такої можливості, але й виходячи з того, що клас потрібен тільки у межах класу AbstractStore. Окрім того, у внутрішньому класі доступні усі поля основного класу, що стане нам у нагоді.

Ознайомтесь зі структурою створеного класу. Відповідно до вимог інтерфейсу Iterator він містить заготовки методів hasNext та next

Метод next має повертати поточний елемент масиву arr і підготуватися до повернення наступного. Одразу після створення ітератора перший елемент масиву з індексом 0 має бути поточним.

Метод hasNext має повертати true, якщо не всі елементи масиву повернуті. Після повернення останнього елементу метод має повертати false.

Для повторного використання ітератора його потрібно створити заново.

Перейдемо до реалізації внутрішнього класу StoreIterator.

Як параметр ітератора можна використовувати індекс поточного елементу, що повертається. Цей параметр визначимо як поле current внутрішнього класу.

Після доопрацювання внутрішній клас може виглядати так:

```
private class StoreIterator implements Iterator<Object> {

	// індекс поточного елементу масиву arr
	int current = 0;

	@Override
	public boolean hasNext() {
		// count - це кількість елементів у масиві
		return current < count;
	}

	@Override
	public Object next() {
		return arr[current++];
	}
}
```

6.2.4 *Тестування методів next та hasNext ітератора*

Щоб перевірити працездатність методів hasNext та next перепишемо метод toString класу AbstractStore, використовуючи цикл for each для сховища виробів. Для масиву цей метод використовувати було б некоректно, адже масив, у загальному випадку, заповнений не повністю.

```
public String toString() {
	StringBuilder sb = new StringBuilder();
	for (Object obj : this)
		sb.append(obj).append("\n");
	return sb.toString();
}
```

Посилання this у цьому методі є посиланням на об’єкт, що викликав даний метод. Тобто це може бути або каталог деревини, або сховище виробів. Таким чином маємо узагальнений метод, що працює для двох видів сховищ. Окрім того тут маємо ще другий рівень узагальнення завдяки використанню поліморфного методу toString, визначеного у класу Object, та перевизначеного у класах об’єктів деревина, брус, циліндр, мішок відходів. Ці методи викликає метод append.

Для перевірки нової реалізації методу слід запустити на виконання клас TestApp. Додаток має працювати так як і раніше.

6.2.5 *Реалізація методу remove для ітератора*

Відкрийте вихідний тест інтерфейсу java.util.Iterator і зафіксуйте його для звіту. Як можна побачити, інтерфейс має більш ніж два методи, але решта методів визначено як дефолтні. Зокрема, так реалізовано метод remove. Але ця реалізація тільки формує виключну ситуацію у разі спроби використання цього методу. Але ми можемо перевизначити цей метод у нашому класі таким чином, щоб він вилучав елемент, щойно виданий ітератором.

Перевизначимо цей метод.

Фактично, видалення елементу означає зсув елементів масиву на одну позицію ліворуч, починаючи з того, що розташований праворуч від того, що видаляється. Цю операцію можна виконати за допомогою методу arraycopy класу System.

Згадайте, або почитайте, як працює метод arraycopy. Після цього уважно проаналізуйте код методу remove, щоб мати можливість під час захисту пояснити, що ж тут написано:

```
@Override
public void remove() {
     System.arraycopy(arr, current,
               arr, current - 1,
               count-- -current--);
}
```

6.2.6 *Тестування методу ітератора remove*

Для тестування методу remove змінимо код методу start класу TestApp. Додаймо до цього методу такий код: (під час написання коду використовуйте автоматичну генерацію коду для циклу while).

```
System.out.println("Перелік виробів до вилучення");
System.out.println(ts.toString());

Iterator<Object> itr = ts.iterator();
while (itr.hasNext()) {
      IWeight obj = (IWeight) itr.next();
      if (obj.weight() > 1.3 ) itr.remove();
}

System.out.println("Перелік виробів після вилучення");
System.out.println(ts.toString());
```

У наведеному прикладі вилучаються вироби з вагою більше 1.3. У вашому застосунку ця цифра може бути іншою.

Об’єкти, що повертає ітератор потребують приведення до свого узагальненого типу.

Зверніть увагу, що метод remove викликається для ітератора, а не для виробу.

6.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Створити у класі AbstractStore ітератор типу java.util.ListIterator і метод listIterator, що його викликає. Клас для ітератора зробіть також внутрішнім і успадкуйте від класу StoreIterator. Реалізовувати усі методи необов’язково, але їх кількість впливає на результат.

Протестуйте ітератор у класі TestApp.

6.4 ВИМОГИ ДО ЗВІТУ

– Назва роботи.
– Мета роботи.
– Тексти класів, що були створені та модифіковані у роботі.
– Результати тестування застосування у вигляді копій консолі.
– Висновки.

6.5 КОНТРОЛЬНІ ПИТАННЯ

1. Що таке ітератори.
2. Інтерфейс Enumeration.
3. Інтерфейс Iterator.
4. Інтерфейс Iterable.
5. Інтерфейс ListIterator.
6. Особливості внутрішніх класів.

7 ЛАБОРАТОРНА РОБОТА №7. ПЕРЕДАЧА СПЕЦИФІЧНИХ ОПЕРАЦІЙ КОРИСТУВАЧА У МЕТОДИ

Мета роботи:

– Навчитися передавати до узагальнених методів специфічні операції користувача.
– Реалізувати узагальнені методи для вибіркової обробки даних.
– Познайомитися з лямбда функціями.

7.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

На останньому етапі попередньої лабораторної роботи було написано код, що видаляє об’єкти зі сховища, у яких вага перевищує задану. Але у інших випадках може виникнути потреба вилучати об’єкти, які відповідають зовсім іншим умовам. При цьому алгоритм вилучення не змінюється, змінюється тільки умова, яку потрібно перевіряти перед вилученням.

Під час даної лабораторної роботи ми маємо створити узагальнений метод вилучення об’єктів із сховища, до якого, в якості параметра, буде передаватися умова вилучення. Тобто до методу мають передаватися якісь операції, що користувач методу вважає за потрібне.

В C ++ для вирішення такого завдання використовують покажчики на функцію, що передають у метод як параметр. В Smalltalk можна було передати в якості параметру код обробки у вигляді об'єкта класу Block. В Pascal така проблема вирішувалася за допомогою процедурного типу даних.

В Java ми можемо передати до методу тільки об’єкт, а вже через цей об’єкт у методі можна викликати потрібну операцію.

Проблема створення об’єктів, через які можна викликати потрібні операції виявилася настільки актуальною, що у Java 8 з’явилося поняття функціональний інтерфейс і визначена низка таких інтерфейсів, з якими ми маємо познайомитися.

7.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Закрийте усі класи з попередньої роботи, щоб уникнути плутанини.

Створіть новий проект для лабораторної роботи та не забувайте про пакети.

Скопіюйте до створених пакетів класи з попередньої роботи.

7.2.1 *Знайомство з функціональними інтерфейсами*

Зайдіть до системної бібліотеки та знайдіть у файлі rt.jar пакет java.util.function. У пакеті визначена велика кількість функціональних інтерфейсів. Познайомтесь з інтерфейсами Function, Consumer, BooleanSupplier.

Для вирішення поставленого завдання нам знадобиться інтерфейс Predicate. Цей інтерфейс визначає метод test, що приймає в якості параметру якийсь об’єкт і повертає true або false. То ж ми можемо створити метод remove,

до якого будемо передавати об’єкт типу Predicate. У самому методі для кожного об’єкту сховища викликати метод test, і якщо він повертає true, вилучати цей об’єкт.

7.2.2 *Реалізація методу remove у класі lab7.store.AbstractStore*

Реалізуймо у класі AbstractStore метод remove. Для послідовного доступу до даних скористуймося ітератором. Це зробить метод більш універсальним, він буде працювати і тому випадку, якщо ми змінимо структуру, яка зберігає дані у сховищі. А для передачі умови видалення скористаймося об’єктом типу Predicate.

```
public void remove(Predicate<Object> prd){
     Iterator<Object> itr = this.iterator();
     while (itr.hasNext()) {
          Object obj = (Object) itr.next();
          if(prd.test(obj)){
               itr.remove();
          }
     }
}
```

Метод можна викликати і для каталогу деревини і для сховища виробів. В умові можна використовувати будь які доступні параметри виробів.

7.2.3 *Використання методу remove*

Проблема використання методу remove полягає у створенні об’єкту типу Predicate. Найбільш довгий і, мабуть, найменш ефективний спосіб полягає у створенні класу, що реалізує інтерфейс Predicate, з подальшим створенням об’єкту. Цей шлях ми розглядати не будемо.

Розглянемо інші шляхи створення такого об’єкту для прикладу, коли треба вилучати мішки з відходами вагою більш ніж maxWeight.

7.2.3.1 **Створення об’єкту з використанням анонімного класу**

Із використанням анонімних класів ми вже зустрічалися раніше. Перевага цього підходу полягає у тому, що клас існує тільки у момент створення об’єкту, і не засмічує програму. Клас не має назви, а для створення об’єкту використовується ім’я інтерфейсу.

Наберіть неповний рядок:

```
Predicate<Object> prd = new Predi
```

Після цього натисніть клавіши Ctrl+Пробіл і виберіть у меню, що з’явилося, рядок «Predicate() Anonymous Inner Type – java.utile.function».

Eclipse має доповнити рядок до такого вигляду, сформувавши пусте тіло анонімного класу:

```
Predicate<Object> prd = new Predicate<Object>() {
};
```

Але внаслідок того, що у тілі класу не визначена функція, ми знову отримуємо повідомлення про помилку. Щоб її виправити, слід підвести курсор до підкресленого слова і дочекатися появи меню виправлення помилок. Далі вибрати «Add unimplemented methods».

Eclipse доповнить тіло методу заготовкою методу, що потрібно реалізувати:

```
Predicate<Object> prd = new Predicate<Object>() {
    @Override
    public boolean test(Object t) {
        // TODO Auto-generated method stub
        return false;
    }
};
```

Тепер ми маємо шаблон, до якого треба додати потрібну нам перевірку, на що вказує коментар //TODO.

Для того, щоб з'ясувати, чи належить продукт до класу Waste, використовуємо інструкцію instanceof, а щоб отримати доступ до методу weight(), приводимо об'єкт до типу IWeight.

Після доопрацювання цього фрагменту коду можна викликати метод remove:

```
float maxWeight = 50f;
Predicate<Object> prd = new Predicate<Object>() {
    @Override
    public boolean test(Object t) {
        return t instanceof Waste
                && ((IWeight) t).weight() > maxWeight;
    }
};
ps.remove(prd );
```

7.2.3.2 **Виклик методу remove з використанням анонімного об'єкту**

У програмуванні на Java є також дуже поширеним використання анонімних об'єктів. У цьому випадку конструкція, за допомогою якої створюється об'єкт, передається у метод як параметр, без визначення назви об'єкту.

```
float maxWeight = 50f;
ps.remove(new Predicate<Object>() {
    @Override
    public boolean test(Object t) {
        return t instanceof Waste
                && ((IWeight) t).weight() > maxWeight;
    }
});
```

7.2.3.3 **Функціональні інтерфейси та лямбда функції**

Функціональний інтерфейс визначає тільки один метод, що потребує реалізації. Внаслідок цього назва цього методу однозначно пов'язана з назвою інтерфейсу. Більш того, коли у якомусь методі параметр має тип функціонального інтерфейсу, то під час компіляції виклику цього методу відомі і тип об'єкту, і сигнатура методу, що має бути викликаний для цього об'єкту.

Ці дві особливості дозволили спростити створення анонімних об'єктів анонімних класів, що реалізують функціональні інтерфейси.

Анонімний об'єкт анонімного класу, що реалізує функціональний інтерфейс можна створювати у вигляді лямбда функції.

Лямбда функція складається з трьох частин.

Перша частина – це круглі дужки, у яких наводиться перелік параметрів функції, якщо вони є. Типи не вказуються, бо вони визначені у інтерфейсі. Імена параметрів довільні.

Друга частина – це комбінація з двох символів ->, що є ознакою лямбда.

Третя частина являє собою тіло методу функціонального інтерфейсу. Якщо функція складається з декількох операторів, то потрібні фігурні дужки і оператор return. Функція може і не використовувати параметри.

Лямбда функція спрощує створення анонімного об'єкту за рахунок того, що не використовує ні назву методу, ні назву інтерфейсу з конструкцією new. Найбільший ефект використання лямбда функції дає для реалізації коротких методів.

Так, наприклад, виклик методу remove, що розглядався у попередньому пункті може бути записаний таким чином:

```
float maxWeight = 50f;
ps.remove((t)-> t instanceof Waste
                && ((IWeight) t).weight() > maxWeight);
```

7.2.4 *Приклад використання інтерфейсу Consumer*

Інтерфейс Consumer визначає метод accept, що приймає на обробку дані і нічого не повертає. У методі accept програміст просто визначає, що потрібно зробити з отриманими даними.

Як приклад використання цього інтерфейсу створімо к класі AbstractStore метод doForAll:

```
public void doForAll(Consumer<Object> cns) {
     Iterator<Object> itr = this.iterator();
     while (itr.hasNext()) {
          Object obj = (Object) itr.next();
          cns.accept(obj);
     }
}
```

У цьому методі за допомогою ітератора ми послідовно отримуємо елементи сховища і викликаємо для їх обробки метод accept об'єкту типу Consumer. Обробка може бути будь якою, це залежить від об'єкту, що

передається як параметр.

За допомогою цього методу можна, наприклад, роздрукувати вміст каталогу деревини:

```
wd.doForAll((t) -> System.out.println(t));
```

Останній вираз можна ще більше спростити, якщо скористатися посиланням на метод, що можна використовувати в лямбда.

```
wd.doForAll( System.out::println);
```

У цьому прикладі вираз System.out::println є посиланням на метод println об’єкту out класу System. Це посилання еквівалентне лямбда виразу, що використовується у попередньому прикладі. Дійсно, адже назва змінної не має значення. Компілятору потрібно повідомити тільки який метод ми хочемо використовувати. Решта визначено інтерфейсом Consumer.

7.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Створити у класі AbstractStore метод doOnlyFor, що буде виконувати якусь обробку частини об’єктів, що відповідають заданій умові.

Протестуйте створений метод у класі TestApp.

Додайте виклик цього тесту та тестів, що були виконані під час роботи, до меню свого консольного застосування.

7.4 ВИМОГИ ДО ЗВІТУ

– Назва роботи.
– Мета роботи.
– Тексти методів, що були створені та модифіковані у роботі.
– Результати тестування застосування у вигляді копій консолі.
– Висновки.

7.5 КОНТРОЛЬНІ ПИТАННЯ

1. Що таке «передача специфічних операцій користувача у методи» і як ця проблема вирішується у різних мовах програмування.
2. Що таке функціональний інтерфейс. Приклади таких інтерфейсів.
3. Що таке дефолтні методи.
4. Що таке анонімний клас та приклад його використання.
5. Що таке анонімний об’єкт та приклад його використання.
6. Що таке лямбда функція та приклад його використання.
7. Використання посилань на функції у лямбда виразах.
8. Написати універсальний метод табулювання функції однієї змінної, використовуючи функціональний інтерфейс Function

8 ЛАБОРАТОРНА РОБОТА № 8.
КОЛЕКЦІЇ JAVA

Мета роботи:

- Познайомитися з поняттям колекція.
- Познайомитися з інтерфейсами колекцій.
- Познайомитися з класами, в яких реалізовані інтерфейси колекцій.

8.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Дослідити можливості роботи з колекціями Java.

Використати колекцію для сховища продуктів у раніше створеному додатку.

8.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Закрийте усі класи з попередньої роботи, щоб уникнути плутанини.

Створіть новий проект для лабораторної роботи.

Скопіюйте до проекту пакетів класи з попередньої роботи та перейменуйте їх у відповідності до номеру роботи.

Створіть у пакеті lab8.test клас TestCollections з методом main. У цьому класі будемо досліджувати можливості роботи з колекціями.

8.2.1 *Дослідження інтерфейсу Collection*

Інтерфейс Collection визначає перелік основних методів роботи з колекціями, які не пов’язані з особливостями внутрішньої побудови колекції. З повним переліком цих методів можна ознайомитися у вихідному тексті інтерфейсу.

Відкрийте вихідний текст інтерфейсу і скопіюйте до звіту перелік методів інтерфейсу із панелі Outline. Можна виділити всі методи та вибрати “Copy Qualified Name”. На захисті лабораторної роботи ви маєте пояснити призначення кожного з методів.

8.2.1.1 **Реалізації інтерфейсу Collection**

Інтерфейс Collection реалізують усі класи колекцій. Найбільш відомі класи Vector, Stack, ArrayList, LinkedList, ArrayDeque, PriorityQueue, HashSet, LinkedHashSet, TreeSet.

Ви маєте знати назви цих класів і мати уяву про внутрішню організацію даних у цих класах.

Ці класи використовують різні структури даних для збереження елементів колекції і, відповідно, різну реалізацію методів інтерфейсу. Це прояв поліморфізму. Визначивши для якогось сховища даних тип Collection можна далі писати код для роботи з цим сховищем, незалежно від внутрішньої

організації цього сховища. У разі потреби реалізацію колекції можна змінити, вибравши інший клас, не змінюючи код обробки.

8.2.1.2 **Експерименти з колекціями**

Створімо у методі main декілька колекцій та роздрукуємо їх зміст. Але перш ніж друкувати цей текст, зверніть увагу на особливості роботи з колекціями.

Створюючи колекцію, ми можемо не визначати обсяг пам'яті для даних, колекція сама потурбується про це.

Колекції можуть містити тільки об'єкти. Якщо до колекцій додаються дані базових типів, то для цих даних автоматично виконується «автобоксинг», тобто дані перетворюються у об'єкти класів «обгорток».

До колекції можна додавати елементи і по одному і групами.

Для колекції можна викликати як звичайний for each, так і метод foreEach з лямбда функцією, що реалізує інтерфейс Consumer.

Тепер наберіть і виконайте цей код.

```
Random rnd = new Random();

Collection<Integer> c1 = new Vector<>();
for (int i = 0; i < 15; i++) {
     c1.add(rnd.nextInt(10));
}

System.out.println("Collection Vector");
for(Integer x : c1)
     System.out.printf("%d ",x);

Collection<Integer> c2 = new TreeSet<>();
c2.addAll(c1);
System.out.println("\nCollection TreeSet");
c2.forEach((x)->System.out.printf("%d ",x));
```

Зверніть увагу на різницю у результатах. Поясніть, чому так.

Протестуйте усі згадані вище класи колекцій і проаналізуйте результати.

8.2.1.3 **Експерименти з груповими операціями над колекціями**

Наберіть код, що наведено нижче, і виконайте його, після чого проаналізуйте результати. Ви маєте пояснити під час захисту, як працюють ці методи.

```
Random rnd = new Random();

Collection<Integer> c1 = new ArrayList<>();
Collection<Integer> c2 = new LinkedHashSet<>();
Collection<Integer> c3 = new ArrayList<>();
for (int i = 0; i < 10; i++) {
     c1.add(rnd.nextInt(10));
     c2.add(rnd.nextInt(10));
}
c3.addAll(c1); c3.removeAll(c2);
System.out.println(c1 + "removeAll" + c2 + " = " + c3);

c3.clear(); c3.addAll(c2); c3.removeAll(c1);
System.out.println(c2+ "removeAll" + c1 + " = " + c3);

c3.clear(); c3.addAll(c1); c3.retainAll(c2);
System.out.println(c1+ "retainAll" + c2 + " = " + c3);

c3.clear(); c3.addAll(c2); c3.retainAll(c1);
System.out.println(c2+ "retainAll" + c1 + " = "+c3);
```

Те, яким чином у наведеному коді ми виводимо зміст колекцій, свідчить про те, що десь перевизначено метод toString() для колекцій. Знайдіть, де це зроблено.

Протестуйте наступний код та зробіть висновки відносно методу containAll:

```
Random rnd = new Random();

Collection<Integer> c1 = new ArrayList<>();
for (int i = 0; i < 10; i++) {
     c1.add(rnd.nextInt(10));
}

Collection<Integer> c2 = new LinkedHashSet<>();
c2.addAll(c1);
boolean b1 = c1.containsAll(c2);
System.out.println(c1 + "containsAll" + c2 + " = " + b1);

Collection<Integer> c3 = new TreeSet<>();
c3.addAll(c1);
boolean b2 = c1.containsAll(c3);
System.out.println(c1 + "containsAll" + c3 + " = " + b2);
```

8.2.2 *Дослідження інтерфейсу List*

Інтерфейс List успадковує і розширює інтерфейс Collection.

Головна особливість цього інтерфейсу – це підтримка поняття індексу. Тобто кожен елемент відповідних колекцій має свій номер, також існують

поняття попередній і наступний. Номери елементів починаються з нуля, як і в масивах, але інтерфейс не гарантує, що колекція реалізована на масиві.

З переліком методів цього інтерфейсу можна ознайомитися у вихідному тексті інтерфейсу. Відкрийте вихідний текст інтерфейсу і скопіюйте до звіту перелік методів інтерфейсу із панелі Outline. Після цього вилучіть із списку методи інтерфейсу Collection, що успадковуються. На захисті лабораторної роботи ви маєте пояснити призначення кожного з методів.

8.2.2.1 **Реалізації інтерфейсу List**

Інтерфейс List реалізовано в уже знайомих Вам класах Vector, Stack, ArrayList, LinkedList.

8.2.2.2 **Дослідження методів інтерфейсу List**

Самостійно реалізуйте приклади на використання методів інтерфейсу List.

Як приклад, розглянемо метод sort(). До цього методу треба передати об’єкт, що реалізує функціональний інтерфейс Comparator. Цей інтерфейс передбачає реалізацію методу compare, що має порівнювати два об’єкти і повертати ціле число. Якщо об’єкти однакові має повертатися нуль. Якщо перший менший за другого, має повертатися від’ємне число, якщо перше більше – то має повертатися додатне число.

```
Random rnd = new Random();

List<Integer> c1 = new ArrayList<>();
for (int i = 0; i < 10; i++) {
     c1.add(rnd.nextInt(10));
}

c1.sort(new Comparator<Integer>() {

     @Override
     public int compare(Integer a, Integer b) {
          return a - b;
     }
});
```

У наведеному прикладі до методу sort передається анонімний об’єкт типу Comparator. Приклад швидкого створення такого об’єкту розглядався у попередній роботі. Такий шлях створення об’єкту вибрано для того, щоб можна було побачити метод compare.

Значно простіше ця частина коду буде виглядати, якщо скористатися лямбда функцією:

```
c1.sort((a, b) -> a - b);
```

8.2.3 *Дослідження класу Collections*

Цей утилітний клас схожий на клас Arrays. У ньому реалізовано низку

операцій, що пов’язані з обробкою колекцій.

Статичні методи цього класу можна поділити на дві групи.

Методи першої групи підходять для будь яких колекцій.

Методи другої групи можна використовувати тільки для колекцій типу List.

8.2.3.1 **Дослідження універсальних методів класу Collections**

Нижче наведені приклади використання деяких універсальних методів класу Collections. Ваше завдання набрати і виконати ці приклади, а до звіту додати результати виконання разом з кодом, що виконувався.

```
Collection<Integer> coll = new ArrayDeque<>();
//Methods for Collection
Collections.addAll(coll, 1, 3, 5, 3 , 4, 2,14);
Collections.addAll(coll, new Integer[]{3,7,12});
System.out.println(coll);
System.out.println(Collections.frequency(coll,3));
System.out.println(Collections.max(coll));
System.out.println(Collections.min(coll));
```

8.2.3.2 **Дослідження методів класу Collections для колекцій List**

Опрацюйте наведені нижче приклади так само, як і у попередньому пункті.

```
// Methods for List
List<Integer> list = new ArrayList<>(coll);
Collections.swap(list, 2, 6);
System.out.println(list);
Collections.sort(list);
System.out.println(list);
System.out.println(Collections.binarySearch(list,3));//4
Collections.sort(list,(a,b)->b-a);
System.out.println(list);
System.out.println(Collections.binarySearch(list,3,(a,b)->b-a));
Collections.reverse(list);
System.out.println(list);
Collections.rotate(list,2);
System.out.println(list);
Collections.shuffle(list);
System.out.println(list);
Collections.replaceAll(list, 3,8);
System.out.println(list);
Collections.fill(list,0);
System.out.println(list);
```

8.2.4 *Дослідження класу Stack*

Перегляньте вихідний текст класу Stack і зафіксуйте у звіти методи, що реалізовані у цьому класі додатково до інтерфейсу List. Додайте також пояснення до цих методів.

8.2.5 *Дослідження інтерфейсу Queue*

Перегляньте інтерфейс Queue. Зафіксуйте його методи у звіті, та надайте пояснення до них.

Цей інтерфейс реалізовано у класах LinkedList, ArrayDeque, PriorityQueue. Перегляньте коментарі до цих класів та поясніть у звіті різницю між ними.

8.2.6 *Дослідження інтерфейсу Deque*

Цей інтерфейс успадковує інтерфейс Queue і його реалізовано у класі ArrayDeque. Перегляньте методи цього інтерфейсу і дайте їм коротку характеристику у звіті.

8.2.7 *Дослідження інтерфейсу Set*

Цей інтерфейс успадковує інтерфейс Collection і його реалізовано у класі HashSet, LinkedHashSet та TreeSet. Перегляньте методи цього інтерфейсу і з'ясуйте, які додаткові можливості надає цей інтерфейс.

8.2.7.1 **Методи equals() и hashCode()**

Класи, що реалізують інтерфейс Set мають гарантувати унікальність об'єктів у колекції.

Щоб познайомитися з поняттям унікальності об'єкту проведемо експеримент, протестувавши такий фрагмент тексту:

```
Set<Wood> set = new HashSet<>();
set.add(new Wood(1,"Lypa",1f));
set.add(new Wood(1,"Lypa",1f));
set.add(new Wood(1,"Lypa",1f));

set.forEach(System.out::println);
```

Виявляється, що колекція приймає абсолютно однакові об'єкти. Справа у тому, що з точки зору Java тут три зовсім різні об'єкти. Для перевірки ідентичності об'єктів Java використовує метод equals класу Object. У цьому методі порівнюються посилання на об'єкти через операцію ==. Кожен із наших об'єктів має своє унікальне місце в пам'яті, тож посилання на ці об'єкти мають різні значення і тому об'єкти різні.

Якщо ж ми хочемо порівнювати об'єкти за їх вмістом, то потрібно у класі перевизначити метод equals з урахуванням наших понять про унікальність. Але при цьому слід перевизначать і метод hashCode (), бо якщо об'єкти однакові то і хеш коди для них також мають бути однакові.

Для підвищення надійності створення цих методів Eclipse пропонує автоматичне створення цих методів через меню Source →Generate hashCode() and equals().

Реалізуйте у класі Wood ці методи з урахуванням своїх уявлень про ідентичність і повторно виконайте код прикладу. Методи hashCode() та equals() наведіть у звіті та вмійте прокоментувати текст на захисті.

8.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Для цієї роботи завдання для самостійної роботи відсутнє.

8.4 ВИМОГИ ДО ЗВІТУ

– Назва роботи.
– Мета роботи.
– Тексти прикладів, що були досліджені під час роботи з результатами виконання.
– Методи досліджуваних інтерфейсів і класів з поясненнями.
– Методи equals та hashCode класу Wood.
– Висновки.

8.5 КОНТРОЛЬНІ ПИТАННЯ

1. Дати пояснення до текстів у звіті.
2. Навести перелік інтерфейсів колекцій та охарактеризувати їх.
3. Навести перелік класів, що реалізують той чи інший інтерфейс і охарактеризувати ці класи.
4. Методи для порівняння об’єктів.

9 ЛАБОРАТОРНА РОБОТА № 9. АСОЦІАТИВНІ МАСИВИ JAVA

Мета роботи:
- Познайомитися з поняттям асоціативний масив.
- Познайомитися з інтерфейсом Map.
- Познайомитися з класами, в яких реалізовані інтерфейс Map.
- Познайомитися з поняттям однаковості об'єктів.
- Реалізувати сховища даних на колекціях

9.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Дослідити класи, що реалізують інтерфейс Map.

Використати колекції для сховища продуктів у раніше створеному додатку.

9.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Створімо проект для лабораторної роботи та пакети lab9.model, lab9.store, lab9.test. Скопіюємо до цих пакетів класи з відповідних пакетів попередньої лабораторної роботи.

9.2.1 *Дослідження інтерфейсу Map*

У пакеті lab9.test створімо клас TestMaps з методом main. У цьому класі будемо досліджувати можливості роботи з колекціями.

9.2.1.1 **Тестування методів інтерфейсу Map**

Виконайте наведений нижче код і проаналізуйте результати

```
Map<String, String> map = new HashMap<>();

map.put("file","файл");
map.put("block","блок");
map.put("dictionary","словник");

map.keySet().forEach(System.out::println);
map.values().forEach(System.out::println);
map.entrySet().forEach(System.out::println);
```

Відкрийте вихідний текст інтерфейсу Map. Проаналізуйте перелік методів. Протестуйте ще декілька методів і наведіть код і результати у звіті.

Зверніть увагу, що інтерфейс Map має ще внутрішній інтерфейс. Наведіть у звіті його назву і призначення.

9.2.1.2 **Дослідження різниці у порядку збереження даних**

Виконайте наведений нижче код і зафіксуйте результат.

```
Map<String, String> map = new HashMap<>();

map.put("file","файл");
map.put("block","блок");
map.put("dictionary","словник");

map.keySet().forEach(System.out::println);
```

Замініть реалізацію інтерфейсу з класу HashMap на LinkedHashMap та зафіксуйте результат.

Замініть реалізацію інтерфейсу з класу LinkedHashMap на TreeMap та зафіксуйте результат.

Поясніть у звіті різницю в результатах.

9.2.2 *Дослідження класу Properties*

У додатках Java досить часто використовується асоціативний масив типу Properties, що містить у собі системну інформвцію. Доступ до цієї карти можна отримати через клас System.

Наберіть наведений нижче код, та ознайомтеся зі змістом цієї карти.

```
public static void main(String[] args) {
      Properties props = System.getProperties();
      props.forEach((key, value) ->
                              System.out.println(key + "->"+value));
 }
```

До звіту додайте значення для таких ключів: java.version, java.home, user.dir, user.name, os.version.

9.2.3 *Реалізація сховищ даних на колекціях і картах*

У попередніх проектах ми використовували масив arr і для каталогу деревини і для сховища продуктів. Але з точки зору прикладного програмування це не зовсім правильно, бо вимагає вирішення багатьох допоміжних проблем. Зокрема, нам доводилося стежити за розміром масиву і розширювати його у разі потреби. Потрібно було також реалізовувати ітератори.

Програмуючи на Java доцільно для вирішення задач накопичення даних використовувати колекції або карти.

У нашому конкретному випадку для створення каталогу деревини найбільше підходить клас HashMap, де в якості ключа будемо використовувати ідентифікаційний номер. Це дозволить дуже швидко вибирати із каталогу будь який об’єкт.

Для збереження продуктів можна використовувати звичайний список, наприклад, LinkedList.

Такі рішення призведуть до корінних змін у класах пакету lab9.store.

9.2.3.1 **Модифікація класу WoodDirectory**

Заголовок класу не змінюємо, а тіло набуває такого вигляду:

```
private Map<Integer, Object> map = new HashMap<>();

{
	map.put(1, new Wood(1, "Модрина", 1.1f));
	map.put(2, new Wood(2, "Ялина", 0.9f));
	map.put(3, new Wood(3, "Сосна", 0.7f));
}

public Wood get(int id){
	return (Wood) map.get(id);
}

public boolean add(Wood newWood) {
	if(map.get(newWood.getId()) != null)
		return false;
	map.put(newWood.getId(), newWood);
	return true;
}
```

Головна зміна полягає у тому, що карта для каталогу прописана вже в цьому класі, а не у суперкласі. Але це призведе до того, що клас доведеться ще доопрацьовувати, бо втрачена універсальність.

9.2.3.2 **Модифікація класу ProductStore**

Заголовок класу теж не змінюємо, а тіло набуває такого вигляду:

```
private List<Object> list = new ArrayList<>();

public boolean add(IWeight newTimber) {
	list.add(newTimber);
	return true;
}
```

Цей клас змінився менше. Головне – це поява сховища для продуктів, і тип цього сховища зовсім інший, ніж у класі WoodDirectory. То ж втрачаємо універсальність і клас доведеться доопрацьовувати.

9.2.3.3 **Модифікація класу AbstractStore**

Виходячи і змін, що відбулися у попередніх класах перш за все вилучимо змінні arr та count, що відповідали за збереження даних, та універсальний метод, який вже не використовується у класах спадкоємцях.

Вилучимо також внутрішні класи ітераторів та методи iterator і listIterator, бо у колекціях ітератори реалізовані.

Після цього будемо виправляти помилки, що з’явилися у класі.

9.2.3.4 **Реалізація інтерфейсу Iterable**

У класі AbstractStore заявлено реалізацію інтерфейсу Iterable. Але, вилучивши внутрішні класи і метод iterator ми ліквідували реалізацію інтерфейсу Iterable. Більш того, у цьому класі реалізація ітератора неможлива,

бо маємо зовсім різні сховища. Тому зробимо клас абстрактним, а методи для виклику ітераторів реалізуємо у підкласах.

У класі ProductStore заготовку методу iterator можна отримати шляхом автоматичного виправлення помилок і потім доопрацювати його до такого вигляду:

```
@Override
public Iterator<Object> iterator() {

     return list.iterator();
}
```

Можна також реалізувати метод listIterator:

```
public ListIterator<Object>listIterator() {

     return list.listIterator();
}
```

У класі WoodDirectory в якості ітератора повернемо ітератор множини значень карти:

```
@Override
public Iterator<Object> iterator() {
     return map.values().iterator();
}
```

9.2.3.5 **Реалізація методу getCount()**

Для виправлення помилки метод getCount() у класі AbstractStore слід зробити абстрактним:

```
public abstract int getCount();
```

Реалізація цього методу у класі ProductStore буде виглядати так:

```
@Override
public int getCount() {
     return list.size();
}
```

Схожою буде реалізація і у класі WoodDirectory:

```
@Override
public int getCount() {
     return map.size();
}
```

9.2.3.6 **Реалізація методу getArr()**

Для виправлення помилки метод getArr() у класі AbstractStore теж зробимо абстрактним:

```
public abstract Object[] getArr();
```

Реалізація цього методу у класі ProductStore буде виглядати так:

```
@Override
public Object[] getArr() {
      return list.toArray();
}
```

Схожою буде реалізація і у класі WoodDirectory:

```
@Override
public Object[] getArr() {
      return map.values().toArray();
}
```

9.2.3.7 **Тестування модифікованого додатку**

Зміна моделі не повинна була вплинути на роботу методу start класу TestApp. Додаток має працювати так само, як і у сьомій роботі.

9.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Протестувати консольний додаток, що було створено раніше, на сумісність з модифікованими класами.

9.4 ВИМОГИ ДО ЗВІТУ

- Назва роботи.
- Мета роботи.
- Тексти прикладів, що були досліджені під час роботи з результатами виконання.
- Тексти модифікованих класів.
- Висновки.

9.5 КОНТРОЛЬНІ ПИТАННЯ

1. Дати пояснення до текстів у звіті.
2. Які інтерфейси розглядалися у роботі? Методи цих інтерфейсів.
3. Навести перелік класів, що реалізують інтерфейс Map і дати характеристику цим класам.
4. Пояснити різницю між модифікованими класами та їх попередньою реалізацією. У чому переваги і недоліки цих варіантів.

10 ЛАБОРАТОРНА РОБОТА № 10. ПОТОКИ ВИКОНАННЯ КОДУ

Мета роботи:

- Познайомитися з поняттям потоку виконання коду.
- Засвоїти основи роботи з потоками виконання коду в Java.
- Познайомитися з механізми синхронізації потоків Java.
- Познайомитися з потокобезпечними колекціями.

10.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Дослідити способи створення потоків виконання коду.
Дослідити способи тимчасової зупинки потоків.
Дослідити потокобезпечні колекції

10.2 ПОРЯДОК ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Створімо новий проект для лабораторної роботи та скопіюємо до нього пакети з попередньої лабораторної роботи та перейменуємо їх у lab10.model, lab10.store, lab10.test. Створімо ще пакети lab10.threads.

У пакеті lab10.test клас TestMap можна вилучити, він нам не знадобиться.

10.2.1 *Створення потоків шляхом успадкування класу Thread*

Створімо у пакеті lab10.threads клас TimberShop, що успадковує клас Thread. Цей клас будемо використовувати як постачальника заданої кількості брусів з випадковими параметрами. Визначимо у цьому класі поля для каталогу деревини (wd), сховища продуктів (ps), об'єкт класу Radom для генерації випадкових параметрів продукту (rnd) та поле для кількості брусів (n), що має виготовити об'єкт класу, що ми створюємо.

Згенеруємо також конструктор з полями та створімо метод createProduct, що буде повертати брус з випадковими параметрами.

Створений клас може виглядати так:

```
public class TimberShop extends Thread {
	WoodDirectory wd;
	ProductStore ps;
	Random rnd = new Random();
	int n;

	public TimberShop(WoodDirectory wd, ProductStore s, int n) {
		super();
		this.wd = wd;
		this.ps = s;
		this.n = n;
	}
```

```
        IWeight createProduct() {
                int woodId = rnd.nextInt(3) + 1;
                Wood wood = wd.get(woodId);
                float length = 1 + rnd.nextFloat() * 10;
                float height = 0.1f + rnd.nextFloat();
                float width = 0.1f + rnd.nextFloat();
                Timber timber = new Timber(wood, length, height, width);
                return timber;
        }
}
```

Ще ми зробимо так, щоб на виготовлення бруса витрачався деякий час. Для цього будемо використовувати виклик методу підрахунку числа Фібоначчі рекурсивним способом, який для чисел з номерами більше 40 потребує помітного часу. Створімо цей метод у класі TimberShop:

```
int fibo(int n) {
        if (n < 2)
                return 1;
        return fibo(n - 1) + fibo(n - 2);
}
```

Після реалізації цих підготовчих кроків реалізуймо у класі метод run з кодом, що буде виконуватися у потоці та імітувати процес виробництва брусів.

```
public void run() {

        for (int i = 0; i < n; i++) {
                fibo(40 + rnd.nextInt(5));
                IWeight timber = createProduct();
                System.out.println(this.getName() + " create " + timber);
        }

}
```

10.2.2 *Тестування класу TimberShop*

Для тестування класу у пакеті lab10.test модифікуємо клас TestApp, з методом main. Залишимо у класі поля для каталогу деревини та сховища для брусів.

```
// Каталог деревини
private WoodDirectory wd = new WoodDirectory();
// Каталог для брусів
private ProductStore ps = new ProductStore();
```

Метод main залишається таким самим.

Метод startApp реалізуємо таким чином:

```
private void startApp() {
        Thread tshop = new TimberShop(wd, ps,3);
        tshop.start();
}
```

Зверніть увагу, потік активізується методом start класу Thread.

Після запуску додатку отримаємо через деякий на консолі інформацію про створені бруси.

Тепер створімо у методі startApp декілька виробників брусів і активізуємо відповідні потоки:

```
private void startApp() {
        Thread tshop1 = new TimberShop(wd, ps ,3);
        Thread tshop2 = new TimberShop(wd, ps ,3);
        Thread tshop3 = new TimberShop(wd, ps ,3);
        tshop1.start();
        tshop2.start();
        tshop3.start();
}
```

Результати виконання свідчать про те, що виробники брусів працюють не послідовно, а паралельно. За наявності декількох процесорів це виконання буде дійсно паралельним. Якщо ж процесор тільки один, то буде псевдопаралельне виконання, тобто операційна система буде виділяти потокам по черзі кванти системного часу.

10.2.3 *Аналіз недоліків спадкування класу Thread*

Далі нам потрібно буде створити клас CylinderShop, який буде у своєму потоці формувати круглі бруси. Тобто ми будемо моделювати одночасну роботу декількох цехів.

Клас CylinderShop буде дуже схожий на клас TimberShop, бо буде відрізнятися тільки методом createProduct. То ж виникає бажання виділити з класу TimberShop суперклас, щоб спростити створення класу CylinderShop за рахунок спадкування. Але цього ми зробити не зможемо, тому що клас TimberShop вже успадковує клас Thread, а множинне спадкування класів у Java неможливе.

Саме у цьому і полягає недолік створення потоків шляхом успадкування класу Thread.

Але проблема, що виникла, просто вирішується, якщо потоки створювати на основі інтерфейсу Runnable.

10.2.4 *Створення потоків з використанням інтерфейсу Runnable*

Реалізація інтерфейсу Runnable можлива у будь якому класі з будь якої ієрархії. Інтерфейс передбачає реалізацію метод run(), що містить код, що буде виконуватися у потоці. Для запуску потоку потрібно створити об’єкт типу Thread через конструктор з параметром типу Runnable. Після цього викликати метод start для потоку. У потоці почне виконуватися метод run об’єкту Runnable.

Для прикладу модіфікуємо вже створений клас TimberShop.

Замість спадкування класу Thread заявимо реалізацію інтерфейсу Runnable.

```
public class TimberShop implements Runnable {
```

Додаймо до класу ще поле name і метод доступу до нього. Це поле будемо використовувати для ідентифікації потоку. Використовувати метод getName потоку, як раніше, ми вже не зможемо, бо наш клас не знає у якому потоці він буде виконувати свій метод run. Це один з недоліків використання інтерфейсу Runnable.

```
String name;

public String getName() {
      return name;
}
```

Доведеться також доопрацювати конструктор класу, додавши до переліку параметрів ім'я потоку.

```
public TimberShop(String name, WoodDirectory wd, ProductStore s, int n) {
            super();
            this.name = name;
            this.wd = wd;
            this.ps = s;
            this.n = n;
}
```

Після цього можна із модифікованого класу виділити суперклас WoodShop, куди перенести усі поля і методи, окрім методу createProduct.

Можна навіть до суперкласу перенести конструктор, змінивши його ім'я.

```
public WoodShop(String name, WoodDirectory wd, ProductStore s, int n) {
      super();
      this.name = name;
      this.wd = wd;
      this.ps = s;
      this.n = n;
}
```

Але у класі TimberShop конструктор з тими ж параметрами теж має бути:

```
public TimberShop(String name, WoodDirectory wd, ProductStore s, int n) {
      super(name, wd, s, n);
}
```

Далі у класі WoodShop треба визначити абстрактний метод createProduct і сам клас зробити абстрактним.

```
protected abstract IWeight createProduct();
```

10.2.5 *Тестування модифікованого класу TimberShop*

Змінимо метод startApp у класі TestApp такий:

```
private void startApp() {
	TimberShop shop1 = new TimberShop("shop1",wd, ps,3);
	Thread tshop1 = new Thread(shop1);
	tshop1.start();
}
```

Як бачимо, такий спосіб створення потоку потребує зайвого кроку.

10.2.6 *Створення класу CylinderShop*

Цей клас достатньо просто створити успадковуючи клас WoodShop. Ви маєте це зробити самостійно.

10.2.7 *Тестування паралельної роботи об'єктів класів TimberShop та CylinderShop*

Для тестування модифікуємо метод startApp:

```
private void startApp() {
	WoodShop shop1 = new TimberShop("timberShop", wd, ps, 3);
	WoodShop shop2 = new CylinderShop("cilinderShop", wd, ps, 3);
	Thread tshop1 = new Thread(shop1);
	Thread tshop2 = new Thread(shop2);
	tshop1.start();
	tshop2.start();
}
```

10.2.8 *Призупинення роботи потоків*

10.2.8.1 **Метод sleep для затримки потоку**

Статичний метод sleep класу Thread призупиняє виконання коду потоку на заданий час. Метод викидає виняток, якщо потік перевано під час зупинки.

Ми можемо скористатися цим методом для імітації затримки у часі на створення продуктів у класах, що були створені вище. Різниця полягає у тому, що під час затримки потік не використовує процесор, тож навантаження на комп'ютер буде менше. Метод sleep часто використовується для мультиплікації.

Для прикладу модифікуємо тіло методу run класу WoodShop:

```
for (int i = 0; i < n; i++) {
	try {
		Thread.sleep(500 + rnd.nextInt(500));
	} catch (InterruptedException e) {
		e.printStackTrace();
	}
	IWeight timber = createProduct();
	System.out.println(this.getName() + " create " + timber);
}
```

Протестуйте цей варіант з різними значеннями затримок.

10.2.8.2 **Метод join для призупинення потоку**

Метод join використовується для призупинки потоку аж поки не завершить роботу інший потік, до якого приєднаний (join) даний потік.

Хай, наприклад, нам потрібно після завершнння роботи потоків вивести на консоль кількість продуктів у сховищі. Для цього додаймо такий рядок у кінці методу startApp класу TestThread, після запуску трьох потоків.

```
System.out.println(ps.getCount());
```

Допишіть цей рядок і запустіть додаток. Ми отримаємо точно не 6, а скоріше за все нуль. Справа у тому, що код написаний у методі start теж виконується у окремому потоці і його виконання продовжується після активізації потоків tshop1 та tshop2, а правильний результат можна отримати тільки після завершення цих потоків. Саме у цьому випадку може стати у нагоді метод join. Але замість одного рядка нам знадобиться вже цілий блок коду:

```
try {
      tshop1.join();
      tshop2.join();
      System.out.println(ps.getCount())
} catch (InterruptedException e) {
      e.printStackTrace();
}
```

Тепер результат має бути правильний.

Слід зазначити, що вставляти такі блоки у код методів не зовсім корректно, хоча код працює. Краще такі блоки коду виконувати окремими потоками, наприклад, так:

```
(new Thread(() -> {
      try {
            tshop1.join();
            tshop2.join();
            System.out.println(ps.getCount())
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
})).start();
```

Тут створюється потік через конструкто з параметром Runnable, який представлено у вигляді лямбда функції.

10.2.9 *Проблема використання спільного ресурсу кількома потоками*

Давайте модифікуємо метод run класу WoodShop, у якому ліквідуємо затримку у часі і замість виведення продукту на консоль будемо додавати продукти до сховища:

```
public void run() {
	for (int i = 0; i < n; i++) {
		IWeight product = createProduct();
		ps.add(product);
	}
}
```

Тепер, якщо ми запустимо наш додаток, два потоки будуть по три рази використовувати спільний ресурс – сховище продуктів, а точніше колекцію типу ArrayList. У випадку такої малої кількості звернень негативні наслідки спільного використання можна і не побачити. Якщо ж ви не отримали число 6, то вам пощастило.

А тепер давайте у методі start змінимо значення параметру n з 3 на 1000. У цьому разі скоріш за все число 2000 ми не отримаємо. А кому пощастить, так той ще й отримає винятокову ситуацію часу виконання.

Справа в тім, що колекція ArrayList не пристосована для багатопотокового використання.

10.2.10 *Синхронізація спльного використання ресурсів*

Класичне вирішення проблеми, яка виникла у попередньому пункті полягає у блокуванні доступу до ресурсу, коли його використовує якийсь потік.

Для блокування ресурсу, блоку коду або методу використовується ключове слово synchronize. Ресурс, до якого воно застосовано, може використовуватися тільки одним потоком.

Проаналізуймо варіанти синхронізації.

10.2.10.1 **Синхронізація методу**

У нашому застосуванні потоки звертаються до сховища через метод add класу ProductStore, отже, можна засинхронізувати цей метод:

```
public synchronized boolean add(IWeight newTimber) {
	list.add(newTimber);
	return true;
}
```

Спробуйте протестувати наш додаток, із синхронізованим методом. Результат має бути правильним.

Синхронізація методу вважається найбільш жорстким способом. Справа у тому, що у цьому разі блокується не тільки метод, але й об’єкт, що його викликав. Тобто виклик методу add робить недоступним наше сховище.

10.2.10.2 **Синхронізація блоку**

Щоб підвищити швидкодію додатку, можна синхронізувати тільки критичну ділянку коду у методі, але у цьому випадку треба визначитися з так званим «монітором», що буде також заблокований на період виконання блоку. За звичай, у якості «монітору» використовують спільний ресурс, що підлягає блокуванню.

Як приклад, змінимо метод add і протестуємо нову версію цього методу:

```
public boolean add(IWeight newTimber) {
	synchronized(list){
		list.add(newTimber);
	}
	return true;
}
```

В якості монітору можна використовувати будь який об’єкт, наприклад, спеціально для цього створений:

```
private Object monitor = new Object();
public  boolean add(IWeight newTimber) {
	synchronized (monitor ){
		list.add(newTimber);
	}
	return true;
}
```

Цей варіант теж має працювати.

10.2.11 *Потокобезпечні колекції*

Блокування ресурсів, що є у спільному користуванні, призводить до уповільнення роботи додатку. Тому такий класичний спосіб вважається недостатньо ефективним.

Зніміть синхронізацію сховища, що була налаштована у попередньому пункті, і проаналізуймо інші способи забезпечення безпечного використання нашого сховища.

10.2.11.1 **Колекція Vector**

Є різні рівні потокової безпеки колекцій. Як ми побачили, клас ArrayList не може забезпечити безпечне багатопотокове використання без додаткових заходів. Дещо кращі справи з колекцією Vector. Якщо замінити ArrayList на Vector у класі ProductStore, то застосування буде працювати вірно. Але не слід тішити себе цим. Якщо справа дійде до ітераторів, Vector теж не гарантуватиме багатопотокову безпеку.

10.2.11.2 **Потокобезпечні оболонки колекцій класу Collections**

Клас Collections має низку статичних методів, що створюють потокобезпечні оболонки для колекцій. Наприклад, якщо до методу цього класу synchrinizelist передати звичайну колекцію типу List, то метод поверне потокобезпечний аналог колекції.

Поміняйте визначення колекції у класі ProductStore на таке:

```
private List<Object> list = Collections.synchronizedList(new ArrayList<>());
```

Протестуйте зміни. Додаток має працювати правильно.

10.2.11.1 **Потокобезпечні колекції пакету java.util.concurrent**

Спосіб створення потокобезпечних колекцій засобами класу Collections вважається не дуже ефективним. У версії Java 5 з’явився пакет

java.util.concurrent, що містить низку нових класів для створення потокобезпечних колекцій. Інтерфейс List реалізовано у класі CopyOnWriteArrayList. Але і цей клас вважається не дуже ефективним, якщо виконується багато операцій запису.

Протестуємо цей клас, для чого поміняємо визначення колекції у класі ProductStore на таке:

```
private List<Object> list = new CopyOnWriteArrayList<>();
```

Додаток теж має працювати правильно.

10.3 ЗАВДАННЯ НА САМОСТІЙНУ РОБОТУ

Традиційного завдання для самостійної роботи у цій лабораторній роботі не передбачено, якщо не вважати таким створення класу CylinderShop та останнього з контрольних питань.

10.4 ВИМОГИ ДО ЗВІТУ.

- Назва роботи.
- Мета роботи.
- Тексти класів, досліджуваних в роботі.
- Тексти методів, що змінювалися.
- Результати експериментів з класами та методами.
- Висновки.

10.5 КОНТРОЛЬНІ ПИТАННЯ

1. Загальні відомості про потоки виконання коду.
2. Способи створення потоків в Java.
3. Конструктори класу Thread.
4. Методи класу Thread.
5. Інтерфейс Runnable, його призначення і використання.
6. Засоби для призупинення потоків.
7. Проблеми використання спільних ресурсів та синхронізація.
8. Способи створення потокобезпечних колекцій.
9. Є карта, де ключами є числа 0, 1, 2 … n-1, а значеннями - колекції довільного розміру, що містять цілі числа. Потрібно спочатку наповнити карту випадковими даними, а потім, шляхом обробки кожної колекції окремим потоком, сформувати масив довжиною n, елементи якого містить середні значення вмісту кожної з колекцій карти.

11 ЛАБОРАТОРНА РОБОТА № 11. ОРГАНІЗАЦІЯ ВЗАЄМОДІЇ ПОТОКІВ, ЩО ВИКОНУЮТЬСЯ ПАРАЛЕЛЬНО

Мета роботи:

– Познайомитися з методами організації взаємодії потоків.

– Закріпити навички роботи з потоками виконання коду в Java шляхом створення багатопоточного додатку.

11.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Продовжити розробку додатку, що моделює роботу деревообробної фірми. Ми маємо опрацювати додаткову умову, яка полягає у тому, що цехи з виробництва брусів та кругляків окрім поставок продуктів до сховища ProductStore постачають залишки виробництва до сховища обмеженого розміру, звідки цех обробки залишків забирає їх на переробку. Якщо це сховище заповнено, то цехи постачальники призупиняються. А обробка залишків може починатися тільки якщо це сховище не пусте.

11.2 ПОРЯДОК ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Створімо новий проект для лабораторної роботи куди скопіюємо пакети з попередньої лабораторної роботи та за допомогою рефакторингу назвемо їх lab11.model, lab11.store, lab11.test та lab11.threads.

11.2.1 *Створення класу lab11.store.WasteStore*

Цей клас будемо використовувати для збереження залишків виробництва, а точніше їх умовної кількості у вигляді цілого числа, з метою спрощення класу. Тобто будемо вважати, що після виготовлення одного бруса або кругляка створюється одна умовна одиниця залишків.

Створімо у пакеті lab11.store клас WasteStore та визначимо у ньому два поля – size, що буде зберігати кількість залишків, та maxSize – яке буде зберігати значення найбільшої кількості залишків, що може прийняти сховище:

```
public class WasteStore {
	private int maxSize;
	private int size = 0;
```

Далі згенеруємо конструктор, через який будемо встановлювати значення maxSize:

```
public WasteStore(int maxSize) {
	super();
	this.maxSize = maxSize;
}
```

Згенеруємо також методи доступу до визначених полів, тільки геттери.

Окрім того створімо два публічних методи, за допомогою яких будемо додавати та вилучати одиниці залишків. У цих же методах будемо виводити на консоль значення залишку після зміни. Це дозволить аналізувати роботу додатку.

```
public void addWithPrint(WoodShop shop) {
    size++;
    System.out.println(shop.getName() + " add 1, became " + size);
}

public void removeWithPrint(WoodShop shop) {
    size--;
    System.out.println(shop.getName() + " remove 1, became " + size);
}
```

11.2.2 *Створення класу lab11.threads.WasteShop*

Об’єкти цього класу будемо використовувати як переробників залишків. Цьому класу потрібно дещо з визначеного у класі WoodShop, але дещо має бути перевизначено.

Створімо у пакеті lab11.threads клас WasteShop, що успадковує клас WoodShop.

Далі напишемо конструктор, або змінімо, якщо його було згенеровано:

```
public WasteShop(String name,  ProductStore s,  WasteStore ws,
            TimberShop timberShop, CylinderShop cylinderShop) {
        this.n = timberShop.getN() + cylinderShop.getN();
        this.name = name;
        this.ps = s;
        this.ws = ws;
}
```

Для того, щоб не було помилки слід згенерувати пустий конструктор у класі WoodShop. Ця помилка з’явилася внаслідок того, що ми не використовуємо конструктор суперкласу з параметрами.

Окрім того, у класі WoodShop слід згенерувати метод getN(), що використовується для визначення кількості мішків із залишками.

Створімо також метод createProduct:

```
IWeight createProduct() {
      float weight = 25 + rnd.nextFloat() * 75;
      Waste waste = null;
      try {
            waste = new Waste(weight);
      } catch (Exception e) {
            e.printStackTrace();
      }
      return waste;
}
```

11.2.3 *Організація взаємодії потоків через методи wait та notyify класу Object*

Відповідно до завдання, переробник залишків може працювати тільки у тому випадку, коли у сховищі є залишки. Так само виробники брусів та кругляків мають зупинитися, якщо сховище залишків повне. Тобто відповідні потоки мають перевіряти умови продовження роботи і якщо вини не виконуються – зупинятися.

Класичним виришенням такої проблеми в Java є використання методів wait та notify класу Object.

Метод wait зупиняє потік, а метод notify відновлює роботу потоку. Слід розуміти, що метод wait викликається у потоці, який слід зупинити, а метод notify викликається у іншому потоці, який відповідає за виконання потрібної умови. Спілкуються ці потоки через вже відомий нам «монітор». Саме цьому об’єкту надсилаються повідомлення wait і notify і на момент відправлення повідомлення «монітор» має бути синхронізовано.

11.2.3.1 **Реалізація методу run у класі WasteShop**

Нижче наведено метод run класу WasteShop, що використовує метод wait для організації чекання появи залишків у сховищі. В якості монітору використовується посилання на сховище ws, що забезпечує блокування цього ресурсу на той час, коли потік з ним працює. Після того, як спрацює wait ресурс звільняється.

```
public void run() {
      for (int i = 0; i < n; i++) {
            // Чекаємо на появу залишків виробництва
            synchronized (ws) {
                  while (ws.getSize() == 0 ){
                        try {
                              ws.wait();
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                  }
                  // Забираємо залишки на обробку
                  ws.removeWithPrint(this);
                  // Повідомляємо про зміну стану wasteStore
                  ws.notify();
            }
            // Створюємо та відправляємо продукт до сховища
            IWeight product = createProduct();
            ps.add(product);
      }
}
```

Зверніть увагу, виклик методу wait() здійснюється всередині циклу, де перевіряється умова очікування. Якщо умова виконується, то метод wait() не викликається. Для забезпечення безпеки потрібен саме цикл, а не if.

А виклик методу notify може бути потрібен іншому потоку, який чекає на вільне місце у сховищі.

11.2.3.2 **Реалізація методу run у класі WoodShop**

Метод run у класі WoodShop визначає поведінку виробників брусів та кругляків.

Самостійно реалізуйте у цьому методі наступний алгоритм роботи виробників.

Виробник створює продукт і відправляє його до сховища продуктів. Після цього перевіряє наявність місця у сховищі залишків.

Якщо сховище залишків повне, потік зупиняється і чекає поки з'явиться вільне місце. Якщо місце є, передає залишки до сховища.

Для того, щоб призупинити роботу виробника у разі переповнення сховища відходів, можна скористатися методом wait, так само як і у попередньому пункті, використовуючи посилання на сховище в якості монітора.

Після збільшення кількості залишків у сховищі треб також сповістити про це потоки, що можуть чекати на цю подію.

11.2.3.3 **Тестування взаємодії потоків**

Для тестування створених або модифікованих класів і методів скористаймося доопрацьованим класом TestApp. Перш за все слід розширити перелік його полів, додавши сховище залишків ws.

```
WoodDirectory wd = new WoodDirectory();
ProductStore ps = new ProductStore();
WasteStore ws = new WasteStore(3);
```

Далі доопрацюємо метод startApp:

```
private void startApp() {
      TimberShop shop1 = new TimberShop("timberShop", wd, ps, ws, wk, 10);
      CylinderShop shop2 = new CylinderShop("cilinderShop", wd, ps, ws, wk, 10);
      WasteShop shop3 = new WasteShop("wasteShop", ps, ws, wk, shop1, shop2);
      Thread tshop1 = new Thread(shop1);
      Thread tshop2 = new Thread(shop2);
      Thread tshop3 = new Thread(shop3);
      tshop1.start();
      tshop2.start();
      tshop3.start();
 }
```

Тестування має показати, що від запуску до запуску потоки працюють по різному, але обробка залишків відбувається тільки при їх наявності, і кількість залишків не буває більше 3.

11.2.4 *Організація взаємодії потоків через об’єкти класів Lock та Condition*

Після появи пакетів java.util.concurrent у версії Java 5, з’явилося багато нових засобів організації паралельної роботи потоків. Використання цих засобів підвищує гнучкість та ефективність цієї роботи.

У лабораторній роботі ми розглянемо тільки один із таких нових способів, а саме використання можливостей, що надає пакет java.util.concurrent.locks. Зокрема будемо використовувати об’єкти типів Lock та Condition.

11.2.4.1 **Реалізація класу WoodLock**

Цей клас ми будемо створювати як спадкоємця класу ReentrantLock, що реалізує інтерфейс Lock. Його доцільно розташувати у пакеті lab11.threads.

У класі створімо дві методи, що будуть повертати об’єкти типу Condition. Перший з цих методів буде повертати об’єкт, пов’язаний з умовою, що сховище залишків пусте. Другий метод буде повертати об’єкт, пов’язаний з умовою, що сховище залишків повне.

Клас може мати такий вигляд:

```
public class WoodLock extends ReentrantLock {

	private Condition full = this.newCondition();
	private Condition empty = this.newCondition();

	public Condition isFull() {
		return full;
	}

	public Condition isEmpty() {
		return empty;
	}
}
```

11.2.4.2 **Модифікація класу WoodShop**

Модифікація класу полягає у тому, що він має приймати посилання на об’єкт типу WoodLock через конструктор і зберігати це посилання доступним для своїх нащадків.

Тож розширимо перелік полів класу WoodShop.

```
protected WoodLock wk;
```

Далі доповнимо перелік параметрів конструктора класу WoodShop параметром типу WoodLock і у конструкторі присвоємо полю wk значення цього параметра.

Зміна конструктора класу WoodShop призведе до необхідності відкорегувати конструктори класів TimberShop та CylyinderShop.

Після проведення цієї підготовчої роботи можна перейти до модифікації методу run цього класу, з використання об’єкту класу WoodLock.

Структурно метод буде виглядати майже так само. Замість блоку, що створювався за допомогою sinchronized, тепер буде фрагмент коду, межи якого визначають виклики метдів lock() та unlock() для об’єкту WoodLock. Окрім того, цей фрагмент коду має бути розташований у конструкції try/cath/finally.

Замість методів wait та notify використовуються методи await і signal, що надсилаються об’єктам Condition.

```
public void run() {
	for (int i = 0; i < n; i++) {
		// Створюємо продукт і передаємо до сховища
		IWeight product = createProduct();
		ps.add(product);
		// Обробляємо залишки
		wk.lock();
		try {
			while (ws.getSize() >= ws.getMaxSize())
				wk.isFull().await();
			// Передаємо залишки до сховища
			ws.addWithPrint(this);
			// Повідомляємо про зміну стану WasteStore
			wk.isEmpty().signal();
		} catch (InterruptedException e) {
			e.printStackTrace();
		} finally {
			wk.unlock();
		}
	}
}
```

11.2.4.3 **Модифікація класу WasteShop**

У цьому класі теж потрібно модифікувати конструктор, але головне – це треба переписати метод run. Це ви маєте зробити самостійно, за зразком з попереднього пункту.

11.2.4.4 **Тестування взаємодії потоків**

Для тестування створених або модифікованих класів і методів скористаймося доопрацьованим класом TestApp. Перш за все слід розширити перелік його полів, додавши об’єкт wk типу WoodLock.

```
WoodLock wk = new WoodLock();
```

Далі потрібно додати цей об’єкт до переліку параметрів у викликах конструкторів.

Результат тестування має довести можливість використання об’єкту класу WoodLock для організації взаємодії потоків під час паралельної роботи.

11.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Окремих завдань на самостійну роботу нема, окрім тих, що розглядалися вище, та в останньому пункті контрольних питань.

11.4 ВИМОГИ ДО ЗВІТУ

- Назва роботи.
- Мета роботи.
- Тексти методів, для реалізованих класів.
- Копії результатів тестування варіантів застосунку.
- Висновки.

11.5 КОНТРОЛЬНІ ПИТАННЯ

1. Пояснити необхідність синхронізації в задачі і показати в коді, як вона реалізована.
2. Поясніть реалізацію взаємодії потоків з використанням методів wait та notify.
3. Поясніть реалізацію взаємодії потоків з використанням об’єктів типу Lock та Condition.
4. Відтворити метод, що імітує створення продукту.
5. Реалізувати дитячу казочку про дітей сороки-ворони, що кашку варила. Кожен з дітей виконував одну з таких робіт: крупу драв, дрова рубав, воду носив, піч топив. А сорока-ворона кашку варила. Потім діти їли кашу. Щоб піч топити потрібні дрова. Щоб кашу варити потрібні крупи, вода і розтоплена піч. А кашу можна їсти після того, як вона звариться.

12 ЛАБОРАТОРНА РОБОТА № 12. РЕАЛІЗАЦІЯ МЕХАНІЗМУ ПОДІЙ В JAVA

Мета роботи:

– Познайомитися з поняттям події і класами Java для створення подій.

– Познайомитися з поняттям слухача події та інтерфейсами Java для створення слухачів подій.

– Познайомитися зі стандартним рішеннями в Java для створення подій та слухачів цих подій.

12.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Закріпити навички роботи з подіями в Java шляхом доопрацювання додатка для паралельної роботи, створивши події надходження виробів до сховищ та засоби обробки цих подій.

У роботі ми розглягнемо особливості роботи з подіями в Java на прикладі події надходження виробу до сховища продуктів.

Особливість роботи з подіями полягає у наявності двох суб'єктів цього процесу – джерела події і обробника події (слухача). Ці суб'єкти, за звичай, мало пов'язані один з одним, можуть створюватися у різні часи, і існувати у різних групах класів.

12.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Створімо новий проект для лабораторної роботи куди скопіюємо пакети з попередньої лабораторної роботи та за допомогою рефакторингу назвемо їх lab12.model, lab12.store, lab12.test та lab12.threads. Ще нам знадобиться пакет lab12.events.

12.2.1 *Реалізація заходів джерела подій, що надають можливість слухачеві реагувати на події*

Джерелом подій буде сховище продуктів, точніше, об'єкт типу ProductStore. Цей об'єкт має генерувати події, що сповіщають про появу нового продукту у сховищі.

12.2.1.1 **Створення класу події lab12.event.ProductEvent**

Доцільно, щоб цей клас успадковував клас EventObject, і, окрім цього, зберігав посилання на продукт, що надійшов до сховища, та умовний час, коли це відбулося:

```
public class ProductEvent extends EventObject {

	private IWeight product;
	private long time;

	public ProductEvent(Object source, IWeight product) {
		super(source);
		this.product = product;
		this.time = System. currentTimeMillis ();
	}

	public IWeight getProduct() {
		return product;
	}

	public long getTime() {
		return time;
	}

	@Override
	public String toString() {
		return time % 1000 + ": " + product;
	}
}
```

12.2.1.2 **Створення інтерфейсу для слухачів подій.**

Для усіх слухачів має бути створений єдиний інтерфейс, через який джерело подій буде спілкуватися зі слухачами. Цей інтерфейс має визначати методи, які джерело події буде викликати у разі виникнення події. Створімо його у пакеті lab12.event:

```
public interface IProductListener extends EventListener {
	public void onProductEvent(ProductEvent e);
}
```

12.2.1.3 **Доопрацювання класу ProdactStore, як джерела подій**

Перш за все додаймо у клас ProductStore поле productListeners для зберігання посилань на слухачів подій:

```
List<IProductListener> productListeners = new CopyOnWriteArrayList<>();
```

Далі створімо методи для додавання та вилучення слухачів з цієї колекції:

```
public void addProductListener(IProductListener listener) {
	productListeners.add(listener);
}

public void removeProductListener(IProductListener listener) {
	productListeners.remove(listener);
}
```

Після цього можна створити методи сповіщення слухачів про подію, що відбулася:

```
protected void fireProductEvent(ProductEvent obj) {
	productListeners.forEach((lsn) -> lsn.onProductEvent(obj));
}
```

На завершення слід додати до методу add виклик створеного методу сповіщення:

```
public  boolean add(IWeight newTimber) {
	synchronized (monitor ){
		list.add(newTimber);
		fireProductEvent(new ProductEvent(this, newTimber));
	}
	return true;
}
```

Після цього сховище готово для генерації подій та оповіщення зареєстрованих слухачів про ці події.

12.2.2 *Створення слухача, який буде обробляти події сховища*

Слухача подій створимо у методі startApp класу TestApp, на його початку, до запуску потоків:

```
// Створюємо слухача події
IProductListener pLis = new IProductListener() {
	@Override
	public void onProductEvent(ProductEvent e) {
		System.out.println(e);
	}
};
```

Далі його слід зареєструвати у джереда подій, тобто у об’єкта ps, що є сховищем продуктів:

```
// Реєструємо слухача у об'єакта ps
ps.addProductListener(pLis);
```

Тепер можна тестувати додаток.

Для того, щоб не виводилася зайва інфрмація слід закоментувати тестові повідомлення у методах класу WasteStore.

Можна зареєструвати декілька слухачів, які можуть виводити різні повідомлення.

Наприклад, другого слухача, що буде виводити інформацію червоним кольором, можна створити у вигляді анонімного об’єкту:

```
ps.addProductListener(new IProductListener(){
	public void onProductEvent(ProductEvent e) {
		System.err.println(e);
	}
});
```

Ще одного слухача можна додати, використовуючи лямбда функцію, адже інтерфейс IProductListener має тільки один метод, отже може вважатися функціональним:

```
ps.addProductListener((e) ->System.err.println(e.getProduct()) );
```

12.3 ЗАВДАННЯ НА САМОСТІЙНУ РОБОТУ

Реалізувати генерацію та обробку якої небудь події у сховищі залишків.

12.4 ВИМОГИ ДО ЗВІТУ

- Назва роботи.
- Мета роботи.
- Тексти реалізованих класів.
- Копії протоколів про події, що відбулися зі сховищами.
- Висновки.

12.5 КОНТРОЛЬНІ ПИТАННЯ

1. Що така подія, клас EventObject.
2. Заходи джерела подій, які надають можливість реагувати на події.
3. Заходи слухачів, що надають можливість реагувати на події.
4. Відтворити послідовність проходження інформації про подію від виникнення до його обробки.

13 ЛАБОРАТОРНА РОБОТА № 13. ЗНАЙОМСТВО З БІБЛІОТЕКОЮ SWING

Мета роботи:

– Ознайомитися з компонентами бібліотеки javax.swing.

– Ознайомитися з технологією побудови графічного інтерфейсу з використанням Swing Designer.

– Ознайомитися з менеджерами компоновки графічних компонентів.

– Ознайомитися з методикою обробки подій, пов'язаних з компонентами.

– Створити програму з графічним інтерфейсом користувача.

13.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

Створити графічний інтерфейс користувача у додатку для деревообробного підприємства. За основу взяти проект з лабораторної роботи 9.

13.2 ПОРЯДОК ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Створімо новий проект для лабораторної роботи і у папку src проекту скопіюємо пакети з лабораторної роботи 9, та за допомогою рефакторингу назвемо їх lab13.model, lab13.store, lab13.test. Ще нам знадобиться пакет lab13.wiev.

13.2.1 *Створення головного вікна*

Створімо візуальний клас lab13.wiev.MainGui.

Для створення візуального класу можна скористуватися функцією меню File->New->Other->Window Builder->Swing Designer->Application Window, або скористатися іконкою Create new visual classes на панелі інструментів (друга з лівої сторони) .

Маємо отримати діалог створення класу, де потрібно заповнити усі поля, рисунок 13.1.

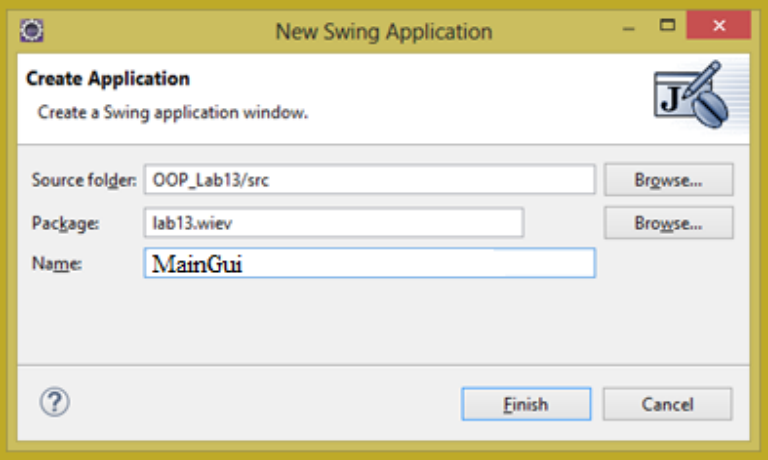

Рисунок 13.1 – Діалог створення візуального додатку

13.2.1.1 **Візуальний редактор**

Візуальний редактор, що з'являється, має дві закладки – Source, для редагування тексту, та Design, для редагування візуальної композиції.

Перш за все попереджаємо, що перехід на закладку Design не завжди відбувається. Якщо з'явилося повідомлення, що не вдалося відкрити вікно дизайнера, нажимайте кнопку Reparse ще і ще. Можливо вам пощастить.

Закладка дизайну, рисунок 13.2, містить багато панелей. Перш за все це панель для створення візуальної композиції.

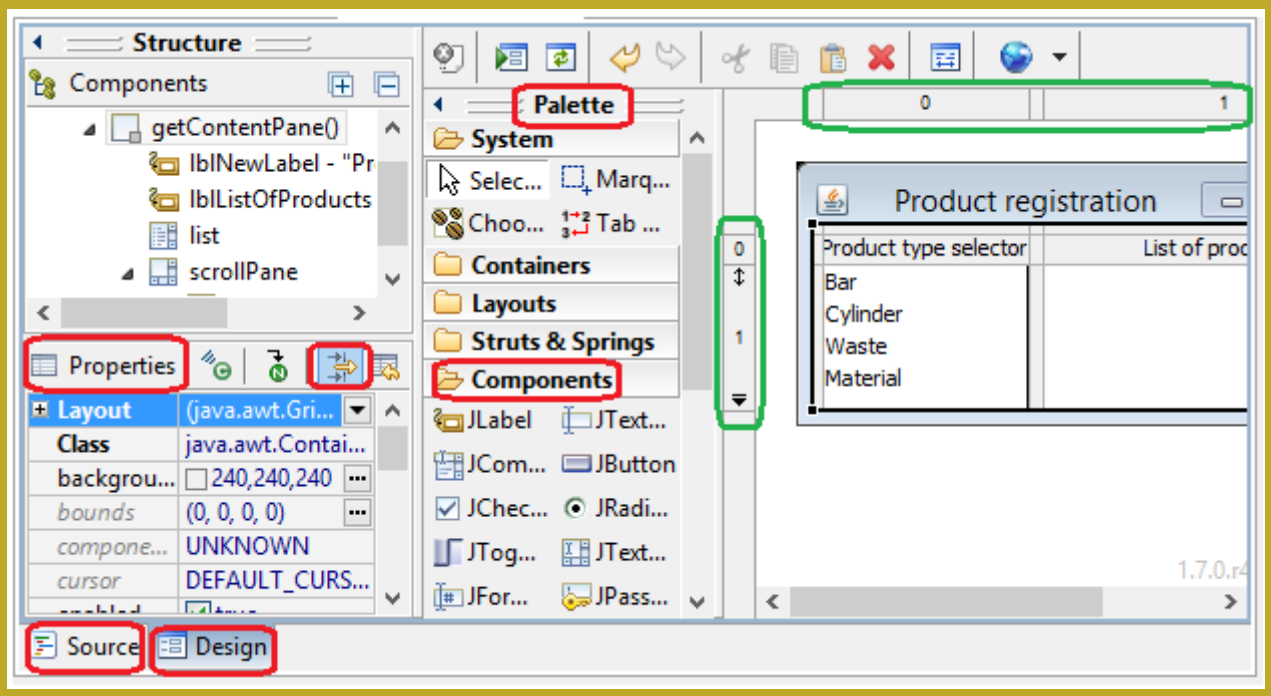

Рисунок 13.2 – Закладка візуального редактора

Далі, слід згадати про панель Palette, з якої можна вибирати потрібні компоненти. Для зручності компоненти об'єднані у групи – Containers, Layouts, Components, …

Для налаштування властивостей компонентів використовується панель Properties. Ця панель працює у двох режимах – повний та скорочений. За замовчуванням панель працює у скороченому режимі. Щоб переключитися у повний режим, використовується кнопка "Show advanced properties".

13.2.1.2 **Редактор коду**

Перейдемо на закладку Source і попрацюймо з кодом класу.

Перш за все ми бачимо, як і раніше, оголошення звичайного класу.

У класі оголошено також поле frame типу JFrame. Фактично саме цей об'єкт ми побачимо на екрані після запуску застосування.

```
public class MainGUI {

    private JFrame frame;
```

Для запуску застосування використовується метод main:

```
/**
 * Launch the application.
 */
public static void main(String[] args) {
	EventQueue.invokeLater(new Runnable() {
		public void run() {
			try {
				MainGUI window = new MainGUI();
				window.frame.setVisible(true);
			} catch (Exception e) {
				e.printStackTrace();
			}
		}
	});
}
```

У цьому методі створюється тимчасовий об’єкт window класу MainGui і через нього, за допомогою методу setVisible, відображується на екрані об’єкт frame.

Усі ці дії виконуються в окремому потоці, що ставиться у системну чергу EventQueue.

Конструктор класу викликає метод initialize, у якому створюється frame, визначаються його розміри та поведінка, пов’язана із закриттям вікна.

```
/**
 * Create the application.
 */
public MainGUI() {
	initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
	frame = new JFrame();
	frame.setBounds(100, 100, 450, 300);
	frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

}
```

Запустіть клас на виконання і познайомтесь із зовнішнім виглядом frame.

Для того, щоб вікно виглядало у режимі виконання так само, як і у режимі проектування так, необхідно у методі main перед рядком, де створюється window, викликати такий метод:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

Зверніть увагу на структура класу, яку пропонує ця версія Swing Designer.

По-перше, усі компоненті створюються у межах методу initialize() і посилання на них містяться у локальних змінних цього методу. Тому за межами методу компоненти недоступні.

По-друге, для компонентів, що були визначені в методі initialize() раніше, недоступні компоненти, що були визначені пізніше.

Проблеми вирішуються за рахунок створення полів класу, які містять посилання на компоненти. Це робити краще за допомогою функції контекстного меню компонента Expose component.

13.2.1.3 **Менеджери компоновки**

Перейдемо у режим Design.

Перед тим, як вибирати та розташовувати компоненти на панелі, слід налаштувати для неї менеджер компоновки.

Менеджери компоновки забезпечують автоматичне розташування компонентів на панелі вікна.

Для вибору потрібного менеджера можна відкрити закладку Layouts на палетці, або скористатися контекстним меню панелі фрейму і вибрати setLayout.

За замовчуванням на панелі налаштовано BorderLayout, який передбачає розташування компонентів у п’яти зонах – NORTH, WEST, EAST, SOUTH, CENTER.

Якщо є бажання відмовитися від автоматичного розташування, слід вибрати Absolute Layout.

Ми рекомендуємо вибрати GridBagLayout.

У разі використання такого менеджера, після розміщення компонента на панелі, ліворуч та зверху вікна дизайнера з’являються зони налаштування колонок та рядків, які мають контекстні меню і допомагають налаштувати поведінку рядка або колонки. На рисунку 13.2 зони налаштування колонок та рядків позначено зеленим кольором.

Діалог налаштування властивостей рядка або колонки відкривається за подвійним кліком.

Окрім того поведінку і положення компонента можна налаштовувати змінюючи параметри властивості Constraints на панелі Properties.

13.2.1.4 **Розміщення компонент на панелі фрейму**

Далі переходимо у режим Design і починаємо формувати головне вікно, яке має бути таким, як показано на рисунку 13.3.

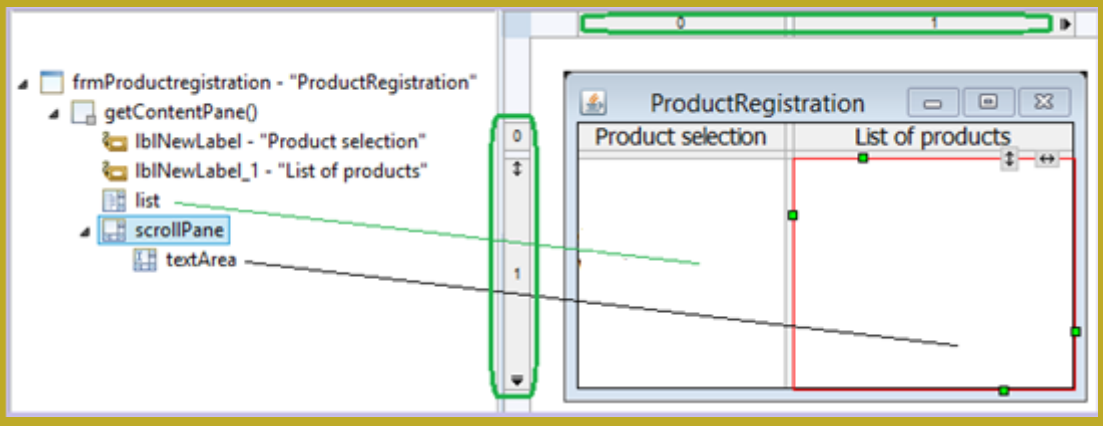

Рисунок 13.3 – Вигляд головного вікна у режимі дизайну

На рисунку ліворуч наведено перелік компонентів, які розташовані на панелі фрейму.

Перш за все сформуємо заголовок вікна. Для цього треба виділити рамку фрейму і налаштувати властивість title на закладці Properties.

Далі відкриємо на палетці закладку Components і розташуємо компоненти JLabel у верхній частині панелі фрейму. Після цього визначимося із властивістю text цих компонентів. Можна змінити і інші властивості, наприклад, font. Такі властивості можна одночасно міняти у декількох компонентів, якщо ці компоненти виділити разом.

Наступним можна встановити компонент JList. Перелік елементів списку ми налаштуємо пізніше.

Далі слід відкрити на палетці закладку Containers і встановити панель JScrollPanel поруч зі списком, а на нижню праву частину цієї панелі встановити компонент JTextArea, що на закладці Components палетки.

Компоненти JList та JTextArea слід зробити доступними для використання у межах класу за допомогою функції контекстних меню компонентів Expose component.

Після налаштування візуальної частини переходимо знову до тексту класу.

Зверніть увагу, що у класі з’явилося два нових поля – list і textArea, і два методи доступу до цих полів. Ці методи ми використовувати не будемо, можна їх вилучити. Наповнення компоненту list даними ми зробимо пізніше.

Проаналізуйте також текст методу initialize.

13.2.2 *Створення діалогів для введення даних*

Інформацію про нові продукти будемо вводити через діалогові вікна, які є об’єктами класів, що успадковують клас JDialog.

13.2.2.1 **Діалог для введення інформації про бруси**

Діалог створюється так само, як і головне вікно, різниця тільки у виборі останнього пункту меню – вибираємо JDialog.

У діалозі, що з’явився, вводимо ім’я класу – DlgTimber.

Після цього потрапляємо до візуального редактора. Якщо перейти до режиму Design, побачимо форму з двома кнопками. Нам її потрібно доопрацювати до вигляду, що представлений на рисунку 13.4. На цьому рисунку ліворуч є список усіх компонентів з їх назвами.

Але спочатку налаштуємо дві властивості діалогу через панель Properties.

Через властивість title визначимо заголовок діалогу – «Timber», або щось інше.

Налаштуємо також значення true для властивості modal. При такому налаштуванні код додатку після виклику діалогу буде чекати на введення даних користувачем, і виконання коду додатку продовжиться тільки після закриття діалогу.

Тепер можна перейти до розміщення елементів для вводу даних на панелі діалогу.

За замовчування, на панелі діалогу встановлюється менеджер компоновки FlowLayout і його можна залишити.

Для вводу розмірів бруса ми будемо використовувати компоненти типу JTextField, а для вибору типу деревини – компонент типу JCombox, які можна знайти на закладці Components палетки.

У компонента comboBox налаштовано властивість border типу TitledBorder. Доступ до цієї властивості можна отримати, якщо викликати їх розширений список (дивись рисунок 13.2).

Розміри компоненту встановлено через властивість prefferedSize (240, 40).

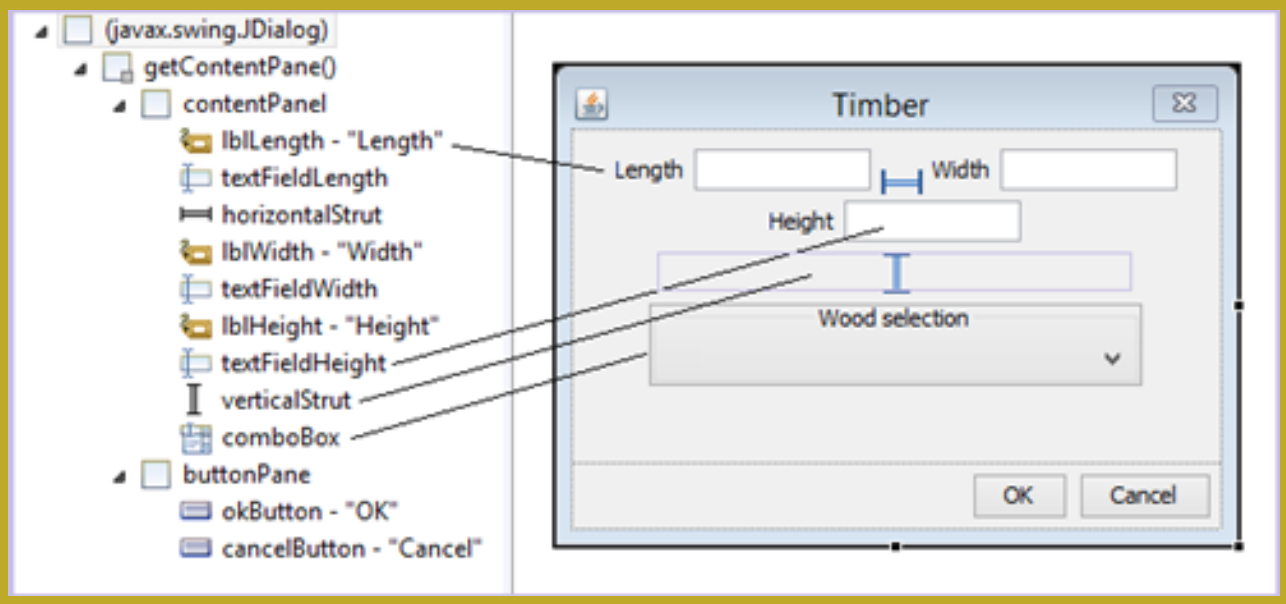

Рисунок 13.4 – Діалог DlgTimber у режимі дизайну

Доцільно поміняти назви компонентів JTextField на textFieldLength, textFieldWidth, textFieldHeight, використовуючи функцію контекстного меню компонентів Rename... Це спростить написання коду у подальшому.

Обов’язково слід зробити ці компоненти, а також comboBox, доступними, скориставшись функцією контекстного меню компонентів Expose Component.

13.2.2.2 **Діалоги для введення даних про круглі бруси та залишки**

Ці діалоги слід створити самостійно, за зразком попереднього діалогу.

13.2.2.3 **Діалог для введення даних про деревину**

Можливий вигляд діалогу DlgWood зображено на рисунку 13.5.

На панелі діалогу встановлено GridBagLayout менеджер компоновки.

Панель scrollPane, яку виділено, займає рядки з 0 по 6. Властивість grow налаштована тільки у рядка 6, та у колонки 1.

Напис List of woods розташовано у верхньому полі панелі scrollPane, а компонент textArea розташовано у головному полі ScrollPane, що знаходиться знизу праворуч.

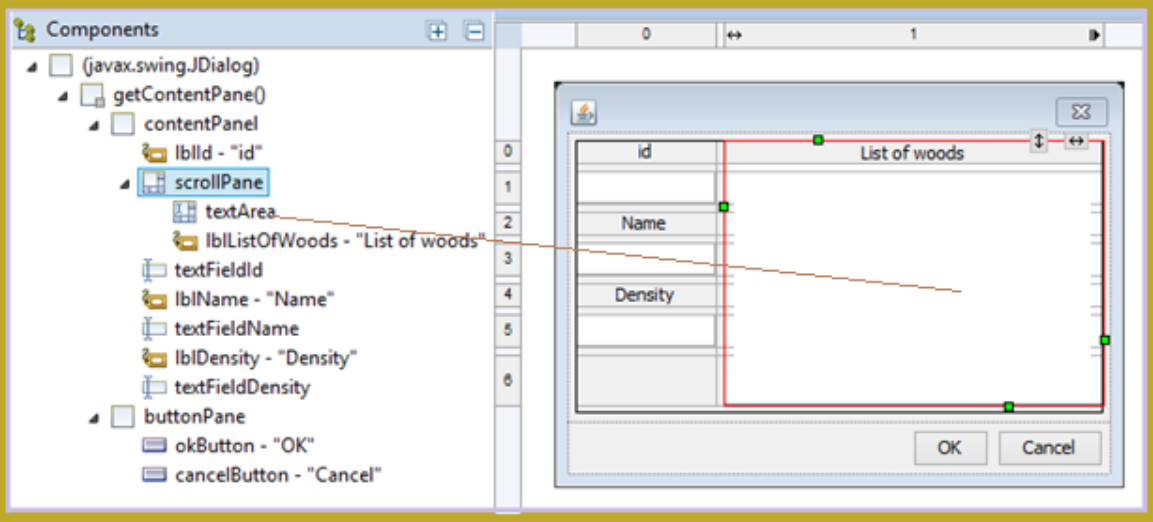

Рисунок 13.5 – Діалог DlgWood у режимі дизайну

13.2.3 *Програмування діалогових вікон*

Після створення візуальних конструкцій діалогів треба забезпечити їх працездатність, додавши до класів необхідний програмний код.

13.2.3.1 **Інтерфейс lab13.wiev.IWoodDialog**

Для того, щоб програмуючи роботу з діалогами можна було писати узагальнений код для роботи з діалогами, створімо інтерфейс, що визначить методи, які будуть потрібні для роботи з усіма діалогами.

```
public interface IWoodDialog {
   public  void setVisible(boolean b);
   public Object getObject();
   public void setWoodDirectory(WoodDirectory wd);
}
```

Далі будемо реалізовувати ці методи у класах діалогів. Фактично, реалізовувати доведеться тільки два останніх методи, бо перший реалізовано у класі JDialog, який використовується як батьківський для створення наших діалогів.

13.2.3.2 **Програмування діалогу DlgTimber**

Відкрийте закладку Source візуального редактора для класу для DlgTimber.

На початку класу ми бачимо перелік знайомих полів, зокрема оголошення комбобоксу. Якщо поля і комбобокс не оголошені, то слід у режимі дизайну викликати функцію Expose component для цих компонентів.

Пропишемо в оголошенні комбобоксу тип даних, що будуть у ньому відображатися. Хоча це будуть об'єкти типу Wood, але виходячи з того, що ми будемо наповнювати комбобокс елементами масиву, які отримаємо через метод getArr класу WoodDirectory, доводиться прописувати тип Object:

```
private JComboBox<Object> comboBox;
```

Далі створімо поле для збереження посилання на об'єкт класу Timber, що буде створюватися у процесі роботи з діалогом:

```
private Timber timber;
```

Тепер заявимо у заголовку класу реалізацію інтерфейсу IWoodDialog:

```
public class DlgTimber extends JDialog implements IWoodDialog{
```

Заявка інтерфейсу у класі призведе до необхідності реалізації методів цього інтерфейсу getObject() та setWoodDirectory(WoodDirectory). Заготовки методів можна згенерувати шляхом автоматичного виправлення коду, а потім їх доопрацювати.

Метод getObject() має виглядати так:

```
public Object getObject() {
      return timber;

}
```

Метод setWoodDirectory(WoodDirectory) передає діалогові посилання на каталог деревини, який використовується для створення моделі для комбобоксу.

```
@Override
public void setWoodDirectory(WoodDirectory wd) {
   ComboBoxModel<Object> model = new DefaultComboBoxModel<>(wd.getArr());
   comboBox.setModel(model);
}
```

Наступний крок – це програмування подій натискання на кнопки діалогу.

Для створення слухача відповідної події слід викликати з контекстного меню кнопки таку послідовність функцій: Add event handler → action → actionPerformed.

Як наслідок, отримаємо у коді класу заготовку створення анонімного обробника події, яку потрібно доповнити потрібним кодом.

Для кнопки "Cancel" все виглядає дуже просто:

```
cancelButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      timber = null;
      DlgTimber.this.setVisible(false);
   }
});
```

Для кнопки “Ok” слід запрограмувати створення об’єкту класу Timber. У наведеному нижче тексті рамочкою виділено текст, яким потрібно доповнити заготовку:

```
okButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      float length = Float.parseFloat(textFieldLength.getText());
      float height = Float.parseFloat(textFieldHeight.getText());
      float width = Float.parseFloat(textFieldWidth.getText());
      Wood wood = (Wood) comboBox.getSelectedItem();
      timber = new Timber(wood, length, height, width);
      DlgTimber.this.setVisible(false);
   }
});
```

На сам кінець згенеруємо метод toString для цього діалогу. Метод буде використовуватися компонентом JList головного вікна під час відображення переліку діалогів, через які будуть вноситися дані про продукт.

Поля діалогу для генерації методу використовувати не треба, можна просто залишити назву класу.

Після створення цього методу діалог готовий до використання.

13.2.3.3 **Використання діалогу DlgTimber**

Використовувати цей діалог буде головне вікно застосування – клас MainGUI. Тому перейдемо до роботи з текстом цього класу.

Перш за все відкоригуємо опис поля list, визначивши тип об’єктів, що будуть вибиратися зі списку. Такими об’єктами будуть наші діалоги, тож опис поля list має бути таким:

```
private JList<IWoodDialog> list;
```

Після цього подовжимо список полів класу, визначивши поля для каталогу деревини, сховища продуктів та тільки що завершеного діалогу, та одразу проініціалізуємо ці поля:

```
private WoodDirectory wd = new WoodDirectory();
private ProductStore ps = new ProductStore();
private DlgTimber dlgTimber = new DlgTimber();
```

Далі знайдіть у методі initialize місце після створення об’єкту list. Там має бути такий текст:

list = new JList<>();

Після цього рядка слід додати код, що визначить модель для цього списку:

```
DefaultListModel<IWoodDialog> model = new DefaultListModel<>();
model.addElement(dlgTimber);
list.setModel(model);
```

Зверніть увагу, в якості елементу списку ми використовуємо діалог. Однак у списку він буде відображатися рядком тексту, який формує метод діалогу toString().

Далі запрограмуємо подію вибору елемента зі списку.

Для цього слід перейти у режим Design і через послідовність функцій контекстного меню Add event handler → mouse → mouseClick створити заготовку обробника події натискання на кнопку миші для списку. Далі доповнити метод mouseClicked таким кодом:

```
public void mouseClicked(MouseEvent e) {
   IWoodDialog dlg = (IWoodDialog) list.getSelectedValue();
   dlg.setWoodDirectory(wd);
   dlg.setVisible(true);
   Object obj = dlg.getObject();
   if (obj != null)
      ps.add((IWeight)obj);
   textArea.setText(ps.toString());
}
```

13.2.3.4 **Програмування та використання діалогів DlgCylinder і DlgWaste**

Ця роботу має бути виконана в межах самостійної роботи за зразком, що розглянуто у попередніх пунктах. Особливістю діалогу DlgWaste є те, що йому не потрібен комбобокс для вибору деревини, тому метод setWoodDirectory слід залишити пустим.

13.2.3.5 **Програмування та використання діалогу DlgWood**

Призначення цього діалогу забезпечувати поповнення каталогу деревини та його перегляд. Він також має бути модальним. Програмування його має деякі особливості, тому розглянемо ці особливості.

Так само, як і в інших діалогах заявимо у заголовку класу реалізацію інтерфейсу IWoodDialog і згенерувати заготовки методів цього інтерфейсу

```
public class DlgWood extends JDialog implements IWoodDialog{
```

У класі діалогу слід визначити поле для посилання на каталог деревини:

```
private WoodDirectory wd;
```

Далі слід реалізувати метод setWoodDirectory, заготовка якого вже було створено. У методі ініціалізується поле wd, та у компонент textArea виводиться початковий вміст каталогу деревини:

```
@Override
public void setWoodDirectory(WoodDirectory wd) {
   this.wd = wd;
   textArea.setText(wd.toString());
}
```

Метод інтерфейсу getObject можна залишити без змін.

Далі слід запрограмувати реакцію на подію натиснення на кнопку«Ok» та «Cancel», використовуючи подію actoinPerformed, або moseClick:

```
okButton.addMouseListener(new MouseAdapter() {
      @Override
      public void mouseClicked(MouseEvent e) {
            int id = Integer.parseInt(textFieldId.getText());
            String name = textFieldName.getText();
            float density = Float.parseFloat(textFieldDensity.getText());
            wd.add(new Wood(id, name, density));
            textArea.setText(wd.toString());
      }
});
cancelButton.addMouseListener(new MouseAdapter() {
      @Override
      public void mouseClicked(MouseEvent e) {
            DlgWood.this.setVisible(false);
      }
});
```

На завершення роботи з діалогом слід згенерувати метод toString, не використовуючи поля, або сформувати цей метод самостійно.

Для тестування діалогу треба у класі MainGUI прописати поле, що містить посилання на цей діалог, та проініціалізувати його, так само, як це було зроблено для діалогу DlgTimber. Після цього додати посилання на діалог до моделі списку.

13.3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Додаткових завдань, окрім визначених у пунктах 13.2.2.2 та 13.2.3.4, нема.

13.4 ВИМОГИ ДО ЗВІТУ

- Назва роботи.
- Мета роботи.
- Зображення візуальних компонентів, що були створені, та код відповідних класів.
- Результати тестування застосування у вигляді копій вікон з результатами.
- Висновки.

13.5 КОНТРОЛЬНІ ПИТАННЯ

1. Загальна характеристика та призначення контейнерів (закладка палетки Containers).
2. Загальна характеристика та призначення менеджерів розташування.
3. Загальна характеристика та призначення елементів управління (закладка палетки Coomponents).
4. Поняття про модель для графічного компоненту.
5. Обробка подій графічних компонент.
6. Пояснити, де у створеному проекті для реалізації інтерфейсу користувача реалізовано можливості, пов’язані з інкапсуляцією, спадковістю та поліморфізмом.

14 ЛАБОРАТОРНА РОБОТА № 14.
РОБОТА З ГРАФІКОЮ

Мета роботи:

- Познайомитися з класами, що реалізують графічні можливості Java.
- Створити застосування для реалізації простої мультиплікації.

14.1 ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

У лабораторній роботі студент має дослідити можливості класів Graphics2D, BasicStroke, GeneralPath, Shape, AffineTransform, BuferedImage для створення зображень.

14.2 ПОРЯДОК ВИКОНАННЯ РОБОТИ

Створімо проект для лабораторної роботи, пакет graph та клас візуальний клас TestGraph типу Application Window.

У класі створімо публічний метод testGraph();

```
public void testGraph() {
      System.out.println("This is testGraph");
}
```

Виклик цього методу пов’яжемо з відкриттям нашого додатку. Для цього теба виділити рамку, саме рамку вікна і викликати послідовність функцій контекстного меню: Add event handler → mouse → mouseClick.

Далі, у тілі методу mouseClicked, що був згенерований в межах створення анонімного об’єкту, вставити виклик нашого методу:

```
testGraph();
```

Через цей метод ми будемо тестувати приклади реалізації графічних можливостей, а зараз перевіримо просто коректність його виклику.

Після запуску застосування і кліку по формі на консолі має з’явитися повідомлення.

14.2.1 *Клас Graphics*

Кожний графічний компонент Java має у своєму складі об'єкт типу Graphics або Graphics2D, який називають графічним контекстом (graphics context).

З огляду на те, що графічний контекст суттєво залежить від конкретної графічної платформи, класи Graphics та Graphics2D зроблені абстрактними. Тому не можна безпосередньо створити екземпляри цих класів. Однак кожна віртуальна машина Java реалізує методи цих класів, створює їх екземпляри для кожного графічного компонента і надає доступ до об'єктів класу Graphics. Для цього слугує метод getGraphics() класу Component. Об’єкти цього класу

передаються також як аргументи методів paint(Graphics), update (Graphics) та інших.

Для використання розширених можливостей класу Graphics2D, об’єкт типу Graphics, що надається названими вище методами, слід привести до типу Graphics2D.

Створімо у методі testGraph змінну типу Graphics2D для панелі фрейму, яку ми будемо використовувати для виведення зображень:

```
Container c = frame.getContentPane();
Graphics2D g = (Graphics2D) c.getGraphics();
```

14.2.1.1 **Налаштування кольорів**

Графічний контекст містить поточний колір малювання і колір фону. Колір є об'єктом класу java.awt.Color.

Налаштування та доступ до поточного кольору контексту забезпечується методами setColor(Color) та getColor(). Методи setBackground(Color) та getBackground() забезпечують доступ до кольору фону.

Якщо немає потреби ретельно підбирати кольори, то можна скористатися однією з тринадцяти статичних констант, наявних у класі Color: BLACK, BLUE, CYAN, DARKGRAY, GRAY, GREEN, LIGHTGRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW.

Налаштуймо кольори об’єкту g у методі testGraph та спробуємо щось зобразити на формі, додавши до тексту методу такий код:

```
int w = c.getWidth();
int h = c.getHeight();

g.setBackground(Color.YELLOW);
g.clearRect(0, 0, w, h);
g.setColor(Color.BLUE);
g.fillRect(0, 0, w, h / 2);
```

Тест методу, що виконує це завдання, та результат його роботи має бути у звіті.

14.2.1.2 **Методи рисування**

У попередньому прикладі ми вже скористалися деякими методами рисування, що можна викликати для графічного контексту.

Ваше завдання познайомитися з переліком цих методів у класі Graphics і за допомогою його методів зобразити на формі лінію, овал, прямокутник, трикутник, а також ці фігури залиті кольором.

Тест методу, що виконує це завдання, та результат його роботи має бути у звіті.

14.2.1.3 **Виведення тексту**

Графічний контекст дозволяє також виводити в область рисування тексти. Щоб налаштувати шрифт для тексту можна скористатися методом setFont(Font).

Протестуйте приклад і зафіксуйте результат:

```
g.setColor(Color.GREEN);
g.setFont(new Font(Font.DIALOG, Font.ITALIC, 32));
g.drawString("Слава Україні", 5, h/2);
```

14.2.2 *Клас BasicStroke*

Клас BasicStroke визначає базовий набір атрибутів пера, яке використовує об'єкт Graphics2D для зображення контурів графічних фігур. Значення атрибутів задаються або числами, або константами класу BasicStroke.

Створити перо з потрібними характеристиками можна за допомогою одного з конструкторів класу BasicStroke.

Основний конструктор BasicStroke (float width, int cap, int join, float miter, float [] dash, float dashBegin) задає наступні параметри:

– width, товщина пера в пікселах;

– cap, стиль оформлення кінця лінії, що визначається однією з констант:

a) CAP_ROUND - закруглений кінець лінії;

b) CAP_SQUARE - квадратний кінець лінії;

c) CAP_BUTT - оформлення відсутнє;

– join, спосіб сполучення ліній, що визначається однією з констант:

a) JOIN_ROUND - лінії сполучаються дугою кола;

b) JOIN_BEVEL - лінії сполучаються відрізком прямої, перпендикулярної бісектрисі кута між лініями;

c) JOIN_MITER - лінії просто стикуються.

– miter, відстань між лініями починаючи з якої застосовується сполучення, не може бути менша за 1;

– dash, масив, що задає довжину штрихів та проміжків між ними для пунктирних ліній. Елементи масиву з парними індексами задають довжину штриха в пікселах, елементи з непарними індексами - довжину проміжку і масив перебирається циклічно;

– dashBegin відстань, на яку зсувається початок лінії.

Решта конструкторів задають деякі характеристики за умовчанням.

Після створення пера його можна передати графічному контексту за допомогою методу setStroke(Stroke) та рисувати.

Щоб подивитися, як працює цей об’єкт можна, можна скопіювати наступний код, та додати його до методу testGraph:

```
int size = h / 4;
int top = h / 16;
int left = w / 16;

g.setColor(Color.ORANGE);
g.fillOval(left, top, size, size);

g.setColor(Color.YELLOW.darker());
g.setStroke(new BasicStroke(3, BasicStroke.CAP_ROUND,
     BasicStroke.JOIN_MITER, 1, new float[]{ 10, 5 }, 0));
```

```
      int x1, x2, y1, y2, r1 = size / 2, r2 = w * 2 / 3;
      for (double fi = 0; fi < 2 * Math.PI; fi += Math.PI / 12) {
         x1 = (int) (left + size / 2 + r1 * Math.cos(fi));
         y1 = (int) (top + size / 2 + r1 * Math.sin(fi));
         x2 = (int) (left + size / 2 + r2 * Math.cos(fi));
         y2 = (int) (top + size / 2 + r2 * Math.sin(fi));
         g.drawLine(x1, y1, x2, y2);
      }

      int penSize = size / 16;
      g.setStroke(new BasicStroke(penSize));
      g.setColor(Color.ORANGE.brighter());
      g.drawOval(left, top, size, size);
```

У звіті має бути цей код і результат його виконання.

Під час захисту роботи студент має пояснити призначення частин коду.

14.2.3 *Інтерфейс Shape*

Інтерфейс Shape оголошує сукупність методів для опису та інспектування 2D геометричних об'єктів.

Особливість цього типу полягає у тому, що програміст може оперувати геометричними фігурами як об’єктами. Наприклад фігуру можна створити, розтягнути, повернути, перемістити а потім вже відобразити.

Інтерфейс Shape реалізовано у групі класів пакету java.awt.geom.

За допомогою цих класів користувач може сконструювати практично будь-яку геометричну форму і відтворити її на екрані за допомогою виклику методів draw() або fill() контексту Graphics2D.

Прості фігури створюються за допомогою таких класів як Line2D, Arc2D, QuadCurve2D, Rectangle2D, Ellipse2D та інших. Особливість цих класів полягає у тому, що для створення об’єктів цих класів слід користуватися конструкторами їх внутрішніх класів, що мають назви Float і Double. Вибір одного з них визначає тип чисел, що задають координати фігури.

Створімо новий метод testShape і будемо його викликати замість, або після методу testGraph. Метод можна скопіювати.

```
   private void testShape() {
      Container c = frame.getContentPane();
      Graphics2D g = (Graphics2D) c.getGraphics();

      int w = c.getWidth();
      int h = c.getHeight();
      g.clearRect(0, 0, w, h);

      Shape line = new Line2D.Float(40, 20, w-40, h-20);
      g.draw(line);

      Rectangle rk = line.getBounds();
      g.setColor(Color.RED);
      g.drawRect(rk.x,rk. y, rk.width, rk.height);
   }
```

У цьому методі створюється об’єкт line – пряма лінія. Але слід мати на увазі, що це графічний об’єкт, а не геометричний. Тобто line не знає, наприклад, які у неї координати кінців. Фактично, цей об’єкт знає своє зображення та прямокутник, що обмежує простір навколо об’єкту. Саме цей прямокутник виведено червоним кольором.

14.2.4 *Клас GeneralPath*

Клас GeneralPath є нащадком класу Path2D.Float і є зручним засобом створення плоских геометричних фігур складної форми. Об’єкти цього класу схожі на колекції, що містять послідовність простих геометричних фігур (segments), і навіть надають доступ до ітератора типу PathIterator.

Фактично, об’єкт класу GeneralPath задає маршрут для об’єкту BasicSroke, що буде вирисовувати задану фігуру. Маршрут складається з послідовності точок і правил їх з’єднання.

Для створення об'єкту цього класу можна скористатися конструктором за замовчуванням GeneralPath().

У створеному об'єкті є атрибут «поточна точка». Початкові значення координат цієї точки – (0, 0), але її можна перемістити у точку (x0, y0) методом moveTo (float x0, float y0).

Від поточної точки до точки (x, y) можна провести:

– відрізок прямої методом lineTo (float x, float y);

– криву Безьє другого порядку методом quadTo(float x1, float yl, float x, float y), яка будується на основі поточної точки та двох заданих;

– криву Безьє третього порядку методом curveTo(float xl, float yl, float x2, float y2, float x, float y), на основі поточної точки та трьох заданих.

Поточною точкою після цього стає точка (x, y). Початкову і кінцеву точки можна з'єднати методом closePath ().

Як приклад, протестуємо метод, де створюється трикутник, після чого за тими ж точками створюється крива Безьє другого порядку:

```
private void testShape1() {
        Container c = frame.getContentPane();
        Graphics2D g = (Graphics2D) c.getGraphics();

        int w = c.getWidth();
        int h = c.getHeight();
        g.clearRect(0, 0, w, h);

        GeneralPath gp = new GeneralPath();
        gp.moveTo(20, h - 10);
        gp.lineTo(40, h - 200);
        gp.lineTo(w -20, h - 100);
        gp.closePath();
        gp.quadTo(40, h-200, w-20, h-100);
        g.draw(gp);

}
```

Самостійно створіть схожий метод, що вирисовує чотирикутник та відповідну криву Безьє третього порядку.

14.2.5 *Режими рисування*

Клас Graphics надає можливість використовувати окрім звичайного другий режим рисування, що має назву XOR і встановлюється методом setXORMode(Color). Цей режим використовує альтернативний колір, що передається у метод як параметр. У цьому режимі колір фігури, що зображується, визначається шляхом об'єднання бітів кольору фону та альтернативного кольору за допомогою логічної операції XOR.

Якщо зображення, що намальоване у такому режимі, повторно відобразити у тому ж самому місті, то воно зникає.

Для того, щоб зображення мало потрібний колір, слід визначити альтернативний колір як результат XOR об'єднання потрібного кольору і кольору фону. Щоб отримати альтеративний колір для потрібного кольору можна скористатися таким методом:

```
private Color altColor(Graphics2D g, Color c) {
    Color back = g.getBackground();
    int rgb = back.getRGB() ^ c.getRGB();
    return new Color(rgb);
}
```

Звичайний режим відновлюється методом setPaintMode().

Як приклад використання альтернативного кольору протестуємо метод, якій забезпечує переміщення червоного круга по панелі довільного кольору:

```
private void testXOR() {
Container c = frame.getContentPane();
Graphics2D g = (Graphics2D) c.getGraphics();
int w = c.getWidth();
int h = c.getHeight();
g.clearRect(0, 0, w, h);
int size = 50;
int x = 0, y = (h - size) / 2, dx = 5;
g.setXORMode(altColor(g, Color.RED));
while (true) {
   g.fillOval(x, y, size, size);
   try {
      Thread.sleep(100);
   } catch (InterruptedException e) {
   }
   g.fillOval(x, y, size, size);
   x += dx;
   if (x > w - size || x <= 0 && dx < 0)
      dx = -dx;
}
}
```

Для того, щоб цей приклад працював, слід скопіювати і попередній метод.

14.2.6 *Афінні перетворення*

Афінне перетворення визначається як об'єкт класу AffineTransform і дозволяє виконувати різноманітні маніпуляції над прямокутником, що обмежує фігуру, і відповідно над самою фігурою.

Найпростіший спосіб для створення такого об’єкту – це використання пустого конструктора. Далі можна налаштовувати створений об’єкт на виконання перетворень за допомогою різноманітних методів. Ось деякі з них:

– translate(double dx, double dy), переміщує фігуру на задану відстань по осям координат;
– scale(double sx, double sy), змінює розміри фігури по осям, відповідно до заданих масштабних коефіцієнтів;
– rotate(double ang, double x0, double y0), повертає фігуру на кут ang відносно точки з координатами x0, y0.

Для отримання трансформованої фігури можна використовувати метод createTransformedShape(Shape sh) об’єктів класу AffineTransform.

Як приклад, розглянемо метод, що відображає на панелі прямокутник, що рухається, змінюється у розмірах та й ще обертається:

```
private Shape sh;
public void movingRect() {
   Container c = frame.getContentPane();
   Graphics2D g = (Graphics2D) c.getGraphics();
   g.setColor(Color.GREEN);
   int w = c.getWidth();
   int h = c.getHeight();
   g.clearRect(0, 0, w, h);
   new Thread(() -> {
      // Створюємо прямокутник
      sh = new Rectangle2D.Float(0, h / 2 - h / 8, w / 4, h / 4);
      while (sh.getBounds().getWidth() > 5) {
         Rectangle r = sh.getBounds();
         // Відображення
         EventQueue.invokeLater(() -> g.fill(sh));
         // Затримка у часі
         try {
            Thread.sleep(500);
         } catch (Exception e) {
         }
         // Відновлення фонового зображення панелі
         c.repaint(r.x, r.y, r.width, r.height);
         // Формування афінного перетворення
         AffineTransform at = new AffineTransform();
         at.translate(40, 0);
         at.scale(0.9, 0.95);
         at.rotate(0.08, r.x + r.width / 2, r.y + r.height / 2);
         // Перетворення зображення
         sh = at.createTransformedShape(sh);
      }
   }).start();
}
```

Для нормальної роботи методу його код потрібно виконувати в окремому потоці. Саме такий потік і створюється у методі.

Для відображення фігури створюється потік методом invokeLate(Runnable) і цей потік стає у чергу на відображення.

Старе зображення вилучається за допомогою методу repaint.

Афінне перетворення формується за рахунок послідовного виклику трьох методів – переміщення, масштабування та повороту.

Наприкінці циклу за допомогою афінного перетворення створюється нова фігура.

14.2.7 *Клас BuferedImage*

Клас успадковує абстрактний клас java.awt.Image і об’єкти цього класу зберігають у собі графічні зображення. Для створення об’єктів цього класу можна використовувати такий конструктор:

BufferedImage(int width, int height, int imageType),

де width та height – розміри зображення у пікселах, а imageType – це одна з констант, що визначає тип зображення, наприклад, BufferedImage. TYPE_INT_ARGB.

Завантажити зображення до об’єкту класу BuferedImage можна декількома способами.

Перший спосіб базується на тому факті, що об’єкт класу BuferedImage має власній графічний контекст типу Graphics, доступ до якого можна отримати через метод getGraphics(). Завдяки цьому для створення зображень можна використовувати усі розглянуті вище засоби для рисування. Однак слід розуміти, що при цьому зображення створюється не на екрані, а в буфері, і не пов’язано з подіями на екрані.

Другий спосіб дозволяє завантажити зображення до буферу з файлів, що мають розширення .jpg або .png. Для завантаження можна використати статичний метод read() класу javax.imageio.ImageIO, який не тільки завантажує файл, але й створює об’єкт типу BuferedImage. У цей метод можна передавати параметри різних типів, але найкраще передавати параметр типу java.net.URL, а самі файли зберігати у проекті, що створюється. У такому випадку файли будуть завантажені у .jar файл і не буде проблем з їх пошуком.

Для того, щоб відобразити вміст об’єкту типу BuferedImage на екрані можна скористатися методом drawImage графічного контексту. Слід тільки використовувати графічний контекст контейнеру, на якому буде відображатися зображення, а не контекст іміджу.

Метод drawImage виглядає так:

```
drawImage(Image img, int x, int y, ImageObserver observer)
```

де img – це імідж, що має бути відображений, x та y – це координати точки на поверхні контейнера, з якої починається відображення зображення, observer – це посилання на об’єкт, який може реагувати на завершення завантаження іміджу. Допустимо значення null або this.

Розміри зображення можна налаштувати під розміри компоненту, на поверхні якого він відображується. Для цього можна використати метод

```
getScaledInstance(int width, int height, int hints),
```

де width та height – потрібні розміри зображення, а hints – це константа, що встановлює правила візуалізації.

14.2.7.1 **Створення панелі із зображенням**

Створімо у проекті пакет other. Завантажимо до цього пакету фото, наприклад d2.jpg. Завантаження можна зробити через контекстне меню проекту, викликавши функції Import... → FileSystem → Next. Далі слід знайти папку із зображенням і виділити потрібний файл у переліку. Щоб побачити файл у пакеті після завантаження, треб зробити refresh проекту.

Далі на початку класу TestGraph, у межах якого ми створюємо методи для дослідження графічних можливостей, визначимо два поля url та іміджу, а також блок ініціалізації, у якому завантажимо .jpg файл до іміджу.

```
private URL url = TestWithFoto.class.getResource("/other/d2.jpg");
private BufferedImage img = null;
{
   try {
      img = ImageIO.read(url);
   } catch (IOException e1) {
      e1.printStackTrace();
   }
}
```

Далі визначимо приватне поле для панелі, що буде використовувати як contentPane для фрейму:

```
private JPanel panel = new JPanel() {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D g2d =(Graphics2D) g;
      Image scaledImg = img.getScaledInstance(getWidth(),
                getHeight(),Image.SCALE_SMOOTH);
      g2d.drawImage(scaledImg, 0, 0, this);
   }
};
```

Тут ми перевизначаємо метод paintComponent класу JPanel, внаслідок чого на поверхні панелі завжди буде відображатися наше фото.

Тепер передаймо нашому фрейму створену панель. Це слід зробити у методі initialize після створення фрейму:

```
frame.setContentPane(panel);
```

Завдання на самостійну роботу

За бажанням можна створити графічну імітацію роботи деревообробної фірми.

Бруси та кругляки з’являються через випадковий час і переміщуються від цехів до сховища залишків. Від переробника залишки теж переміщуються до сховища.

Сховища можна позначати прямокутниками з числом, що відображає кількість продуктів. Продукти позначати простими фігурами – квадратом, колом, трикутником.

14.3 Вимоги до звіту.

- Назва роботи.
- Мета роботи.
- Код та результати тестування прикладів, виконаних під час роботи.
- Клас та зображення самостійно розробленого додатку, якщо такий є.
- Висновки.

14.4 Контрольні питання

- Класи Graphics та Graphics2d і їх методи.
- Клас Color та його конструктори.
- Клас BasicStroke та його налаштування.
- Клас GeneralPath та його методи для рисування.
- Використання методу paintComponent(Graphics).
- Режими рисування.
- Клас Image і афінні перетворення.
- Пояснення до тексту самостійно створеного додатку.