ДЕРЖАВНА СЛУЖБА УКРАЇНИ З НАДЗВИЧАЙНИХ СИТУАЦІЙ

ЛЬВІВСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ БЕЗПЕКИ ЖИТТЄДІЯЛЬНОСТІ

Кафедра інформаційних технологій та систем електронних комунікацій

Р. Р. Головатий, Д. А. Райта, О. В. Хлевной, Ю. С. Назар, О. О. Смотр, Ю. О. Борзов

ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ JAVA

Збірник задач для підготовки до технічних співбесід

(рівень Trainee)

Навчальний посібник

для студентів і курсантів спеціальності 122 «Комп'ютерні науки»
Львівського державного університету безпеки життєдіяльності

Львів — 2026

УДК 004.438(075.8)

Рекомендовано до друку вченою радою Львівського державного університету безпеки життєдіяльності (протокол № ___ від «___» _____ 2026 р.)

Автори:

Головатий Роман Русланович — кандидат технічних наук;
Райта Діана Анатоліївна.
Хлевной Олександр Вікторович — кандидат технічних наук, доцент;
Назар Юлія Сергіївна — доктор філософії;
Смотр Ольга Олексіївна — кандидат технічних наук, доцент;
Борзов Юрій Олексійович — кандидат технічних наук, доцент;

Рецензенти:

Тригуба Анатолій Миколайович — доктор технічних наук, професор, завідувач кафедри інформаційних технологій Львівського національного університету ветеринарної медицини та біотехнологій імені С. З. Гжицького;

Ткачук Ростислав Львович — доктор технічних наук, професор, професор кафедри управління інформаційною безпекою Львівського державного університету безпеки життєдіяльності;

Основи програмування мовою Java : збірник задач для підготовки до технічних співбесід (рівень Trainee) : навч. посіб. / Р. Р. Головатий, Д. А. Райта, О. В. Хлевной, Ю. С. Назар, О. О. Смотр, Ю. О. Борзов — Львів : ЛДУБЖД, 2026. — ___ с.

Збірник задач укладено за матеріалами типових співбесід на позицію Java Trainee та Junior Software Engineer. Посібник призначений для студентів і курсантів спеціальності 122 «Комп'ютерні науки» Львівського державного університету безпеки життєдіяльності, які готуються до професійної діяльності у сфері розробки програмного забезпечення, зокрема інформаційних систем для потреб цивільного захисту. Приклади задач побудовано на предметній області Державної служби України з надзвичайних ситуацій: облік пожежно-рятувальної техніки, реєстрація викликів, чергування караулів, навчальний процес університету.

Кожне з 97 питань збірника містить розширену теоретичну довідку, типовий приклад задачі з повним покроковим розбором та п'ять варіантів індивідуальних завдань для самостійної роботи.

Розглянуто та схвалено на засіданні кафедри інформаційних технологій та систем електронних комунікацій ЛДУБЖД (протокол № ___ від «___» _____ 2026 р.)

УДК 004.438(075.8)

© Головатий Р. Р., Райта Д. А., Хлевной О. В., Назар Ю. С., Смотр О. О., Борзов Ю. О., 2026
© Львівський державний університет безпеки життєдіяльності, 2026

ЗМІСТ

ВСТУП....4
ЧАСТИНА 1. ОСНОВИ МОВИ JAVA....6
Розділ 1. Основи мови Java....6
Розділ 2. Колекції та базове введення-виведення....36
ЧАСТИНА 2. БАГАТОПОТОЧНІСТЬ ТА БАЗИ ДАНИХ....51
Розділ 3. Багатопоточність....51
Розділ 4. Реляційні бази даних і мова SQL....53
Розділ 5. Технологія JDBC....64
ЧАСТИНА 3. ВЕБ-РОЗРОБКА....73
Розділ 6. Об’єктно-реляційне відображення Hibernate / JPA....73
Розділ 7. Протокол HTTP....82
Розділ 8. Технологія сервлетів....85
ЧАСТИНА 4. ДОПОМІЖНІ ТЕХНОЛОГІЇ....95
Розділ 9. Алгоритми, структури даних і безпека....95
Розділ 10. Формати обміну даними....103
Розділ 11. Операційна система Linux....107
Розділ 12. Серіалізація в Java....113
Розділ 13. Логування....118
Розділ 14. Мережне програмування в Java....120
Розділ 15. Веб-сервіси REST і SOAP, контроль версій, збірка, CI....121
ВИСНОВКИ....134
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ....135

ВСТУП

Стрімкий розвиток індустрії розробки програмного забезпечення та цифровізація сфери цивільного захисту висувають нові вимоги до випускників спеціальності 122 «Комп'ютерні науки» Львівського державного університету безпеки життєдіяльності. Випускник, що претендує на початкову позицію Trainee або Junior Software Engineer, повинен не лише володіти теоретичним матеріалом з мови програмування, об'єктно-орієнтованої парадигми, основ роботи з реляційними базами даних, веб-розробки та технології JDBC, а й уміти впевнено застосовувати ці знання у практичних ситуаціях, насамперед під час технічної співбесіди з потенційним роботодавцем [77; 78].

Цей збірник задач укладений за матеріалами типових співбесід, що проводяться провідними технологічними компаніями для кандидатів на позицію Java Trainee. Структура матеріалу охоплює базові розділи мови програмування Java та суміжних технологій: синтаксис та семантика мови, об'єктно-орієнтовані конструкції, обробка винятків, робота з колекціями та потоками введення-виведення, основи багатопоточного програмування, реляційні бази даних та мова SQL, програмний інтерфейс JDBC, об'єктно-реляційне відображення Hibernate/JPA, протокол HTTP, технологія сервлетів, формати обміну даними, основи роботи в Linux, серіалізація, логування, веб-сервіси REST та SOAP, системи контролю версій, системи збірки та неперервної інтеграції.

Виконання індивідуальних завдань збірника передбачає закріплення теоретичних знань, отриманих на лекційних та лабораторних заняттях, формування навичок алгоритмічного мислення, набуття досвіду структурованого пояснення технічних рішень та оцінки складності алгоритмів. Кожне питання збірника супроводжується розширеним теоретичним поясненням, що за глибиною та повнотою відповідає очікуванням технічних інтерв'юерів у провідних компаніях галузі.

Збірник є завершеною методичною розробкою, призначеною для самостійної роботи студентів і курсантів ЛДУБЖД. Предметна область прикладів (пожежно-рятувальні підрозділи, чергова частина, навчальний процес університету) наближена до майбутньої професійної діяльності курсантів і дозволяє одразу бачити практичне застосування мови Java в інформаційних системах цивільного захисту. Посібник може використовуватися як основа для проведення практичних і лабораторних занять, як матеріал для підготовки до іспитів, а також для самостійної підготовки до співбесід під час пошуку першого місця працевлаштування у сфері розробки програмного забезпечення [80].

Структура та методичні рекомендації щодо роботи з матеріалом

Збірник складається з чотирьох тематичних частин, що об'єднують п'ятнадцять розділів і дев'яносто сім питань. Частина 1 «Основи мови Java» охоплює синтаксис та об'єктно-орієнтовані конструкції, обробку винятків, роботу з колекціями та потоками введення-виведення. Частина 2 «Багатопоточність та бази даних» присвячена основам багатопоточного програмування, реляційним базам даних, мові SQL та програмному інтерфейсу JDBC. Частина 3 «Веб-розробка» розглядає об'єктно-реляційне відображення Hibernate/JPA, протокол HTTP та технологію сервлетів. Частина 4 «Допоміжні технології» охоплює алгоритми та структури даних, основи безпеки, формати обміну даними, операційну систему Linux, серіалізацію, логування, мережне програмування, веб-сервіси, системи контролю версій, системи збірки та неперервної інтеграції.

Кожне з питань збірника побудоване за єдиною структурою, що включає три обов'язкові компоненти.

Теоретична частина містить розширене академічне пояснення теми із зазначенням ключових понять, концептуальних особливостей, типових пасток та найкращих практик. Обсяг та глибина теоретичної довідки розраховані на те, щоб здобувач, опрацювавши матеріал, був готовий не тільки відтворити відповідь, але й аргументовано пояснити її

технічному інтерв’юеру, відповісти на додаткові запитання, навести альтернативи та оцінити їх переваги.

Приклад задачі містить умову типового завдання, повну реалізацію з покроковим розбором у вигляді двостовпцевої таблиці (програмний код у лівому стовпці, словесне пояснення у правому). Над кожним лістингом вказано ім’я файлу, у якому розташовується відповідний фрагмент коду, що допомагає здобувачам початкового рівня правильно структурувати власну реалізацію. Приклад слугує еталонним зразком виконання індивідуальних варіантів. Варіанти для самостійної роботи містять п’ять однотипних завдань з різними вхідними умовами, оформлені у вигляді табличного переліку. Передбачається, що здобувач самостійно виконує всі п’ять варіантів за зразком розібраного прикладу. Це забезпечує закріплення теоретичних знань через багаторазове практичне застосування та формує впевненість у самостійній роботі з матеріалом.

Для ефективної підготовки до співбесіди рекомендується дотримуватися такої послідовності роботи з кожним питанням збірника:

1) Уважно опрацювати теоретичну частину, виокремити ключові поняття, скласти короткий конспект власними словами.

2) Самостійно сформулювати очікувану відповідь на питання, не звертаючись до тексту; порівняти з матеріалом збірника, виявити прогалини.

3) Опрацювати розібраний приклад: спершу спробувати розв’язати самостійно, потім порівняти з еталонним розв’язком та покроковим розбором.

4) Виконати п’ять варіантів індивідуальних завдань, оформити код у середовищі розробки (рекомендовано IntelliJ IDEA Community Edition або Eclipse), перевірити його коректність шляхом запуску.

5) Усно проговорити отриманий розв’язок партнерові чи самостійно перед камерою, моделюючи ситуацію реальної співбесіди.

Виконання індивідуальних завдань оформлюється у вигляді звіту або репозиторію у системі контролю версій. Звіт повинен включати: формулювання теми, короткий теоретичний вступ власними словами, умови всіх п’яти варіантів, повний код розв’язків кожного варіанту з коментарями, скріншоти результатів виконання та короткі висновки. Під час самостійної роботи доцільно використовувати сучасні асистенти штучного інтелекту (наприклад, Claude від компанії Anthropic) для пояснення незрозумілих фрагментів коду, пошуку помилок і перевірки власних відповідей; водночас код варіантів здобувач має написати й уміти пояснити самостійно, оскільки на співбесіді доступу до таких інструментів не буде [75; 76; 79]. Список використаних джерел у звіті оформлюють відповідно до ДСТУ 8302:2015 [81].

Контроль засвоєння матеріалу здійснюється шляхом перевірки виконання варіантів, проведення усної співбесіди викладачем у форматі, наближеному до реальної технічної співбесіди, та підсумкового тестування. Здобувач вважається таким, що засвоїв матеріал розділу, якщо здатний без використання додаткових джерел відповісти на 80% теоретичних питань відповідного розділу та виконати індивідуальне завдання за наданим зразком.

Насамкінець — одне побажання від авторів. Не намагайтеся «пройти» збірник за тиждень. Краще одне питання на день, але чесно, з власним кодом і власними помилками.

ЧАСТИНА 1. ОСНОВИ МОВИ JAVA

Розділ 1. Основи мови Java

Питання 1. Різниця між класом (class) та об’єктом (object)

Теоретична частина

У парадигмі об’єктно-орієнтованого програмування клас (class) визначається як абстрактний тип даних — формальний опис властивостей та поведінки, тоді як об’єкт (object, instance) є його матеріалізованим втіленням у пам’яті програми. Клас можна порівняти з кресленням будівлі: воно існує як план, але не є самою будівлею. Об’єкт же — конкретна будівля, споруджена за цим кресленням. У мові Java клас оголошується ключовим словом class та містить поля (fields), методи (methods), конструктори (constructors) і внутрішні типи. Об’єкт створюється оператором new, який виконує три дії: виділяє пам’ять у купі (heap), викликає конструктор для ініціалізації полів, повертає посилання (reference) на створений екземпляр [2; 6].

Принципово важливо розуміти: на основі одного класу може бути створено необмежену кількість об’єктів, кожен з яких матиме власні значення нестатичних полів, але всі вони поділятимуть один спільний байт-код методів. Змінна, оголошена типом класу (наприклад, FireTruck truck), — це не сам об’єкт, а посилання (адреса) на нього в пам’яті JVM. Звідси відома пастка: присвоєння truck2 = truck1 копіює посилання, але не сам об’єкт — обидві змінні вказують на той самий екземпляр.

Клас існує під час компіляції як тип у системі типів Java, а об’єкт існує лише під час виконання (runtime) як ділянка пам’яті у купі. Тому, наприклад, інструкції class Foo {} та new Foo() належать до різних фаз життя програми.

Якщо після цього абзацу різниця все ще здається абстрактною — не страшно. На першому курсі її майже ніхто не відчуває одразу; розуміння приходить після кількох власних класів.

Приклад задачі

Умова. Описати клас FireTruck (модель пожежного автомобіля, рік випуску) для автопарку навчальної пожежно-рятувальної частини Львівського державного університету безпеки життєдіяльності (ЛДУБЖД), створити три об’єкти з різними значеннями, продемонструвати, що модифікація стану одного не впливає на інші.

Завдання потребує двох файлів: окремий файл з оголошенням класу та файл з методом main, у якому створюються об’єкти.

Лістинг 1.1. Реалізація з покроковим розбором

Файл `FireTruck.java` окремий файл з оголошенням класу

```
class FireTruck {
    String model;
    int year;
}
```

Окремий файл FireTruck.java містить лише оголошення класу FireTruck з двома нестатичними полями. Ім’я файлу обов’язково має збігатися з ім’ям класу. Це лише опис типу — жодного пожежного автомобіля в пам’яті ще немає.

Файл `Main.java` точка входу програми, метод main

```
public class Main {
  public static void main(String[] args) {
    FireTruck c1 = new FireTruck();
    c1.model = "АЦ-40";
    c1.year  = 2021;
```

У файлі Main.java оголошуємо клас Main з методом main — стандартна точка входу Java-програми. Створюємо перший об’єкт (автоцистерну АЦ-40) у купі та задаємо його полям конкретні значення. Змінна c1 — посилання на цей об’єкт.

```
    FireTruck c2 = new FireTruck();
    c2.model = "АД-30";
    c2.year  = 2018;

    FireTruck c3 = new FireTruck();
    c3.model = "АР-2";
    c3.year  = 2023;
```

Створюємо ще два незалежні об’єкти — автодрабину АД-30 та рукавний автомобіль АР-2. Поля c2 і c3 не пов’язані з полями c1 — кожен об’єкт займає свою ділянку купи. Це фундаментальна властивість роботи з об’єктами в Java.

```
    System.out.println(
        c1.model + " " +
        c2.model + " " +
        c3.model);
  }
}
```

Вивід у консоль: АЦ-40 АД-30 АР-2. Кожен об’єкт зберіг своє значення поля model — це підтверджує, що три об’єкти повністю незалежні один від одного.

Зверніть увагу. *Спробуйте після виведення змінити c2.model = "АЦ-5" і ще раз надрукувати всі три моделі. Якщо зміна торкнулася лише одного об’єкта — ви правильно зрозуміли різницю між класом і об’єктом.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу описати клас з вказаними полями в окремому файлі. У методі main файлу Main.java створити три об’єкти з різними значеннями та вивести їх стан у консоль, щоб продемонструвати їх незалежність.

№	Назва класу (файл)	Поля
1	FireStation.java	number, address, crewSize
2	Cadet.java	name, platoon, gpa
3	Extinguisher.java	type, mass
4	Hydrant.java	x, y
5	Rescuer.java	name, rank, experience

Питання 2. Різниця між нестатичними та статичними полями. Чи може локальна змінна бути static?

Теоретична частина

Поле класу в Java належить або до екземпляра (instance field, нестатичне), або до самого класу (static field, статичне). Нестатичне поле зберігається в межах кожного об’єкта окремо: створено n об’єктів — існує n незалежних копій цього поля. Доступ здійснюється через посилання на об’єкт: obj.field [2; 14].

Статичне поле, оголошене з модифікатором static, належить класу як цілому. Незалежно від кількості створених об’єктів, у пам’яті JVM існує лише одна копія такого поля, яка ініціалізується один раз під час завантаження класу завантажувачем класів (ClassLoader).

Доступ до неї відбувається через ім’я класу: ClassName.field. Типове застосування — константи, лічильники, спільна конфігурація, кеші.

Існує також тонкий нюанс: статичне поле зберігається у спеціальній області пам’яті — метаданих класу (Metaspace у HotSpot з Java 8+), тоді як нестатичні поля зберігаються в купі разом з об’єктом-власником. Локальна змінна (local variable), оголошена всередині методу, не може мати модифікатора static. Причина: модифікатор static має сенс лише для членів класу — він керує тим, належить член класу чи екземпляру. Локальні ж змінні розташовуються у стеку викликів, створюються при виклику методу і знищуються при виході з нього. Концепція «єдиної копії, спільної для всіх викликів» до них не застосовна. Спроба написати static int x; у тілі методу призведе до помилки компіляції «illegal start of expression».

Приклад задачі

Умова. Створити клас FireCall (виклик, що надійшов до чергової частини), у якому є статичне поле total (загальна кількість зареєстрованих викликів) та нестатичне поле id (унікальний номер кожного виклику). Зареєструвати три виклики та вивести їх id і загальну кількість.

Реалізація — у двох файлах: FireCall.java з описом класу та Main.java з демонстрацією роботи статичного лічильника викликів.

Лістинг 2.1. Реалізація з покроковим розбором

Файл `FireCall.java` клас зі статичним лічильником

```
class FireCall {
    static int total = 0;
    int id;

    FireCall() {
        total++;
        this.id = total;
    }
}
```

Поле total оголошене з модифікатором static — воно одне на весь клас і доступне через ім’я FireCall.total. Поле id індивідуальне для кожного виклику. Конструктор інкрементує total і присвоює нове значення полю id.

Файл `Main.java` реєстрація трьох викликів

```
public class Main {
  public static void main(String[] args) {
    FireCall a = new FireCall();
    FireCall b = new FireCall();
    FireCall c = new FireCall();
```

Реєструємо три виклики FireCall. Кожен виклик конструктора виконує total++ — тому після трьох викликів поле total дорівнює 3. Поля a.id, b.id, c.id отримують значення 1, 2, 3 відповідно.

```
    System.out.println(a.id + " " + b.id + " " +
c.id);
    System.out.println(FireCall.total);
  }
}
```

Вивід: «1 2 3» та «3». Доступ до total відбувається через ім’я класу без створення екземпляра — це підтверджує, що статичне поле належить класу, а не об’єкту.

Зверніть увагу. *Типова помилка — звертатися до статичного поля через об'єкт: a.total. Компілятор це дозволить, але читач коду подумає, що total належить саме виклику a. Пишіть FireCall.total.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу описати клас із вказаним статичним та нестатичним полем у окремому файлі. У методі main файлу Main.java створити три об'єкти, продемонструвати, що статичне поле спільне для всіх, а нестатичне — індивідуальне.

№	Назва класу (файл)	Статичне поле	Нестатичне поле
1	FireStation.java	city	stationNumber
2	Cadet.java	university	name
3	DutyLog.java	recordsCount	shiftName
4	Dispatcher.java	emergencyPhone, region	operatorName
5	ReportIdGenerator.java	nextReportId	createdAt

Питання 3. Модифікатори доступу. final для змінних. Примітивні типи. Autoboxing. Масиви

Теоретична частина

Мова Java реалізує принцип інкапсуляції через чотири рівні доступу до членів класу. Модифікатор private обмежує видимість межами одного класу, package-private (відсутність модифікатора) — межами поточного пакета, protected — пакетом і класами-нащадками (навіть з інших пакетів), public — необмежено. Чим вужча видимість, тим легше підтримувати інваріанти класу та змінювати реалізацію без впливу на клієнтський код. На заняттях ми радимо курсантам просте правило: усе, що можна зробити private, робіть private.

Модифікатор final у застосуванні до змінної означає, що її значення може бути присвоєне рівно один раз. Для примітивної змінної це створює фактичну константу. Для змінної-посилання забороняється переприсвоєння самого посилання, проте не забороняється зміна стану об'єкта, на який воно вказує: final List l = new ArrayList(); l.add("x"); — валідний код [1; 14].

Java підтримує вісім примітивних типів: цілочисельні byte (8 біт), short (16), int (32), long (64), числа з плаваючою комою float (32) та double (64), логічний boolean і символьний char (16 біт, Unicode). Примітиви зберігаються за значенням, не мають методів, не можуть бути null. Кожному примітиву відповідає клас-обгортка (wrapper class): Integer, Long, Double, Boolean тощо. Autoboxing — це автоматичне перетворення примітива в обгортку (int → Integer), а unboxing — зворотне. Виконується компілятором прозоро. Однак ці операції мають накладні витрати: створюється новий об'єкт у купі. Окремий ризик — NullPointerException при unboxing значення null: Integer x = null; int y = x; кине NPE.

Масив (array) у Java — це особливий об'єкт фіксованої довжини, що зберігає послідовність елементів одного типу. Оголошення: int[] a = new int[5]; або int[] a = {1, 2, 3};. Доступ за індексом: a[i]. Довжина: a.length (це поле, не метод). Масиви індексуються з нуля; спроба доступу за неприпустимим індексом призводить до ArrayIndexOutOfBoundsException.

Приклад задачі

Умова. Створити клас Cadet (курсант університету), що поєднує всі чотири концепції: приватне final-поле (особистий номер), autoboxing/unboxing, масив з трьох цілих (бали за нормативи), демонстрація рівнів доступу.

Усі демонстрації виконуються в одному файлі Cadet.java, що містить точку входу main.

Лістинг 3.1. Реалізація з покроковим розбором

Файл `Cadet.java` усі чотири концепції в одному класі	

`public class Cadet {`
`  private final int id = 1042;`
`  protected String name = "Петренко";`
	Поле id (особистий номер курсанта) оголошене з модифікаторами private та final — видиме тільки в цьому класі, значення присвоюється один раз. Поле name з модифікатором protected — доступне нащадкам та класам у тому ж пакеті.

`  public static void main(String[] args) {`
`  int prim = 5;`
`  Integer boxed = prim;`
`  int unboxed = boxed;`
	Конструкція Integer boxed = prim; — це autoboxing: компілятор підставляє Integer.valueOf(prim). Зворотне int unboxed = boxed; — unboxing через boxed.intValue().

`  int[] arr = {85, 92, 78};`
`  System.out.println(arr.length);`
`  System.out.println(arr[1]);`
`  }`
`}`
	Масив балів за три нормативи з пожежно-стройової підготовки створюється літералом. Поле arr.length — публічне final-поле масиву (не метод!), повертає 3. Доступ за індексом 1 повертає 92 — індексація з нуля.

Порада. *Поле id оголошене як final, тому рядок id = 2000; усередині main не скомпілюється. Перевірте це самостійно — повідомлення компілятора корисно побачити хоча б раз.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити клас з усіма чотирма концепціями. Реалізувати масив із вказаною кількістю цілих чисел (дані чергової частини або навчального процесу) та обчислити три величини, які підкреслюють тему варіанту. Вивести результати у консоль.

№	Назва final-поля	Розмір масиву	Дані масиву та що обчислити
1	MAX_CREW	5	кількість викликів за 5 днів: сума, мінімум, максимум
2	MAX_HOSE_LENGTH	7	довжини рукавних ліній, м: сума, середнє, кількість парних
3	MAX_WATER_TANK	6	витрата води за виїзд, л: сума, добуток ненульових, максимум
4	MAX_ALARM_RANK	8	зміна температури у приміщенні: середнє, мінімум, кількість від'ємних
5	MAX_SCORE	10	бали курсантів взводу: сума, максимум, кількість більших за середнє

Питання 4. Як працюють оператори =, == і !=

Теоретична частина

Оператор = — оператор присвоєння (assignment operator). Зліва стоїть змінна (lvalue), справа — значення (rvalue). Для змінних примітивного типу копіюється саме значення. Для змінних посилального типу копіюється посилання — тобто адреса об'єкта в купі, а не сам об'єкт. Це фундаментальне джерело багатьох пасток: a = b при посилальних типах означає, що тепер дві змінні вказують на той самий об'єкт [10].

Оператор == — оператор порівняння на рівність. Семантика залежить від типу операндів. Для примітивів порівнюються значення. Для посилальних типів порівнюються посилання, тобто перевіряється, чи вказують обидві змінні на той самий об'єкт у пам'яті. Звідси класична пастка: new String("a") == new String("a") поверне false, бо це різні об'єкти, попри однаковий вміст. Оператор != — логічне заперечення ==. Семантика симетрична.

Для порівняння вмісту посилальних типів використовується метод equals(Object), успадкований від Object. За замовчуванням Object.equals реалізує порівняння посилань (як ==), однак класи на кшталт String, Integer, BigDecimal, List, Map перевизначають його для порівняння за значенням.

Особливий випадок — Integer-кеш: значення в діапазоні від -128 до 127 кешуються JVM, тому Integer a = 100; Integer b = 100; a == b → true. Але для значень поза кешем (наприклад, 200) — false. Ця тонкість часто з'являється на співбесідах як перевірка глибини знань.

Окремо зауважимо: питання про == для рядків — мабуть, найпопулярніше на співбесідах Trainee-рівня. Готуйтеся до нього ґрунтовно.

Приклад задачі

Умова. Продемонструвати класичну пастку зі String-літералами та new String, де == та equals дають різні результати. Створити три рядки з назвою служби «ДСНС» трьома способами та виконати попарне порівняння.

Реалізація в одному файлі Demo.java. Демонструє різницю між порівнянням посилань та порівнянням вмісту.

Лістинг 4.1. Реалізація з покроковим розбором

Файл Demo.java порівняння рядків трьома способами

```
public class Demo {
  public static void main(String[] args) {
    String s1 = "ДСНС";
    String s2 = "ДСНС";
    String s3 = new String("ДСНС");
```

Літерали "ДСНС" автоматично потрапляють у String Pool — спеціальну область пам’яті, де JVM зберігає унікальні рядки-літерали. Тому s1 та s2 — це одне й те саме посилання. Конструкція new String("ДСНС") явно створює новий об’єкт у купі.

```
    System.out.println(s1 == s2);
    System.out.println(s1 == s3);
    System.out.println(s1.equals(s3));
  }
}
```

Вивід: true, false, true. Перше порівняння — те саме посилання у пулі. Друге — різні об’єкти. Третє — equals порівнює послідовність символів і повертає true.

Зверніть увагу. *Якщо замінити третій рядок на String s3 = new String("ДСНС").intern();, порівняння s1 == s3 поверне true. Подумайте, чому.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити три об’єкти вказаного типу різними способами та виконати три попарні порівняння через == і equals(). Пояснити в коментарях, чому отримано саме такі результати.

№	Тип значення	Тестові значення
1	Integer	200 (номер частини, autobox), 200 (autobox), new Integer(200)
2	Integer	101 (номер виклику ДСНС, autobox), 101 (autobox), new Integer(101)
3	String	"рятувальник", "рятувальник", new String("рятувальник")
4	Double	0.1 + 0.2 (тиск у рукаві, МПа), 0.3, Double.valueOf(0.3)
5	ArrayList	три списки номерів караулів [1, 2, 3], створені різними способами

Питання 5. Різниця між постфіксним value++ та префіксним ++value інкрементами

Теоретична частина

Обидва оператори здійснюють однакову дію над змінною — збільшують її значення на одиницю. Різниця полягає у тому, яке значення повертає вираз. Постфіксний інкремент value++ повертає СТАРЕ значення змінної, після чого виконується саме збільшення. Префіксний інкремент ++value спочатку збільшує змінну, після чого повертає НОВЕ значення. Якщо оператор використовується як окремий стейтмент (наприклад, a++;), різниці немає — змінна збільшується, повернене значення не використовується. Різниця проявляється тоді, коли результат виразу присвоюється змінній, передається аргументом до методу або використовується в умові [10; 14].

Класична пастка співбесіди: int i = 0; i = i++; — після виконання значення і дорівнює нулю, не одиниці.

Причина: постфіксний інкремент повертає старе значення (0), яке потім присвоюється і, перезаписуючи інкрементоване нове значення.

Аналогічна семантика діє для оператора декременту: -- ставиться до або після змінної з відповідною поведінкою. Для примітивних типів у Java різниці у швидкодії між формами немає — обирати слід ту, що робить вираз зрозумілішим.

Приклад задачі

Умова. Продемонструвати різницю поведінки префіксної та постфіксної форми інкременту на прикладі лічильника виїздів пожежно-рятувальної частини за зміну. Використати дві окремі змінні: одна — постфіксно, інша — префіксно.

Реалізація в одному файлі Demo.java. Простий приклад, що візуально показує різницю значень після виконання обох операторів.

Лістинг 5.1. Реалізація з покроковим розбором

Файл `Demo.java` порівняння постфіксної та префіксної форм

```
public class Demo {
  public static void main(String[] args) {
    int a = 5; // виїздів
    int b = a++;
    System.out.println(a + " " + b);
```

Нехай a — кількість виїздів за зміну. Вираз a++ обчислюється у послідовності: повертає старе значення a (5), потім інкрементує a до 6. Тому b отримує 5, а a стає 6. Вивід: «6 5».

```
    int c = 5;
    int d = ++c;
    System.out.println(c + " " + d);
  }
}
```

Вираз ++c навпаки: спочатку інкрементує c до 6, потім повертає нове значення 6. Тому і c, і d отримують значення 6. Вивід: «6 6». Це підтверджує, що пара (++value) повертає вже інкрементоване значення.

Зверніть увагу. *Вираз a = a++ — улюблене питання інтерв'юерів. Не пишіть так у робочому коді: навіть досвідчені розробники витрачають час, щоб зрозуміти результат.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати програму, що демонструє різницю між постфіксною та префіксною формами інкременту (лічильники викликів, рукавів, курсантів у строю тощо). Використати змінні зі стартовим значенням з таблиці; обчислити вираз з таблиці; вивести значення змінних та результат виразу.

№	Стартові значення	Вираз для обчислення
1	calls = 0	calls = calls++ (продемонструвати calls = 0)
2	hoses = 1, crew = 1	x = hoses++ + ++crew
3	trucks = 5, cadets = 5	x = trucks++ + ++cadets
4	floor = 10	у методі evacuate(floor++) показати значення в методі та після
5	i = 0	у циклі while(i++ < 5) вивести номер рятувальника i; пояснити останнє значення

Питання 6. Як працює оператор switch

Теоретична частина

Оператор switch є інструкцією багатоальтернативного вибору. Він обчислює вираз і передає управління на мітку case з відповідним значенням. Якщо жоден case не збігається, виконується гілка default (за наявності). Класичний switch у Java історично підтримував лише byte, short, int, char (та їхні обгортки); з Java 5 додано enum-типи, з Java 7 — String, з Java 14 — switch-вирази, з Java 21 — pattern matching [2; 7].

Тут ховається, мабуть, найпідступніша пастка цього оператора. Ключова особливість традиційного switch — fall-through: після виконання тіла case управління «провалюється» до наступного case, поки не зустріне break або кінець блоку switch. Це часте джерело помилок: забутий break призводить до виконання кількох гілок поспіль. Іноді fall-through використовують свідомо для об'єднання обробників, але такі випадки потребують коментаря.

Сучасний синтаксис switch-вираз (Java 14+) використовує стрілку: case A -> ...; — він не має fall-through (виконує лише вказаний блок), повертає значення і трактується як вираз. Для багаторядкових тіл використовується yield. У Java 21 додано pattern matching у switch, що дозволяє розгалуження за типом. Концептуально switch — це структурована альтернатива до if-else-if з фіксованим набором значень. Сучасний компілятор оптимізує switch у спеціальні інструкції байт-коду (tableswitch або lookupswitch), які працюють швидше за послідовність if.

Приклад задачі

Умова. Реалізувати визначення рівня пожежної небезпеки за номером (1–3) двома способами: класичний switch з break та сучасний switch-вираз.

Обидва способи демонструються в одному файлі Demo.java у методі main.

Лістинг 6.1. Реалізація з покроковим розбором

Файл `Demo.java` класичний switch із break

```
int level = 3;
String name;
switch (level) {
    case 1: name = "Низький"; break;
    case 2: name = "Середній"; break;
    case 3: name = "Високий"; break;
    default: name = "Невідомо";
}
```

Класична форма потребує явного break після кожного case для уникнення fall-through. Для level=3 виконається третя гілка, name отримає значення "Високий". Без break управління «провалилося» б у наступні case.

Файл `Demo.java` сучасний switch-вираз (Java 14+)

```
String s = switch (level) {
    case 1 -> "Низький";
    case 2 -> "Середній";
    case 3 -> "Високий";
    default -> "Невідомо";
};
System.out.println(s);
```

Сучасна форма зі стрілкою -> не має fall-through: виконується лише потрібна гілка. switch тут є виразом — повертає значення, яке присвоюється змінній s. Це усуває ризик забутого break.

З досвіду викладачів. *Зверніть увагу: у сучасній формі switch-вираз зобов'язаний охопити всі можливі значення, тому гілка default тут обов'язкова. Для enum її можна опустити, якщо перелічено всі константи.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати switch у файлі Demo.java, що за вхідним значенням повертає відповідний результат. Реалізувати обома способами: класичним switch з break та сучасним switch-виразом.

№	Вхідне значення	Що повертає switch
1	клас пожежі (A, B, C, D, F)	вид горючих речовин
2	оцінка курсанта (1–5)	характеристика: "незадовільно"..."відмінно"
3	enum рівня тривоги	дія чергового караулу
4	номер місяця	пора року та сезонна пожежна небезпека (метод season)
5	тип вогнегасника ("ВП", "ВВК", "ВВП")	вогнегасна речовина та класи пожеж, які ним гасять

Питання 7. Як працює оператор for-each. Як зробити клас придатним для for-each

Теоретична частина

Оператор for-each (enhanced for, Java 5+) є спрощеним синтаксисом для ітерації по масивах та об’єктах, що реалізують інтерфейс Iterable. Синтаксис: for (Type element : iterable) { ... }. На етапі компіляції цей цикл перетворюється на еквівалентний код з використанням Iterator (для колекцій) або класичного індексованого циклу (для масивів) [8].

for-each приховує деталі ітерації — індекс або стан курсора.

Це робить код лаконічним та безпечним. Однак це накладає й обмеження: немає доступу до індексу елемента, неможливо видаляти елементи з колекції під час ітерації (це призведе до ConcurrentModificationException).

Щоб клас можна було використовувати у for-each, він має реалізувати інтерфейс java.lang.Iterable<T>. Цей інтерфейс містить один абстрактний метод: Iterator<T> iterator(). Інтерфейс Iterator<T>, своєю чергою, вимагає реалізації методів boolean hasNext() та T next(); опційно — void remove() та forEachRemaining(Consumer). Компілятор перетворює цикл for (T t : collection) на еквівалент while (it.hasNext()) { T t = it.next(); ... } з прихованим створенням ітератора. Кожен виклик iterable.iterator() створює новий незалежний курсор.

Приклад задачі

Умова. Створити власний клас FloorRange, що ітерує номери поверхів від from до to (виключно) для обходу будівлі рятувальниками, і використати його у циклі for-each.

Клас FloorRange реалізує Iterable у файлі FloorRange.java; використання — у файлі Main.java.

Лістинг 7.1. Реалізація з покроковим розбором

Файл `FloorRange.java` власний клас з реалізацією Iterable

```
import java.util.Iterator;

class FloorRange implements Iterable<Integer> {
    private final int from, to;
    FloorRange(int f, int t) { from = f; to = t; }
```

Клас FloorRange реалізує інтерфейс Iterable<Integer> — це обов'язкова умова, щоб об'єкт можна було використати у for-each. Поля from і to задають межі діапазону поверхів.

```
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            int cur = from;
            public boolean hasNext() {
                return cur < to;
            }
            public Integer next() {
                return cur++;
            }
        };
    }
}
```

Метод iterator() повертає анонімний Iterator. Поле cur зберігає стан курсора. hasNext() перевіряє, чи не досягнуто межі to. next() повертає поточне значення і просуває курсор (постфіксний інкремент).

Файл `Main.java` використання у for-each

```
public class Main {
  public static void main(String[] args) {
    for (int x : new FloorRange(1, 5)) {
        System.out.print(x + " ");
    }
  }
}
```

Цикл for-each прозоро використовує iterator(). Вивід: «1 2 3 4» — рятувальники послідовно обстежують поверхи. Число 5 не виводиться, бо діапазон напіввідкритий (to виключно). Компілятор сам викликає hasNext() та next().

Зверніть увагу. *Якщо в циклі for-each спробувати видаляти елементи з колекції, отримаємо ConcurrentModificationException. Для FloorRange це не загрожує, бо клас не зберігає колекції — він обчислює значення «на льоту».*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити власний клас, що реалізує Iterable<Integer>, у окремому файлі. У файлі Main.java продемонструвати ітерацію через for-each.

№	Назва класу (файл)	Що ітерує
1	EvenFloors.java	парні номери поверхів від 0 до max
2	Fibonacci.java	перші n чисел Фібоначчі (умовне зростання площі пожежі)
3	Squares.java	площі квадратних приміщень зі стороною від 1 до n
4	Countdown.java	зворотний відлік від n до 1 перед виїздом караулу
5	PowersOfTwo.java	степені двійки до заданої межі (модель поширення вогню)

Питання 8. Оголошення класу. Наслідування. Реалізація інтерфейсу. final/abstract. Статичні та нестатичні методи

Теоретична частина

Клас оголошується ключовим словом class: [модифікатори] class Name [extends Super] [implements I1, I2]. В одному файлі може бути лише один public-клас верхнього рівня, причому ім'я файлу має збігатися з його іменем. Наслідування здійснюється через extends. Java підтримує тільки одиничне наслідування класів, але клас може реалізовувати довільну кількість інтерфейсів через implements [1; 17; 18].

Інтерфейс оголошується ключовим словом interface. До Java 8 інтерфейс міг містити лише абстрактні методи й константи; з Java 8 додано default-методи та static-методи; з Java 9 — private-методи. Усі методи інтерфейсу за замовчуванням public, поля — public static final.

Модифікатор final у класі забороняє наслідування (приклад — String). final-метод не може бути перевизначений у нащадку. Модифікатор final не застосовується до інтерфейсу. Модифікатор abstract означає, що клас не може бути інстанційований напряму (new AbstractFoo() — помилка компіляції). Абстрактний клас призначений для наслідування і може містити як абстрактні методи (без тіла), так і конкретні. Інтерфейс за замовчуванням абстрактний.

Статичний метод належить класу, а не екземпляру: викликається через ClassName.method(). Не має доступу до this. Нестатичний (instance) метод викликається через посилання на об'єкт і має доступ до this. Статичні методи не перевизначаються — лише приховуються у нащадку.

Запам'ятати все це з першого разу важко, і це нормально. Найкраще працює практика: спробуйте навмисно написати new на абстрактному класі й подивіться, що скаже компілятор.

Приклад задачі

Умова. Створити ієрархію: абстрактний клас FireVehicle (пожежний автомобіль) реалізує інтерфейс Deployable (вивід на позицію); конкретний клас Tanker (автоцистерна) наслідує FireVehicle. Додати статичний метод.

Інтерфейс, абстрактний клас та конкретний клас — кожен у своєму файлі; демонстрація у Main.java.

Лістинг 8.1. Реалізація з покроковим розбором

Файл `Deployable.java, FireVehicle.java` інтерфейс та абстрактний клас

```
interface Deployable {
    void deploy();
}

abstract class FireVehicle implements Deployable {
    abstract double capacity();
    static String type() {
        return "FireVehicle";
    }
}
```

Deployable — інтерфейс з одним абстрактним методом. FireVehicle — абстрактний клас: реалізує Deployable, але залишає deploy() та capacity() абстрактними. Метод type() статичний — викликається через FireVehicle.type() без створення об'єкта.

Файл `Tanker.java` конкретний клас-нащадок

```
class Tanker extends FireVehicle {
    double v;
    Tanker(double v) { this.v = v; }
    double capacity() {
        return v * 1000;
    }
    public void deploy() {
        System.out.println("АЦ на позиції");
    }
}
```

Tanker наслідує FireVehicle (extends) і є конкретним: реалізує обидва абстрактні методи — capacity() (місткість цистерни в літрах) та deploy(). Тепер new Tanker(4.0) дозволено, тоді як new FireVehicle() призвело б до помилки компіляції.

Зверніть увагу. *Поміркуйте, чи можна додати в інтерфейс Deployable метод з реалізацією. Можна — це default-метод (Java 8+). Але зловживати цим не варто: інтерфейс має описувати контракт, а не логіку.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити інтерфейс, абстрактний клас та конкретний клас-нащадок (кожен у своєму файлі). У Main.java створити об'єкти та викликати методи.

№	Інтерфейс / абстр. клас	Конкретні класи-нащадки
1	Rescuer (метод rescue)	Firefighter, Diver
2	Equipment (метод check)	Extinguisher, BreathingApparatus
3	Alarm (метод notify)	SmokeAlarm, HeatAlarm
4	UniversityStaff (метод work)	Lecturer, Instructor
5	Room (метод evacuationTime)	Classroom, Dormitory

Питання 9. Що таке сигнатура методу

Теоретична частина

Сигнатура методу (method signature) в Java — це сукупність імені методу та впорядкованого списку типів його формальних параметрів. До сигнатури НЕ входять: тип повернення (return type), імена параметрів, модифікатори доступу, клауза throws, generic-параметри (з певними винятками, пов'язаними зі стиранням типів). Сигнатура використовується компілятором та JVM для розрізнення методів. Це базис механізму перевантаження (method overloading): у класі може бути кілька методів з однаковим іменем, якщо їхні сигнатури відрізняються. Наприклад, void log(int) та void log(String) — різні методи. Натомість int sum(int) та double sum(int) — конфлікт: однакова сигнатура, різний тип повернення; помилка компіляції [14].

При перевизначенні методу (override) у класі-нащадку сигнатура методу-нащадка має точно збігатися з сигнатурою предка. Тип повернення може бути коваріантним (підтипом оригіналу). Анотація @Override допомагає компілятору перевірити коректність перевизначення.

Окремо варто згадати концепцію erasure: при компіляції generic-параметри стираються. Через це методи void process(List<String>) і void process(List<Integer>) у одному класі є помилкою — їхні сигнатури після стирання збігаються.

Приклад задачі

Умова. Створити клас Dispatcher (диспетчер оперативно-рятувальної служби) з трьома перевантаженими методами send, що мають різні сигнатури, і викликати кожен.

Клас Dispatcher у файлі Dispatcher.java; демонстрація викликів — у Main.java.

Лістинг 9.1. Реалізація з покроковим розбором

Файл `Dispatcher.java` три перевантажені методи send

```
class Dispatcher {
    void send(int x) {
        System.out.println("частина: " + x);
    }
    void send(String s) {
        System.out.println("адреса: " + s);
    }
    void send(int x, int y) {
        System.out.println(x + "," + y);
    }
}
```

Три методи з однаковим іменем send, але різними сигнатурами: send(int) — номер частини, send(String) — адреса виклику, send(int, int) — частина та кількість відділень. Компілятор розрізняє їх за типом і кількістю параметрів — це перевантаження (overloading).

Файл `Main.java` виклик кожного методу

```
public class Main {
  public static void main(String[] args) {
    Dispatcher p = new Dispatcher();
    p.send(5);
    p.send("Клепарівська, 35");
    p.send(1, 2);
  }
}
```

Компілятор обирає конкретний метод на етапі компіляції за статичними типами аргументів (static dispatch). Вивід: «частина: 5», «адреса: Клепарівська, 35», «1,2». Кожен виклик потрапляє у відповідне перевантаження.

Порада. *Спробуйте додати до класу метод int send(int x). Компілятор видасть помилку: сигнатура send(int) уже існує, а тип повернення до сигнатури не входить.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити клас із трьома перевантаженими методами (різні сигнатури) у окремому файлі. У Main.java викликати кожне перевантаження та переконатися у правильному виборі компілятора.

№	Назва класу (файл)	Перевантаження методу
1	WaterCalculator.java	flow(int,int), flow(double,double), flow(int,int,int)
2	ReportFormatter.java	format(int), format(double), format(String)
3	DutyLogger.java	log(String), log(int), log(Throwable)
4	CallSign.java	join(String,String), join(String,int), join(int,String)
5	RoomArea.java	area(int), area(int,int), area(double)

Питання 10. Передача параметрів за значенням vs за посиланням

Теоретична частина

Коротко: Java завжди передає параметри за значенням. Крапка.

Java беззастережно передає всі параметри методам ЗА ЗНАЧЕННЯМ (pass-by-value). Розрізнення тут тонке: для змінних примітивного типу копіюється сама величина. Для змінних посилального типу копіюється ЗНАЧЕННЯ ПОСИЛАННЯ — тобто адреса об’єкта в купі. У результаті всередині методу маємо локальну змінну, що вказує на той самий об’єкт, що й у викликача.

Наслідки: метод не може змінити змінну виклику (переприсвоїти її іншим значенням), але може змінити стан об’єкта, на який ця змінна вказує. Якщо всередині методу написати arg = new Foo() — це локальне перевизначення, на змінну виклику воно не вплине. Якщо ж arg.field = 1 — викликач побачить зміну [2; 5].

Це принципово відрізняється від pass-by-reference (як у C++ з оператором &). У pass-by-reference переприсвоєння параметра всередині методу впливало б на оригінальну змінну. У Java це неможливо. Метод swap(a, b) ніколи не змінить значення оригінальних змінних викликача. Окрема пастка стосується immutable-об’єктів (String, Integer): навіть якщо передати їх у метод, всередині неможливо змінити їхній стан — у класів немає мутаторів. Тому всі операції над ними створюють нові об’єкти.

Приклад задачі

Умова. Продемонструвати на прикладі рукава пожежної автоцистерни, що метод може змінити стан переданого об’єкта, але не може перевизначити саму змінну посилання у викликача.

Клас Hose (пожежний рукав) у файлі Hose.java; метод-модифікатор та демонстрація — у Main.java.

Лістинг 10.1. Реалізація з покроковим розбором

Файл `Hose.java, Main.java` клас-контейнер та метод модифікації

```
class Hose { int v; } // довжина, м

public class Main {
  static void mod(Hose b, int x) {
    b.v = x;
    b = new Hose();
    b.v = 999;
  }
```

Метод mod приймає посилання на Hose. Рядок b.v = x; змінює поле спільного об’єкта — це буде видно ззовні. Рядок b = new Hose(); створює новий об’єкт, але локальна b більше не пов’язана з оригіналом викликача.

```
  public static void main(String[] args) {
    Hose box = new Hose();
    mod(box, 20);
    System.out.println(box.v);
  }
}
```

Викликаємо mod(box, 20). Вивід: «20», а не «999». Зміна поля (b.v=20) видима, бо це той самий об’єкт. Створення нового Hose усередині методу не вплинуло на змінну box викликача — підтвердження pass-by-value.

Зверніть увагу. *Цей приклад варто розібрати на папері: намалюйте дві змінні-«стрілки» box і b та один об’єкт Hose. Після рядка b = new Hose() стрілка b просто «переїжджає» на новий об’єкт.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати метод, що приймає параметр, та продемонструвати поведінку передачі за значенням. У Main.java викликати метод і вивести результат, що ілюструє тему варіанту.

№	Метод	Що продемонструвати
1	swap(int crewA, int crewB)	оригінальні змінні (склад двох караулів) НЕ міняються
2	swap(int[] posts)	елементи масиву постів МІНЯЮТЬСЯ місцями
3	clear(List<String> calls)	список викликів спорожнів у викликача
4	increment(Integer trips)	кількість виїздів НЕ змінилася поза методом
5	append(StringBuilder report)	оригінальний текст донесення ЗМІНИВСЯ

Питання 11. Що таке конструктор. Скільки конструкторів. Чи може бути static. Чи може мати тип повернення

Теоретична частина

Конструктор (constructor) — спеціальний блок коду, що виконується при створенні нового об’єкта оператором new. Його ім’я завжди збігається з іменем класу, він не має оголошеного типу повернення (навіть void), не успадковується нащадками. Основне призначення — ініціалізація полів нового об’єкта [7; 19].

Якщо програміст не оголосив жодного конструктора, компілятор автоматично генерує невидимий конструктор за замовчуванням без параметрів.

Як тільки оголошено бодай один власний конструктор, дефолтний зникає. Це часта пастка: додавши конструктор з параметрами, можна випадково «зламати» код, що покладався на new Foo().

Кількість конструкторів у класі не обмежена — за умови, що вони мають різні сигнатури (перевантаження конструкторів). Усередині одного конструктора можна викликати інший через this(arguments) — обов’язково першим рядком. Це прийом для уникнення дублювання коду ініціалізації. Конструктор не може мати модифікатор static — його семантика прив’язана до створення екземпляра, він має доступ до this. Конструктор не може мати тип повернення. Метод з ім’ям класу, але з оголошеним типом повернення, є валідним методом, а не конструктором (плутанина, якої варто уникати).

Приклад задачі

Умова. Створити клас Cadet (курсант ЛДУБЖД) з трьома перевантаженими конструкторами, що делегують роботу один одному через this().

Клас Cadet у файлі Cadet.java; створення об’єктів — у Main.java.

Лістинг 11.1. Реалізація з покроковим розбором

Файл `Cadet.java` три перевантажені конструктори

```
class Cadet {
    String name;
    int course;

    Cadet() {
        this("невідомий", 1);
    }
    Cadet(String name) {
        this(name, 1);
    }
```

Два конструктори делегують ініціалізацію третьому через виклик this(...). Конструктор без параметрів задає значення за замовчуванням, конструктор з одним параметром — лише прізвище, курс 1. Виклик this(...) має бути першим рядком.

```
    Cadet(String name, int course) {
        this.name = name;
        this.course = course;
    }
}
```

Третій конструктор — «головний»: він безпосередньо ініціалізує обидва поля. Усі інші конструктори зрештою викликають саме його. Це усуває дублювання логіки присвоєння.

Файл `Main.java` створення трьох об’єктів

```
public class Main {
  public static void main(String[] args) {
    new Cadet();
    new Cadet("Коваль");
    new Cadet("Мельник", 3);
  }
}
```

Три виклики обирають різні конструктори за кількістю аргументів. Перший дає («невідомий», 1), другий — («Коваль», 1), третій — («Мельник», 3). Компілятор обирає конструктор за сигнатурою.

Зверніть увагу. *Якщо видалити конструктор без параметрів, рядок new Cadet() перестане компілюватися: компілятор не створює конструктор за замовчуванням, коли в класі вже є інші.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити клас із трьома перевантаженими конструкторами у окремому файлі, де простіші конструктори делегують ініціалізацію через this(...). У Main.java створити три об’єкти різними конструкторами.

№	Назва класу (файл)	Поля / конструктори
1	Rescuer.java	name, rank, experience: (), (name), (name, rank)
2	Hydrant.java	x, y: (), (x), (x, y)
3	Room.java	width, length: (), (side), (width, length)
4	Extinguisher.java	type, mass, charge: (), (type), (type, mass, charge)
5	FireStation.java	number, crewSize: (), (number), (number, crewSize)

Питання 12. Що таке анонімний клас

Теоретична частина

Анонімний клас (anonymous inner class) — клас без власного імені, оголошений і інстанційований одним виразом. Синтаксис: new SuperType() { /* тіло класу */ }. SuperType може бути або інтерфейсом, або (не-final) класом. Анонімний клас неявно або наслідує клас, або реалізує інтерфейс. Анонімні класи компілюються в окремі .class-файли з іменами на кшталт Outer$1.class. Типове застосування — одноразова реалізація інтерфейсу або підкласу: слухач подій, Runnable, Comparator. Усередині можна оголошувати поля та методи, але НЕ можна оголошувати конструктори (немає імені) [9; 12; 20].

Анонімний клас має доступ до полів обрамлюючого класу та до фінальних (effectively final) локальних змінних обрамлюючого методу. З Java 8 діє правило effectively final: змінна вважається такою, якщо їй присвоюють лише один раз.

З Java 8 більшість випадків анонімних класів елегантніше пишеться через лямбда-вирази, але лише для функціональних інтерфейсів (з єдиним абстрактним методом). Для класів з кількома методами анонімний клас залишається актуальним. Нам самим під час роботи в індустрії доводилося підтримувати старий код, де анонімні класи трапляються на кожному кроці, тому читати їх треба вміти.

Приклад задачі

Умова. Реалізувати Comparator для сортування списку населених пунктів району виїзду за довжиною назви через анонімний клас.

Демонстрація в одному файлі Main.java з використанням Collections.sort.

Лістинг 12.1. Реалізація з покроковим розбором

Файл `Main.java` анонімна реалізація Comparator

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    List<String> list =
        new ArrayList<>(List.of("Брюховичі",
"Винники", "Львів"));
```

Створюємо змінюваний список з трьох назв населених пунктів району виїзду пожежно-рятувальної частини. Мета — відсортувати його за довжиною назв за зростанням за допомогою анонімного класу Comparator.

```
    Collections.sort(list,
        new Comparator<String>() {
          public int compare(String a,
String b) {
            return a.length() - b.length();
          }
        });
    System.out.println(list);
  }
}
```

Вираз new Comparator<String>(){...} одночасно оголошує анонімний клас та створює його екземпляр. Метод compare перевизначений inline. Вивід: «[Львів, Винники, Брюховичі]». Цей же код коротше: Comparator.comparingInt(String::length).

З досвіду викладачів. *Той самий Comparator у сучасному стилі записується одним рядком: list.sort(Comparator.comparingInt(String::length)). На співбесіді доречно показати обидва варіанти.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати вказаний інтерфейс через анонімний клас у файлі Main.java та продемонструвати його роботу.

№	Інтерфейс	Що реалізувати в анонімному класі
1	Runnable	вивід "Караул виїхав", запуск у новому потоці
2	Comparator<Integer>	сортування часу прибуття підрозділів за спаданням
3	Comparator<String>	сортування прізвищ курсантів без урахування регістру
4	Iterator<Integer>	видача номерів поверхів від 1 до 5
5	Runnable	вивід поточного часу заступання на чергування

Питання 13. Вкладені статичні, нестатичні внутрішні та локальні класи

Теоретична частина

Java підтримує чотири типи вкладених класів: статичні вкладені (static nested), нестатичні внутрішні (inner), локальні (local) та анонімні. Перші три розглянемо детально [2; 14].

Статичний вкладений клас (static nested class) оголошується з модифікатором static. Він не пов’язаний з екземпляром зовнішнього класу: створити його можна як Outer.Nested n = new Outer.Nested(); — без об’єкта Outer. Має доступ до статичних членів зовнішнього класу, але не до нестатичних. Це фактично окремий клас, логічно згрупований з іншим.

Нестатичний внутрішній клас (inner class) оголошується без static. Він жорстко прив’язаний до екземпляра обрамлюючого класу: створення можливе тільки через outerInstance.new Inner(). Внутрішній клас має неявне посилання на enclosing-об’єкт (Outer.this) і бачить усі його поля, включно з приватними. Класичне застосування — Iterator усередині реалізації колекції.

Локальний клас (local class) оголошується безпосередньо всередині методу. Невидимий за межами цього блоку. Має доступ до полів обрамлюючого класу та до фінальних (effectively final) локальних змінних. Сьогодні майже завжди замінюється лямбдою або анонімним класом. Кожен вкладений клас породжує окремий .class-файл з назвою на кшталт Outer$Nested.class. Для внутрішніх класів JVM додає синтетичне поле this$0 з посиланням на обрамлюючий екземпляр.

Приклад задачі

Умова. Створити клас FireStation (пожежно-рятувальна частина) з трьома типами вкладених класів, що демонструють різницю в семантиці.

Усе в одному файлі FireStation.java та демонстрація створення — у Main.java.

Лістинг 13.1. Реалізація з покроковим розбором

Файл `FireStation.java` три типи вкладених класів

```
class FireStation {
    int x = 1;        // номер частини
    static int y = 2; // кількість караулів

    static class Settings {
        int v = y;
    }
    class Crew {
        int v = x;
    }
```

Settings має доступ лише до статичного y — він не прив'язаний до екземпляра FireStation. Crew (караул) має доступ до нестатичного x через неявне посилання FireStation.this. Тому Crew потребує об'єкта FireStation для створення.

```
    void alarm() {
        class Call {
            int v = x;
        }
        System.out.println(new Call().v);
    }
}
```

Call оголошений усередині методу alarm() — видимий лише в його межах. Має доступ до поля x обрамлюючого класу. Поза методом такий клас використати неможливо. Вивід: «1».

Файл `Main.java` створення вкладених класів

```
public class Main {
  public static void main(String[] args) {
    new FireStation.Settings();
    new FireStation().new Crew();
  }
}
```

Settings створюється без екземпляра FireStation — через FireStation.Settings. Crew потребує спершу створити FireStation, потім викликати .new Crew() на ньому — це підкреслює прив'язку внутрішнього класу до екземпляра.

Зверніть увагу. *Звичайна помилка початківців — спроба написати new FireStation.Crew(). Внутрішньому класу потрібен «батьківський» об'єкт, тому без екземпляра FireStation нічого не вийде.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити зовнішній клас (Department, University, Squad тощо) із вказаним типом вкладеного класу. У Main.java продемонструвати створення вкладеного класу та доступ до полів зовнішнього класу.

№	Тип вкладеного класу	Що реалізувати
1	static nested	клас Coordinates (lat, lon) усередині класу Incident
2	inner	клас Member, що звертається до приватного поля Squad
3	local	локальний клас усередині методу evacuate() класу Building
4	inner	клас Route (маршрут виїзду) з inner-класом Node
5	static nested	два static nested класи в University (Faculty, Department), перевірити їх незалежність

Питання 14. Що таке enum-клас та як його оголосити

Теоретична частина

Перерахування (enumeration, enum) — спеціальний тип у Java, що дозволяє оголосити фіксований набір іменованих констант. Синтаксис: enum Color { RED, GREEN, BLUE }. Кожна константа всередині — це окремий публічний статичний final об’єкт типу Color, який створюється JVM в одному екземплярі. Фактично кожна enum-константа є типобезпечним singleton'ом [1].

Під капотом компілятор перетворює enum у спеціальний клас, що неявно наслідує java.lang.Enum<E>. Тому enum не може мати інших суперкласів, але може реалізовувати інтерфейси. Конструктор enum завжди private за замовчуванням.

Усередині enum-класу можна оголошувати поля, конструктори, методи. Кожна константа може мати власну реалізацію методу (constant-specific bodies). Це елегантна заміна switch. Базові методи: values() — повертає масив усіх констант у порядку оголошення; valueOf(String) — повертає константу за іменем; name() — повертає рядкове ім’я; ordinal() — порядковий номер (від 0).

Enum-типи ефективно інтегруються з мовою: безпечні для switch, мають оптимізовані колекції EnumSet та EnumMap. Це бажаний вибір для типобезпечних констант замість набору public static final int.

Порада з практики: якщо в коді з’являються «магічні» числа 1, 2, 3 для позначення станів — це майже завжди кандидат на enum.

Приклад задачі

Умова. Створити enum FireClass (класи пожеж) з полями «мінімальна витрата води» та «площа гасіння», конструктором і методом обчислення інтенсивності подачі вогнегасної речовини; вивести її для всіх класів.

Enum FireClass у файлі FireClass.java; перебір усіх констант — у Main.java.

Лістинг 14.1. Реалізація з покроковим розбором

Файл `FireClass.java` enum з полями, конструктором, методом

```
enum FireClass {
    A(12.0, 50.0),
    B(20.0, 40.0);

    private final double flow;
    private final double area;

    FireClass(double f, double s) {
        flow = f; area = s;
    }
```

A(...) та B(...) — це виклики private-конструктора з аргументами. Кожна константа — окремий екземпляр enum зі своїми значеннями полів flow (л/с) і area (м²). Конструктор enum завжди private.

```
    double intensity() {
        return flow
             / area;
    }
}
```

Метод intensity() обчислює інтенсивність подачі (л/(с·м²)) як відношення витрати до площі. Кожна константа використовує власні значення flow і area. Це звичайний метод екземпляра enum.

Файл `Main.java` перебір через values()

```
public class Main {
  public static void main(String[] args) {
    for (FireClass p : FireClass.values()) {
        System.out.println(
            p.name() + ": " + p.intensity());
    }
  }
}
```

Метод values() автоматично згенерований компілятором — повертає масив усіх констант. Метод name() повертає рядкове ім'я («A», «B»). Цикл for-each перебирає всі класи пожеж і виводить: «A: 0.24», «B: 0.5».

Зверніть увагу. *Константи enum можна порівнювати через ==, і це навіть краще за equals: кожна константа існує в єдиному екземплярі.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити enum із вказаними константами та полями у окремому файлі. У Main.java перебрати всі константи через values() та продемонструвати метод.

№	Enum (файл)	Константи / поле / метод
1	Rank.java	PRIVATE..COLONEL (звання служби цивільного захисту); метод isOfficer()
2	CallStatus.java	NEW, IN_PROGRESS, LOCALIZED, DONE; поле description
3	Direction.java	N, S, E, W; метод opposite() (напрямок вітру)
4	AlarmLevel.java	RANK_1..RANK_5; поле units; метод totalUnits()
5	Operation.java	PLUS, MINUS, MUL; абстрактний метод apply(int,int)

Питання 15. Анотації. Базовий клас Object та його методи. Перевизначення hashCode/equals

Теоретична частина

Анотація (annotation) — метадані, що додаються до елементів коду: класів, методів, полів, параметрів. Сама по собі анотація не змінює виконання коду, але обробляється або компілятором (@Override, @SuppressWarnings, @Deprecated), або під час виконання через Reflection (@Test, @Autowired, @Entity). Синтаксис: @AnnotationName(параметр = значення). Власна анотація оголошується через @interface [1; 16].

Базовий клас усіх класів у Java — java.lang.Object. Якщо клас не вказує extends, він неявно наслідує Object. Ключові методи: boolean equals(Object), int hashCode(), String toString(), Class<?> getClass(), wait()/notify()/notifyAll(), clone().

Контракт equals/hashCode критичний.

Якщо a.equals(b) повертає true, то a.hashCode() мусить дорівнювати b.hashCode(). Зворотне не обов'язкове. Метод equals має задовольняти: рефлексивність, симетричність, транзитивність, консистентність, x.equals(null) == false.

Метод hashCode не зобов'язаний давати унікальні значення, але має бути стабільним і узгодженим з equals. Реалізовується через комбінацію хешів полів. У Java 7+ є помічники: Objects.equals(a, b) та Objects.hash(field1, field2, ...). Без коректного перевизначення equals/hashCode об'єкти не працюють як ключі в HashMap/HashSet: вони розглядаються як різні навіть з однаковими полями. Це найчастіша проблема новачків.

Приклад задачі

Умова. Реалізувати клас Hydrant (пожежний гідрант на карті району виїзду) з коректним перевизначенням equals та hashCode за координатами x та y.

Клас Hydrant у файлі Hydrant.java; перевірка рівності — у Main.java.

Лістинг 15.1. Реалізація з покроковим розбором

Файл `Hydrant.java` перевизначення equals та hashCode

```
import java.util.Objects;

class Hydrant {
    final int x, y;
    Hydrant(int x, int y) {
        this.x = x; this.y = y;
    }
```

Клас Hydrant з двома незмінними (final) полями x та y. Для коректної роботи у колекціях на основі хешів (наприклад, реєстр гідрантів) необхідно перевизначити equals та hashCode узгоджено за цими полями.

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Hydrant p))
            return false;
        return x == p.x && y == p.y;
    }
    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

Спочатку перевірка this == o (оптимізація). Далі instanceof з pattern matching (Java 16+) одночасно перевіряє тип і приводить. Objects.hash комбінує хеші полів. Контракт виконано: рівні гідранти мають рівні хеші.

Файл `Main.java` перевірка рівності

```
public class Main {
  public static void main(String[] args) {
    Hydrant a = new Hydrant(1, 2);
    Hydrant b = new Hydrant(1, 2);
    System.out.println(a.equals(b));
    System.out.println(a.hashCode() ==
b.hashCode());
  }
}
```

Два різні об’єкти з однаковими координатами. Вивід: «true», «true». equals повертає true за вмістом, а хеші однакові — це дозволяє коректно використовувати Hydrant як ключ у HashMap.

Порада. *Перевірте себе: приберіть метод hashCode і покладіть два однакові гідранти в HashSet. Розмір множини стане 2 — саме так виглядає порушений контракт equals/hashCode.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати клас із перевизначеними equals та hashCode за вказаними полями. У Main.java створити два рівних об’єкти та перевірити рівність і збіг хешів.

№	Клас (файл)	Поля для equals/hashCode
1	Cadet.java	name, birthYear
2	Certificate.java	number
3	FireTruck.java	model, plateNumber
4	IncidentLocation.java	latitude, longitude
5	Equipment.java	inventoryId

Питання 16. Різниця між String, StringBuilder та StringBuffer

Теоретична частина

Клас String у Java є незмінним (immutable). Будь-яка операція (concat, replace, substring, toLowerCase) повертає новий об'єкт. Незмінність дає безпечну роботу між потоками, кешування хеш-коду, ефективний String Pool. Але має ціну: часті модифікації породжують велику кількість тимчасових об'єктів і навантажують збирач сміття (GC) [13; 21].

Клас StringBuilder є змінною (mutable) послідовністю символів з внутрішнім буфером char[]. Операції append, insert, delete працюють «по місцю» — з амортизованою складністю O(1) для append. Не потокобезпечний, але швидкий. Інструмент для побудови рядків у циклах.

Клас StringBuffer — функціонально ідентичний StringBuilder, але всі його методи synchronized. Це робить його потокобезпечним, але з накладними витратами. У сучасному Java рідко потрібен. Правило вибору: один рядок або кілька конкатенацій — String. Побудова рядка у циклі — StringBuilder. Тільки якщо точно потрібна потокобезпечність — StringBuffer.

Пастка: метод toString() у StringBuilder створює новий об'єкт String. Тому виклик sb.toString() двічі дасть два різних об'єкти String з однаковим вмістом.

На співбесіді досить чітко назвати три відмінності: незмінність, потокобезпечність і швидкодія.

Приклад задачі

Умова. Продемонструвати різницю продуктивності при формуванні рядка-коду маршруту з п'яти цифр (0–4) через String += проти StringBuilder.append.

Обидва підходи в одному файлі Main.java для порівняння.

Лістинг 16.1. Реалізація з покроковим розбором

Файл `Main.java` неефективний підхід через String +=

```
public class Main {
  public static void main(String[] args) {
    String s = "";
    for (int i = 0; i < 5; i++) {
        s += i;
    }
    System.out.println(s);
```

Кожна ітерація s += і створює новий об'єкт String (фактично новий StringBuilder, append, toString). Для n елементів — $O(n^2)$ часу і пам'яті. Вивід: «01234». Для невеликих циклів прийнятно, для великих — катастрофа.

Файл `Main.java` ефективний підхід через StringBuilder

```
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 5; i++) {
        sb.append(i);
    }
    System.out.println(sb.toString());
  }
}
```

StringBuilder використовує спільний буфер, що іноді розширюється — амортизована O(n). Метод append додає елемент по місцю, без створення тимчасових об'єктів. Вивід: «01234» — той самий результат, але ефективніше.

Зверніть увагу. *Для п'яти ітерацій різниці у швидкості ви не помітите. Збільшіть межу циклу до 100 000 і заміряйте час через System.nanoTime() — результат зазвичай вражає курсантів.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java реалізувати побудову рядка двома способами (String += та StringBuilder) для вказаного завдання та порівняти результат.

№	Завдання
1	Сконкатенувати номери викликів від 1 до 100 у один рядок через пробіл.
2	Побудувати рядок-роздільник журналу чергування з 50 символів «*».
3	Реверсувати рядок «ЛДУБЖД» через StringBuilder.reverse().
4	Зібрати CSV-рядок зі списку прізвищ курсантів взводу через кому.
5	Побудувати рядок із чисел Фібоначчі (перші 15) через дефіс.

Питання 17. Що таке generic-клас, як його оголосити

Теоретична частина

Generic-клас (узагальнений клас) — клас, параметризований одним або декількома типами. Синтаксис: class Box<T> { T value; }. Тут T — параметр типу, який замінюється конкретним типом при використанні: Box<String>, Box<Integer>. Generic забезпечує типобезпечність на стадії компіляції без явних кастів і дублювання коду. Конвенція іменування параметрів: T (type), E (element), K і V (key/value), R (return), N (number). Назви параметрів локальні для класу/методу [11].

Java реалізує generics через стирання типів (type erasure): у байт-коді generic-інформація стирається, T замінюється на свою межу (зазвичай Object). Через це є обмеження: не можна new T(), не можна T[] arr = new T[10], не можна instanceof T, не можна статичні поля типу T.

Generic-методи мають власний список параметрів типу перед типом повернення: <T> T pickFirst(List<T> list). Bounded type обмежує множину типів: <T extends Number>. Wildcards (?): List<? extends Number> — коваріантність, List<? super Integer> — контраваріантність (правило PECS).

Приклад задачі

Умова. Реалізувати узагальнений клас Locker (шафка спорядження рятувальника), що може зберігати предмет будь-якого типу з типобезпечним доступом.

Generic-клас Locker у файлі Locker.java; використання з різними типами — у Main.java.

Лістинг 17.1. Реалізація з покроковим розбором

Файл `Locker.java` узагальнений клас з параметром T

```
class Locker<T> {
    private T value;
    public void set(T v) {
        value = v;
    }
    public T get() {
        return value;
    }
}
```

T — параметр типу. Поле value, параметр методу set і тип повернення get() — усі типу T. При інстанціації Locker<String> компілятор підставить String замість T і перевірятиме типи на етапі компіляції.

Файл `Main.java` використання без кастів

```
public class Main {
  public static void main(String[] args) {
    Locker<String> b = new Locker<>();
    b.set("Каска");
    String s = b.get();
    System.out.println(s);
  }
}
```

Locker<String> параметризує клас типом String. Метод get() повертає одразу String — без явного касту. Вивід: «Каска». Без generics довелося б зберігати Object і кастити: (String) b.get() — без гарантії типу.

Зверніть увагу. *Спробуйте покласти в Locker<String> число: b.set(42). Помилка з'явиться ще на етапі компіляції — у цьому й полягає головна цінність generics.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити узагальнений клас у окремому файлі. У Main.java продемонструвати його використання з різними типами без явних кастів.

№	Generic-клас (файл)	Що реалізувати
1	Pair.java	поля first, second типів A, B (наприклад, рятувальник і посада); геттери
2	Stack.java	push, pop, peek на основі ArrayList<T> (стек донесень)
3	Cache.java	get/put на основі Map<K, V> (кеш адрес гідрантів)
4	Triple.java	три поля A, B, C (частина, караул, автомобіль); метод toString
5	Container.java	add, get(index) на основі List<T> (перелік спорядження)

Питання 18. Що означає List<T extends Number>

Теоретична частина

Конструкція List<T extends Number> — це список елементів типу T, де T є generic-параметром, обмеженим зверху (upper bound) класом Number. Усередині класу або методу T трактується як підтип Number: можна викликати методи Number (intValue, doubleValue), присвоювати їх змінній типу Number, але не методи специфічніших підтипів [1; 11].

Ключова відмінність від wildcard-конструкції List<? extends Number>: у випадку з T параметр типу є КОНКРЕТНИМ (хоч і невідомим до інстанціації) типом, який зберігає свою ідентичність. Можна оголосити змінну T x, повернути T, прийняти T як параметр. Wildcard натомість анонімний.

Обмеження extends можна комбінувати: <T extends Number & Comparable<T>> вимагає, щоб T був Number І реалізовував Comparable. Перша межа має бути класом, решта — інтерфейсами. Ключове слово extends в контексті generics означає «є підтипом» (включаючи інтерфейси), а не «наслідує клас». Це часто заплутує початківців.

Приклад задачі

Умова. Реалізувати generic-метод sum, що обчислює сумарну витрату води підрозділами на пожежі для будь-якого списку чисел (Integer, Double, Long) через bounded type.

Generic-метод у файлі WaterUtil.java; виклики з різними типами — у Main.java.

Лістинг 18.1. Реалізація з покроковим розбором

Файл `WaterUtil.java` generic-метод з bounded type

```
import java.util.List;

class WaterUtil {
    static <T extends Number> double sum(
            List<T> list) {
        double s = 0;
        for (T n : list) {
            s += n.doubleValue();
        }
        return s;
    }
}
```

Параметр T обмежений зверху: <T extends Number>. Тому T гарантовано має метод doubleValue() — він оголошений у Number. Метод працює для будь-якого нащадка Number: Integer, Double, Long тощо.

Файл `Main.java` виклики з різними типами

```
import java.util.List;

public class Main {
  public static void main(String[] args) {
    System.out.println(WaterUtil.sum(List.of(1, 2,
3)));
    System.out.println(WaterUtil.sum(List.of(1.5,
2.5)));
  }
}
```

Перший виклик з List<Integer> (тонни води від трьох автоцистерн) повертає 6.0, другий з List<Double> — 4.0. Компілятор перевіряє типи: sum(List.of("a")) дав би помилку компіляції, бо String не є нащадком Number.

З досвіду викладачів. *Метод sum повертає double навіть для цілих чисел. Для обліку витрат води це прийнятно, але для грошових сум краще використовувати BigDecimal.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати generic-метод з bounded type <T extends Number> у окремому файлі. У Main.java викликати його зі списками різних числових типів.

№	Метод	Що обчислює
1	average	середній час прибуття підрозділів на виклик
2	max	максимальна площа пожежі (через doubleValue)
3	countPositive	кількість додатних приростів температури
4	min	мінімальний тиск у рукавній лінії
5	sumPositives	сумарна довжина лише ненульових рукавних ліній

Питання 19. Ієрархія винятків. Checked vs unchecked exceptions

Теоретична частина

Базовим класом усіх винятків у Java є java.lang.Throwable. Лише об’єкти цього типу можуть бути викинуті через throw. Від Throwable походять дві гілки: Error та Exception. Клас Error представляє серйозні проблеми JVM, які зазвичай не передбачено обробляти: OutOfMemoryError, StackOverflowError, VirtualMachineError. Спроба «полагодити» їх у коді небезпечна [1; 2].

Клас Exception представляє винятки, що можуть бути оброблені. Поділяються на checked та unchecked. Checked exceptions — всі підкласи Exception, КРІМ

RuntimeException: IOException, SQLException, ClassNotFoundException. Компілятор примусово вимагає обробити їх у try/catch або декларувати в throws.

Unchecked exceptions — підкласи RuntimeException та Error: NullPointerException, IllegalArgumentException, IndexOutOfBoundsException, ArithmeticException. Компілятор не примушує їх обробляти. Зазвичай це помилки програміста — баги, які мають бути виправлені, а не перехоплені.

Проєктна порада: успадковуйте Exception для винятків, які клієнт мусить явно обробляти; RuntimeException — для помилок використання API. Курсанти на заняттях часто запитують, чи можна «просто ловити Exception скрізь». Технічно можна, але так ви сховаєте від себе справжні помилки.

Приклад задачі

Умова. Продемонструвати обидві категорії: ловлю checked-винятку IOException під час читання журналу викликів та unchecked NullPointerException.

Обидва приклади в одному файлі Main.java з обробкою через try/catch.

Лістинг 19.1. Реалізація з покроковим розбором

Файл `Main.java` checked-виняток (компілятор вимагає обробку)

```
import java.nio.file.*;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    try {
        Files.readAllLines(Path.of("calls.txt"));
    } catch (IOException e) {
        e.printStackTrace();
    }
```

IOException — checked-виняток: вказує на зовнішню проблему (файл журналу викликів може не існувати). Без try/catch або throws код НЕ скомпілюється. Компілятор примушує обробити такий виняток.

Файл `Main.java` unchecked-виняток (компілятор не вимагає)

```
    String s = null; // адреса не вказана
    try {
        s.length();
    } catch (NullPointerException e) {
        System.out.println("NPE caught");
    }
  }
}
```

NullPointerException — unchecked: вказує на баг у коді (доступ до null). Компілятор не вимагає try/catch — ми додали його добровільно. Вивід: «NPE caught». Зазвичай такі винятки виправляють, а не ловлять.

Зверніть увагу. *Порожній блок catch — одна з найгірших звичок. Якщо виняток справді можна проігнорувати, напишіть хоча б коментар, чому саме.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити власний виняток вказаного типу у окремому файлі та продемонструвати його кидання й обробку у Main.java.

№	Власний виняток (файл)	Тип / коли кидати
1	IncidentException.java	checked; при порушенні правил реєстрації виклику
2	ValidationException.java	unchecked; при некоректних даних курсанта
3	HydrantNotFoundException.java	checked; коли гідрант за координатами не знайдено
4	WaterLevelException.java	unchecked; при від’ємному залишку води в цистерні
5	ReportParseException.java	checked; при помилці розбору рядка донесення

Питання 20. try/catch/finally. Стек викликів. Збереження стеку при re-throw. Garbage Collection

Теоретична частина

Блок try обмежує ділянку коду, де можливе виникнення винятку. Блоки catch перехоплюють винятки конкретних типів послідовно зверху вниз — спрацьовує перший підходящий. Блок finally виконується завжди — і при нормальному, і при винятковому завершенні, і навіть при return із try. Для AutoCloseable-ресурсів є зручніший синтаксис try-with-resources [13; 15].

Стек викликів (call stack) у винятку — послідовність методів від точки кидання до точки перехоплення. Метод getStackTrace() повертає масив StackTraceElement; printStackTrace() виводить його. При повторному киданні (re-throw) важливо зберегти оригінальний стек через конструктор з cause: throw new MyException("...", original); — буде секція «Caused by:».

Garbage Collector у Java використовує алгоритм трасування досяжності, а не reference counting. Об’єкт досяжний, якщо існує ланцюг посилань із GC roots: стек активних потоків, статичні поля, JNI-посилання. Через це цикл посилань A→B→C→A без зовнішніх посилань коректно збирається GC: трасування від коренів його не досягає. Це перевага трасуючого алгоритму над reference counting. Сучасні GC (G1, ZGC, Shenandoah) використовують generational hypothesis: молоде покоління збирається часто і швидко, старе — рідко.

Приклад задачі

Умова. Реалізувати try-catch-finally з повторним киданням винятку зі збереженням оригінального стека через cause (на прикладі збою зчитування датчика пожежної сигналізації).

Демонстрація вкладеного try з re-throw у файлі Main.java.

Лістинг 20.1. Реалізація з покроковим розбором

Файл `Main.java` вкладений try з re-throw та cause

```
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    try {
        try {
            throw new IOException("датчик диму");
        } catch (IOException e) {
            throw new
RuntimeException("сигналізація", e);
        } finally {
            System.out.println("finally
executed");
        }
    } catch (RuntimeException ex) {
        ex.printStackTrace();
    }
  }
}
```

Внутрішній catch ловить IOException і перекидає її як RuntimeException, передаючи оригінал як cause (другий аргумент). Це зберігає ланцюг винятків. Блок finally виконається перед розповсюдженням винятку вгору.

Зовнішній catch ловить RuntimeException. Метод printStackTrace() покаже обидва рівні: основний виняток та секцію «Caused by: java.io.IOException: датчик диму». Порядок виводу: спочатку «finally executed», потім трасування.

Зверніть увагу. *Приберіть другий аргумент e у конструкторі RuntimeException і порівняйте трасування. Секція «Caused by» зникне, а разом з нею — найцінніша інформація для пошуку причини збою.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати обробку винятків у файлі Main.java згідно теми варіанту та продемонструвати результат.

№	Що продемонструвати
1	try-catch-finally з return у try (закриття журналу чергування); показати, що finally все одно виконується.
2	Перехопити IOException під час читання файлу донесення та перекинути як RuntimeException із cause.
3	Каскад catch для FileNotFoundException та IOException під час відкриття плану евакуації; пояснити порядок.
4	try-with-resources із власним класом RadioChannel, що реалізує AutoCloseable.
5	Multi-catch (IOException | RuntimeException e) при обробці даних датчиків; продемонструвати переваги.

Питання 21. FizzBuzz: реалізація за 5 хвилин на папері

Теоретична частина

FizzBuzz — класична задача-розминка для співбесід. Формулювання: вивести числа від 1 до 100; для кратних 3 — «Fizz»; для кратних 5 — «Buzz»; для кратних обом (тобто 15) — «FizzBuzz». Незважаючи на тривіальність, задача дозволяє оцінити базову зрілість програміста. У прикладі нижче замість «Fizz/Buzz» використано слова «Fire/Rescue» — логіка від цього не змінюється [22; 23; 65].

Найважливіший момент — порядок перевірок. Перевірка на кратність 15 (або, еквівалентно, на кратність 3 І 5) має йти ПЕРШОЮ. Якщо спочатку перевірити % 3 — для числа 15 алгоритм виведе «Fizz» і завершить ітерацію, ніколи не дійшовши до «FizzBuzz». Це найчастіша пастка.

Існує два класичні підходи: каскадний if-else-if (швидкий, читабельний, очікуваний на більшості співбесід) та конкатенація рядків з тернарними операторами (елегантніший, легше узагальнюється).

Задача також дозволяє обговорити: узагальнення (параметризований метод), функціональний стиль (IntStream), тестабельність (повертати List<String> замість друку).

Із нашого досвіду: більшість кандидатів, які «провалюють» FizzBuzz, роблять це не через незнання, а через хвилювання. Розв’яжіть задачу кілька разів на папері до співбесіди.

Приклад задачі

Умова. Реалізувати класичний FizzBuzz для нумерації 100 викликів за добу: кратні 3 — «Fire», кратні 5 — «Rescue», кратні 15 — «FireRescue».

Реалізація у файлі Main.java з каскадним if-else-if.

Лістинг 21.1. Реалізація з покроковим розбором

Файл `Main.java` класичний FizzBuzz (FireRescue)

```
public class Main {
  public static void main(String[] args) {
    for (int i = 1; i <= 100; i++) {
        if (i % 15 == 0)
            System.out.println("FireRescue");
        else if (i % 3 == 0)
            System.out.println("Fire");
        else if (i % 5 == 0)
            System.out.println("Rescue");
        else
            System.out.println(i);
    }
  }
}
```

Перевірка % 15 стоїть ПЕРШОЮ — це критично. Якби спочатку йшла % 3, то для числа 15 виконалася б гілка «Fire», і «FireRescue» ніколи б не вивелося. Еквівалентний запис: i % 3 == 0 && i % 5 == 0.

Решта гілок: кратні 3 (але не 15) — «Fire», кратні 5 (але не 15) — «Rescue», інакше саме число. Завдяки else-if кожне число потрапляє рівно в одну гілку. Складність циклу — O(n).

Порада. *На співбесіді не поспішайте. Спочатку вголос проговоріть порядок перевірок, потім пишіть код — інтерв'юер оцінює хід думок не менше, ніж результат.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати варіацію FizzBuzz у файлі Main.java згідно теми варіанту.

№	Варіація завдання
1	FizzBuzz для діапазону 1–30 (номери курсантів у строю) без використання оператора % (через лічильники).
2	FizzBuzz через конкатенацію рядків (без if-else-if).
3	Узагальнений метод fizzBuzz(int n, int a, int b, String wa, String wb).
4	FizzBuzz через IntStream, що повертає List<String>.
5	FizzBuzzBang: ділення на 3, 5, 7 з трьома словами «Fire», «Rescue», «Evac».

Розділ 2. Колекції та базове введення-виведення

Питання 22. Супер-інтерфейс для List, Set, Map. Чи є Collection базовим для всіх трьох?

Теоретична частина

Архітектура Java Collections Framework має дві паралельні гілки інтерфейсів. Перша — інтерфейс Collection<E>, базовий для одновимірних колекцій. Від нього походять List<E> (упорядкована з дублікатами), Set<E> (без дублікатів), Queue<E> та Deque<E> (черги) [11; 16].

Друга гілка — окремий інтерфейс Map<K, V>, що НЕ наслідує Collection. Це типова помилка на співбесідах — заявити, що Collection базовий і для Map.

Причина окремого існування Map: її елемент — пара ключ-значення (Map.Entry<K, V>), що принципово відрізняється від поняття одиничного елемента у Collection. Map надає три методи-переходи: keySet() повертає Set ключів, values() — Collection значень, entrySet() — Set<Map.Entry>.

Над Collection стоїть інтерфейс Iterable<E>, що має єдиний абстрактний метод iterator() та з Java 8 default-метод forEach. Будь-який Collection є Iterable, але не кожен Iterable є Collection. Ієрархія: Iterable → Collection → {List, Set, Queue, Deque}. Map стоїть осторонь. Map.entrySet() та Map.keySet() повертають Set'и, які вже є частиною колекційної ієрархії і отже Iterable.

Java 21 додала інтерфейс SequencedCollection з методами getFirst, getLast, addFirst, addLast, reversed.

Приклад задачі

Умова. Продемонструвати на даних чергової частини, що List, Set, Queue є Collection та Iterable, а Map — лише вершина окремої гілки.

Демонстрація ієрархії типів у файлі Main.java через присвоєння.

Лістинг 22.1. Реалізація з покроковим розбором

Файл `Main.java` List/Set є Collection, Map — ні

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    Iterable<Integer> it = List.of(101, 102, 103);
    Collection<Integer> c = new ArrayList<>();
    List<Integer>       l = new ArrayList<>();
    Set<Integer>        s = new HashSet<>();
    Map<String, Integer> m = new HashMap<>();
```

List успадковує Collection, яка успадковує Iterable — тому всі ці присвоєння валідні. Map оголошена окремо. Спроба написати Collection<?> c2 = m; призвела б до помилки компіляції — Map НЕ є Collection.

```
    Collection<Integer> vals = m.values();
    Set<String>          ks   = m.keySet();
    System.out.println(vals + " " + ks);
  }
}
```

Хоча Map не є Collection, вона дає колекційні представлення: values() повертає Collection значень, keySet() — Set ключів. Через ці методи Map можна перебирати у for-each.

Зверніть увагу. *Методи keySet() та values() повертають «живі» представлення Map. Якщо видалити ключ із keySet(), пара зникне і з самої мапи.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java продемонструвати взаємозв’язки між інтерфейсами колекцій згідно теми варіанту.

№	Що продемонструвати
1	Створити власний клас Platoon, що реалізує Iterable<String> (прізвища курсантів), але не є Collection.
2	Присвоїти List<Integer> номерів частин у Collection<Integer>; пояснити валідність.
3	Вивести розміри keySet(), values(), entrySet() для Map «частина → кількість автомобілів» з трьома парами.
4	Перетворити Set<String> районів виїзду на List<String> і навпаки.
5	Перебрати Map «курсант → бал» через entrySet у циклі for-each, вивести пари ключ=значення.

Питання 23. Методи інтерфейсів Iterable та Iterator

Теоретична частина

Інтерфейс Iterable<T> — джерело ітерації. Має один абстрактний метод: Iterator<T> iterator(). З Java 8 додано default void forEach(Consumer) та default Spliterator<T> spliterator() (для паралельних потоків). Реалізація Iterable — необхідна і достатня умова для використання у for-each [8].

Інтерфейс Iterator<T> — об'єкт-курсор, що зберігає стан позиції. Має три основні методи: boolean hasNext() — перевіряє наявність наступного без зсуву; T next() — повертає наступний і просуває курсор; default void remove() — видаляє останній повернутий (опційний). З Java 8 — forEachRemaining(Consumer).

Семантика next() після кінця: NoSuchElementException. Правильний шаблон — завжди викликати hasNext() перед next(). Кожен виклик iterator() створює новий незалежний курсор. Концепція fail-fast: стандартні колекції використовують лічильник modCount. Якщо колекція модифікується НЕ через iterator.remove() під час ітерації, ітератор кидає ConcurrentModificationException. Це спосіб виявити баг у одному потоці.

Fail-safe ітератори (CopyOnWriteArrayList, ConcurrentHashMap) не кидають CME — вони працюють зі знімком колекції. Саме тому їх використовують у багатопотокових програмах.

Приклад задачі

Умова. Продемонструвати пряму роботу з Iterator при перегляді номерів караулів, оминаючи синтаксис for-each.

Робота з ітератором у файлі Main.java.

Лістинг 23.1. Реалізація з покроковим розбором

Файл `Main.java` пряма ітерація через Iterator

`import java.util.*;`
 	
`public class Main {`
`  public static void main(String[] args) {`
`  List<Integer> list = List.of(1, 2, 3); // караули`
`  Iterator<Integer> it = list.iterator();`
	Метод iterator() повертає об'єкт-курсор, що стоїть перед першим елементом. Цей же курсор неявно використовується циклом for-each — тут ми працюємо з ним напряму для розуміння механізму.

`  while (it.hasNext()) {`
`  Integer x = it.next();`
`  System.out.print(x);`
`  }`
`  }`
`}`
	hasNext() повертає true, поки є непройдені елементи. next() повертає поточний і просуває курсор. Вивід: «123». Саме у такий код компілятор перетворює цикл for-each.

Зверніть увагу. *Цикл while з ітератором — саме те, у що компілятор розгортає for-each. Корисно хоча б раз написати його вручну, щоб for-each перестав бути «чорною скринькою».*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу з Iterator у файлі Main.java згідно теми варіанту.

№	Що реалізувати
1	Власний Iterator<Integer>, що видає номери поверхів від 1 до n.
2	Iterator над позивним «ЛЬВІВ-101», що видає окремі символи як Character.
3	Викликати next() після hasNext()==false; обробити NoSuchElementException.
4	Продемонструвати ConcurrentModificationException при видаленні виконаного виклику у for-each.
5	Видалити закритий виклик через Iterator.remove(); переконатися, що CME не виникає.

Питання 24. Як зробити колекцію незмінною

Теоретична частина

Незмінна колекція (unmodifiable/immutable) — колекція, методи мутації якої (add, remove, clear, set, put) кидають UnsupportedOperationException. Це інструмент захисту: ви публікуєте колекцію назовні, але не дозволяєте її змінювати [1; 9].

Перший механізм: Collections.unmodifiableList(orig) / unmodifiableSet / unmodifiableMap. Вони повертають обгортку-view над оригіналом. Це не копія: якщо змінити оригінал, обгортка покаже зміни. Дає лише «протекцію від клієнта».

Другий механізм (Java 9+): List.of(...), Set.of(...), Map.of(...). Повертають справді незмінні колекції: вміст копіюється, спроба мутації — виняток. Особливості: null заборонений, дублікати в Set.of заборонені, повторні ключі в Map.of заборонені. Третій механізм: Stream.toList() (Java 16+) та Collectors.toUnmodifiableList/Set/Map().

«Незмінність» поширюється лише на саму колекцію. Якщо елементи — змінні об’єкти, їхній стан можна модифікувати. Для повної (deep) immutability елементи теж мають бути immutable.

Приклад задачі

Умова. Продемонструвати на списку номерів пожежно-рятувальних частин різницю між обгорткою-view та справді незмінною колекцією, створеною через List.of.

Порівняння двох підходів у файлі Main.java.

Лістинг 24.1. Реалізація з покроковим розбором

Файл `Main.java` обгортка-view vs незмінна копія

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    List<Integer> orig = new
ArrayList<>(List.of(1, 2, 3));
    List<Integer> wrap =
Collections.unmodifiableList(orig);
    List<Integer> immut = List.of(1, 2, 3);
```

wrap — обгортка над orig (view, не копія). immut — справді незмінна колекція з власною копією даних. Спроба wrap.add(...) чи immut.add(...) кине UnsupportedOperationException.

```
    orig.add(99);
    System.out.println(wrap);
    System.out.println(immut);
  }
}
```

Модифікуємо оригінал: orig.add(99). Вивід wrap: «[1, 2, 3, 99]» — обгортка бачить зміни оригіналу. Вивід immut: «[1, 2, 3]» — справді незмінна копія, незалежна від orig.

З досвіду викладачів. *Запам'ятайте просте правило: якщо колекцію повертає метод вашого класу, а змінювати її ззовні не можна, повертайте List.copyOf(...) або обгортку unmodifiableList.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java продемонструвати створення незмінної колекції згідно теми варіанту та спробу її модифікації.

№	Що продемонструвати
1	Створити List.of номерів частин (1, 2, 3) та спробувати add(); перехопити виняток.
2	Створити unmodifiable-обгортку над списком чергових, змінити оригінал; показати, що бачить обгортка.
3	Перетворити Set<String> навчальних корпусів на unmodifiable через Collections.unmodifiableSet.
4	Створити Map.of("АЦ",3,"АД",1) та спробувати put("АР",2).
5	Спробувати створити List.of(null); пояснити NullPointerException.

Питання 25. Ітерація по ключах та значеннях Map

Теоретична частина

Map<K, V> надає три представлення: keySet() повертає Set<K> ключів, values() повертає Collection<V> значень, entrySet() повертає Set<Map.Entry<K, V>>. Усі три — view над оригіналом: модифікація через них впливає на саму Map [16].

Для проходу по парах оптимально використовувати entrySet — це єдиний прохід, у якому ключ і значення доступні одразу. Альтернатива через keySet + get(key) для кожного ключа виконує додатковий пошук у хеш-таблиці. Для великих мап різниця суттєва.

З Java 8 є метод forEach(BiConsumer): map.forEach((k, v) -> ...). Найлаконічніший варіант. Інтерфейс Map.Entry надає getKey(), getValue(), setValue(V). Stream-варіант: map.entrySet().stream().

Порядок ітерації залежить від реалізації. HashMap не гарантує порядку. LinkedHashMap зберігає порядок вставки. TreeMap впорядковує за ключами.

Приклад задачі

Умова. Продемонструвати три способи ітерації по парам Map «тип пожежного автомобіля → кількість у частині»: entrySet, keySet+get, forEach.

Три способи в одному файлі Main.java для порівняння.

Лістинг 25.1. Реалізація з покроковим розбором

Файл `Main.java` три способи ітерації

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    Map<String, Integer> m =
        Map.of("АЦ", 3, "АД", 1, "АР", 2);

    for (var e : m.entrySet()) {
        System.out.println(
            e.getKey() + " = " + e.getValue());
    }
```

Спосіб 1 — entrySet: найефективніший, один прохід. Кожен елемент — Map.Entry з методами getKey() та getValue(). Це рекомендований спосіб для перебору пар.

```
    for (String k : m.keySet()) {
        System.out.println(k + " = " + m.get(k));
    }

    m.forEach((k, v) ->
        System.out.println(k + " = " + v));
  }
}
```

Спосіб 2 — keySet + get: менш ефективний, бо для кожного ключа додатковий хеш-лукап. Спосіб 3 — forEach з лямбдою: найлаконічніший, по швидкості еквівалентний entrySet.

Зверніть увагу. *Порядок виводу пар у прикладі може відрізнятися від порядку в Map.of — така мапа порядку не гарантує. Якщо порядок важливий, використовуйте LinkedHashMap або TreeMap.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java виконати операцію над Map згідно теми варіанту, використовуючи перебір по парах.

№	Що реалізувати
1	Створити Map «прізвище курсанта → бал» з 5 пар та проітерувати через entrySet.
2	Обчислити сумарну кількість особового складу всіх караулів через values + stream.
3	Створити TreeMap «номер частини → адреса» та проітерувати у відсортованому порядку ключів.
4	Знайти Entry «район → кількість викликів» з максимальним значенням.
5	Вивести лише ключі (назви кафедр) через keySet та лише значення через values.

Питання 26. Як реалізувати Map тільки на масивах

Теоретична частина

Найпростіша реалізація Map на масивах: два паралельні масиви keys[] та values[]. Метод put(K, V) виконує лінійний пошук по keys: якщо ключ знайдено — оновлює value; інакше додає в кінець обох масивів. Метод get(K) — аналогічний пошук. Складність — O(n) [60; 61].

Це не оптимальна структура, але вона коректно відображає семантику Map і часто пропонується на співбесідах. Переваги — мінімум абстракцій; недоліки — повільність,

фіксованість розміру без resize. Більш ефективна реалізація — хеш-таблиця: масив buckets, де індекс = hash(key) % buckets.length. Колізії розв'язуються через chaining (зв'язані списки) або open addressing. Складність — амортизована O(1).

Стандартний HashMap комбінує підходи: масив Node з chaining; коли довжина списку перевищує поріг (8) і масив >= 64, список перетворюється на red-black tree (O(log n)). При перевищенні load factor (0.75) масив подвоюється. Інша структура — TreeMap на red-black tree: O(log n), підтримує впорядкованість ключів.

Приклад задачі

Умова. Реалізувати найпростіший ArrayMap з двома паралельними масивами та операціями put/get (наприклад, для довідника «позивний → номер частини»).

Клас ArrayMap у файлі ArrayMap.java; використання — у Main.java.

Лістинг 26.1. Реалізація з покроковим розбором

Файл `ArrayMap.java` Map на двох паралельних масивах

```
class ArrayMap<K, V> {
    Object[] keys = new Object[16];
    Object[] vals = new Object[16];
    int size = 0;

    void put(K k, V v) {
        for (int i = 0; i < size; i++)
            if (keys[i].equals(k)) {
                vals[i] = v; return;
            }
        keys[size] = k;
        vals[size] = v;
        size++;
    }
```

Два паралельні масиви: keys і vals однакового розміру. Метод put робить лінійний пошук ключа: якщо знайдено — оновлює значення; інакше додає в кінець. Складність O(n).

```
    @SuppressWarnings("unchecked")
    V get(K k) {
        for (int i = 0; i < size; i++)
            if (keys[i].equals(k))
                return (V) vals[i];
        return null;
    }
}
```

Метод get теж лінійно шукає ключ через equals і повертає відповідне значення, або null. Реальна реалізація потребувала б resize, обробки null-ключа та видалення.

Зверніть увагу. *Реалізація навмисно спрощена: при додаванні сімнадцятого елемента масиви переповняться. Додати розширення масивів — хороша вправа (див. варіант 2).*

Варіанти для самостійної роботи

Завдання. За зразком прикладу розширити клас ArrayMap (або реалізувати власну версію) у окремому файлі згідно теми варіанту. У Main.java продемонструвати роботу.

№	Що реалізувати додатково
1	Метод remove(K key) зі зсувом залишку масивів (вилучення частини з довідника).
2	Динамічне розширення масивів удвічі при заповненні довідника позивних.
3	Методи size(), isEmpty(), containsKey(K) для реєстру гідрантів.
4	Метод keys(), що повертає масив усіх позивних.
5	Просту хеш-функцію та chaining через зв'язані списки для довідника адрес.

Питання 27. Сортування List. Перетворення Array↔List. LinkedList vs ArrayList

Теоретична частина

Сортування List: Collections.sort(list) — природний порядок (Comparable); Collections.sort(list, comparator) — за Comparator. З Java 8 — list.sort(comparator). Сучасно: list.stream().sorted().toList(). Алгоритм — TimSort, O(n log n). Масив→List: Arrays.asList(array) — fixed-size List (add/remove кинуть виняток). Для модифікованого: new ArrayList<>(Arrays.asList(array)). Пастка: Arrays.asList на int[] поверне List<int[]> з одним елементом. Для примітивів — IntStream.of(array).boxed().toList() [13; 61].

Collection→array: toArray() повертає Object[]; типобезпечний — toArray(new String[0]) або toArray(String[]::new).

ArrayList базується на масиві: O(1) get(i), O(1) add у кінець (амортизовано), O(n) add/remove у середину. Чудова cache locality.

LinkedList — двозв'язний список: O(1) на кінцях, O(n) get(i). Погана cache locality.

На практиці ArrayList майже завжди швидший навіть там, де LinkedList теоретично має перевагу. Тож на співбесіді краще чесно сказати «за замовчуванням я беру ArrayList», ніж розповідати теорію, якою ніхто не користується.

Приклад задачі

Умова. Продемонструвати сортування списку часу прибуття підрозділів (хв), перетворення масиву позивних на список та назад.

Усі операції в одному файлі Main.java.

Лістинг 27.1. Реалізація з покроковим розбором

Файл `Main.java` сортування та конверсії

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    List<Integer> list =
        new ArrayList<>(List.of(9, 6, 7));
    Collections.sort(list);
    list.sort(Comparator.reverseOrder());
    System.out.println(list);
```

Collections.sort сортує за зростанням: [6, 7, 9]. Потім list.sort(reverseOrder()) — за спаданням: [9, 7, 6]. Обидва методи модифікують список на місці. Алгоритм — TimSort, O(n log n).

```
    String[] arr = {"Львів-1", "Львів-2",
"Львів-3"};
    List<String> l =
        new ArrayList<>(Arrays.asList(arr));
    String[] back = l.toArray(new String[0]);
    System.out.println(l + " " + back.length);
  }
}
```

Arrays.asList(arr) загорнуто в new ArrayList<>() — інакше був би fixed-size список. Зворотне перетворення toArray(new String[0]) дає типобезпечний String[]. Вивід: «[Львів-1, Львів-2, Львів-3] 3».

Порада. *Пам'ятайте, що Arrays.asList(arr) без обгортки повертає список фіксованого розміру: виклик add() кине UnsupportedOperationException.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java виконати операцію зі списком згідно теми варіанту.

№	Що реалізувати
1	Відсортувати List<String> назв населених пунктів району виїзду за довжиною за спаданням.
2	Перетворити int[] балів курсантів на List<Integer> через IntStream.of().boxed().
3	Перетворити Set<Integer> номерів аудиторій на Integer[] через toArray.
4	Порівняти час 100 000 операцій add() (реєстрація викликів) у ArrayList та LinkedList.
5	Порівняти час 100 000 операцій get(i) у ArrayList та LinkedList.

Питання 28. HashSet проти TreeSet

Теоретична частина

HashSet — реалізація Set на основі HashMap. Порядок ітерації не визначений — залежить від хешів елементів. Операції add, remove, contains — амортизована O(1) при хорошій хеш-функції. Допускає один null [60].

TreeSet — реалізація NavigableSet на основі red-black tree. Зберігає елементи в природному порядку (Comparable) або за Comparator. Операції — O(log n). Не дозволяє null (кидає NullPointerException).

Перевага TreeSet — навігаційні методи: first(), last(), floor(e), ceiling(e), higher(e), lower(e), а також діапазонні subSet, headSet, tailSet. Ідеальний для задач на діапазони та впорядкований обхід. LinkedHashSet — третій варіант: HashMap + двозв'язний список. Зберігає порядок вставки, операції майже O(1).

Критичне правило для HashSet: елементи мають коректно реалізовувати equals та hashCode. Для TreeSet — елементи мають бути порівнянними.

Приклад задачі

Умова. Порівняти поведінку HashSet та TreeSet на однакових вхідних даних — номерах пожежно-рятувальних частин.

Порівняння у файлі Main.java.

Лістинг 28.1. Реалізація з покроковим розбором

Файл `Main.java` HashSet vs TreeSet

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    Set<Integer> hs = new HashSet<>(List.of(3, 1,
2));
    Set<Integer> ts = new TreeSet<>(List.of(3, 1,
2));
    System.out.println(hs);
    System.out.println(ts);
```

HashSet не гарантує порядку (часто [1,2,3] для малих int, але це випадковість). TreeSet завжди впорядкований за зростанням: [1, 2, 3]. Не покладайтеся на видимий порядок HashSet.

```
    TreeSet<Integer> t = (TreeSet<Integer>) ts;
    System.out.println(t.first());
    System.out.println(t.ceiling(2));
  }
}
```

TreeSet дає навігаційні методи: first() повертає найменший елемент (1), ceiling(2) — найменший елемент >= 2 (це 2). Таких методів немає у HashSet — це ключова перевага TreeSet.

Зверніть увагу. *Порядок у HashSet може змінитися навіть між запусками на різних версіях JVM. Тестувати код, що покладається на цей порядок, — гарантовано отримати «плаваючі» помилки.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java продемонструвати роботу з множиною згідно теми варіанту.

№	Що реалізувати
1	Додати 10 випадкових номерів викликів у HashSet та TreeSet; порівняти порядки.
2	Спробувати додати null у TreeSet; перехопити NullPointerException.
3	Створити TreeSet прізвищ курсантів з власним Comparator (за спаданням).
4	Використати TreeSet.subSet(2, 5) для отримання діапазону поверхів.
5	Створити клас Rescuer з equals/hashCode та додати у HashSet.

Питання 29. Назвіть три методи інтерфейсу List

Теоретична частина

Інтерфейс List<E> представляє впорядковану колекцію з доступом за індексом і дозволеними дублікатами. Найбільш фундаментальні методи: boolean add(E e) — додає в кінець; void add(int index, E e) — вставляє на позицію; E get(int index) — повертає за індексом; E set(int index, E e) — замінює і повертає старий; E remove(int index) — видаляє за індексом; boolean remove(Object o) — видаляє за значенням [16].

Допоміжні методи: int size(); boolean isEmpty(); boolean contains(Object o); int indexOf(Object o); void clear(); List<E> subList(int from, int to). Конверсії: Object[] toArray() та T[] toArray(T[] a); Iterator<E> iterator() та ListIterator<E> listIterator() (двонапрямний).

Сучасні (Java 8+): forEach(Consumer), removeIf(Predicate), replaceAll(UnaryOperator), sort(Comparator), stream(). Фабрики: List.of(...), List.copyOf(coll), Stream.toList(). Java 21: SequencedCollection додає getFirst(), getLast(), addFirst(E), addLast(E), reversed().

На співбесіді достатньо назвати add, get, remove і коротко описати їхні варіанти.

Приклад задачі

Умова. Продемонструвати три фундаментальні методи List на списку підрозділів, що прямують на виклик: add, get, remove.

Демонстрація у файлі Main.java.

Лістинг 29.1. Реалізація з покроковим розбором

Файл `Main.java` add, get, remove

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("АЦ-40");
    list.add("АД-30");
    list.add(0, "ОШ");
```

Метод add має дві форми: без індексу (додає в кінець) та з індексом (вставляє на позицію, зсуваючи решту). Першим ставимо оперативний штаб (ОШ). Після цих операцій список: [ОШ, АЦ-40, АД-30].

```
    System.out.println(list.get(1));
    list.remove("АЦ-40");
    System.out.println(list);
  }
}
```

get(1) повертає елемент за індексом — «АЦ-40». remove("АЦ-40") видаляє за значенням (Object). Вивід: «АЦ-40», потім «[ОШ, АД-30]». Увага: remove(Integer) трактується як індекс — часта пастка.

Зверніть увагу. *Пастка зі списком цілих: для List<Integer> виклик list.remove(1) видалить елемент з індексом 1, а не число 1. Для видалення значення пишіть list.remove(Integer.valueOf(1)).*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java виконати операції зі списком згідно теми варіанту.

№	Що реалізувати
1	Створити список караулу з 5 прізвищ, видалити третього за індексом.
2	Знайти indexOf для курсанта, якого немає у списку взводу; пояснити результат -1.
3	Замінити автомобіль на позиції 2 через set(); вивести замінений.
4	Створити subList(1, 3) списку чергових і змінити його; пояснити вплив на оригінал.
5	Видалити всі виклики, адреса яких починається з «вул.», через removeIf.

Питання 30. Як працює remove(Object) у Set<Product>

Теоретична частина

Метод remove(Object o) у Set виконує пошук елемента, рівного o (за методом equals), і видаляє його. Повертає boolean. Механіка пошуку залежить від реалізації [1; 10].

Для HashSet: спочатку обчислюється hashCode об’єкта o, він мапиться у номер ведра. Потім всередині ведра — пошук за equals. O(1) у середньому. Для TreeSet: пошук по red-black tree через compareTo (або Comparator); equals не використовується безпосередньо. TreeSet вважає елементи рівними, якщо compareTo повертає 0. O(log n).

Якщо клас Product не перевизначив hashCode/equals, remove працює за Object.equals (порівняння посилань). Тобто set.remove(new Product("A")) не знайде «такого ж» Product, доданого через інший new Product("A") — це різні об’єкти.

Правило: для коректної роботи Set ОБОВ’ЯЗКОВО перевизначити hashCode і equals узгоджено.

Це найчастіша помилка новачків — створюють клас, кладуть у Set, дивуються, чому remove не працює.

Приклад задачі

Умова. Продемонструвати, як працює remove у HashSet без перевизначення equals/hashCode, на прикладі обліку спорядження.

Клас Equipment у файлі Equipment.java; демонстрація проблеми — у Main.java.

Лістинг 30.1. Реалізація з покроковим розбором

Файл `Equipment.java, Main.java` Equipment без перевизначення equals/hashCode

```
class Equipment {
    String name;
    Equipment(String n) { name = n; }
}

public class Main {
  public static void main(String[] args) {
    Set<Equipment> s = new HashSet<>();
    s.add(new Equipment("Апарат"));
```

Клас Equipment НЕ перевизначає equals і hashCode. Додаємо один об’єкт Equipment("Апарат") — дихальний апарат. Стандартний Object.equals порівнює посилання, а не вміст полів.

```
    s.remove(new Equipment("Апарат"));
    System.out.println(s.size());
  }
}
```

Намагаємося видалити «такий же» Equipment("Апарат") — але це ІНШИЙ об’єкт у пам’яті. Object.equals дає false, remove нічого не знаходить. Вивід: «1» — елемент НЕ видалено. З коректним equals/hashCode за name видалення спрацювало б.

З досвіду викладачів. *Для IDE-середовищ це не проблема: IntelliJ IDEA генерує equals та hashCode автоматично (Alt+Insert). Але пояснити, що саме згенеровано, ви маєте вміти самі.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати клас у окремому файлі та продемонструвати роботу remove у Set згідно теми варіанту.

№	Що реалізувати
1	Equipment з equals/hashCode за полем inventoryId; перевірити коректний remove.
2	HashSet<Equipment> без override; показати, що remove не працює.
3	TreeSet<Rescuer> з Comparable (за стажем); пояснити роль compareTo.
4	Equipment, де equals за name, а hashCode за inventoryId; продемонструвати баг.
5	Видалити списане спорядження через iterator.remove() замість Set.remove.

Питання 31. InputStream/OutputStream, Reader/Writer та їхні реалізації

Теоретична частина

Java надає два паралельних API для I/O: байтовий і символьний. Байтовий — InputStream/OutputStream — працює з byte. Символьний — Reader/Writer — працює з char (Unicode). Між ними мости: InputStreamReader та OutputStreamWriter з charset. Реалізації InputStream: FileInputStream (з файлу), ByteArrayInputStream (з масиву), BufferedInputStream (буферизація), DataInputStream (типізовані примітиви), ObjectInputStream (серіалізація). OutputStream — симетричні [3].

Реалізації Reader: FileReader, StringReader, BufferedReader (метод readLine!), InputStreamReader (міст з charset). Writer: FileWriter, StringWriter, BufferedWriter, PrintWriter.

Правило: для текстових даних — Reader/Writer з явним charset (UTF-8). Для бінарних — InputStream/OutputStream. Завжди закривати через try-with-resources.

Сучасний підхід (NIO.2, Java 7+): Files та Path — Files.readAllLines(path), Files.lines(path), Files.newBufferedReader(path). Часто кращий вибір.

Приклад задачі

Умова. Прочитати текстовий файл журналу викликів та записати підсумок у новий файл через BufferedReader/BufferedWriter з try-with-resources.

Читання та запис у файлі Main.java.

Лістинг 31.1. Реалізація з покроковим розбором

Файл `Main.java` читання файлу з буферизацією

```
import java.nio.file.*;
import java.io.*;

public class Main {
  public static void main(String[] args) throws
IOException {
    try (BufferedReader r =
         Files.newBufferedReader(Path.of("calls.tx
t"))) {
        String line;
        while ((line = r.readLine()) != null)
            System.out.println(line);
    }
```

Files.newBufferedReader створює BufferedReader з UTF-8 за замовчуванням. try-with-resources гарантує виклик close() навіть при винятку. readLine() повертає null у кінці файлу — стандартна ідіома циклу.

```
    try (BufferedWriter w =
         Files.newBufferedWriter(Path.of("report.t
xt"))) {
        w.write("Чергування завершено");
        w.newLine();
    }
  }
}
```

Files.newBufferedWriter створює BufferedWriter. Метод write записує рядок, newLine() додає перенос рядка (платформонезалежний). try-with-resources автоматично закриває потік і скидає буфер.

Зверніть увагу. *Якщо файлу calls.txt не існує, програма завершиться з NoSuchFileException. Для навчального прикладу цього досить, але в реальній системі чергової частини таку ситуацію треба обробити.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java реалізувати операцію введення-виведення згідно теми варіанту.

№	Що реалізувати
1	Прочитати файл даних датчика температури побайтово через FileInputStream та вивести байти.
2	Записати рядок донесення про пожежу у файл через FileWriter.
3	Скопіювати файл плану евакуації, читаючи блоками по 4096 байт через буфер.
4	Прочитати журнал викликів та підрахувати кількість рядків (викликів).
5	Створити DataOutputStream та записати номер частини (int), площу пожежі (double), адресу (String).

Питання 32. Прочитати всі рядки файлу та вивести у консоль

Теоретична частина

Існує декілька канонічних способів прочитати файл порядково; вибір залежить від розміру файлу [3; 16].

Найпростіший (Java 7+): Files.readAllLines(Path) повертає List<String> усіх рядків. Підходить для невеликих файлів, бо повністю завантажує вміст у пам'ять. Для файлу на 10 ГБ — OutOfMemoryError. Для великих файлів: Files.lines(Path) повертає Stream<String> з лінивою оцінкою — рядки читаються по мірі споживання. ОБОВ'ЯЗКОВО в try-with-resources: Stream тримає файловий дескриптор.

Класичний спосіб: BufferedReader з циклом readLine() — найгнучкіший, дає контроль над читанням. readLine повертає null у кінці файлу.

З Java 11+ є Files.readString(Path) — весь файл як один String. Для маленьких файлів найкомпактніше. Усі методи за замовчуванням використовують UTF-8.

Приклад задачі

Умова. Прочитати файл зі списком курсантів навчальної групи усіма трьома основними способами та вивести в консоль.

Три способи в одному файлі Main.java.

Лістинг 32.1. Реалізація з покроковим розбором

Файл `Main.java` readAllLines та Files.lines

```
import java.nio.file.*;

public class Main {
  public static void main(String[] args) throws
Exception {
    Files.readAllLines(Path.of("cadets.txt"))
         .forEach(System.out::println);

    try (var s =
Files.lines(Path.of("cadets.txt"))) {
        s.forEach(System.out::println);
    }
```

Спосіб 1 — readAllLines: читає всі рядки в List і виводить. Просто, але для великих файлів небезпечно (пам'ять). Спосіб 2 — Files.lines: лінивий Stream, читає по мірі споживання; try-with-resources закриває файловий дескриптор.

```
    try (var r =
         Files.newBufferedReader(Path.of("cadets.t
xt"))) {
        String line;
        while ((line = r.readLine()) != null)
            System.out.println(line);
    }
  }
}
```

Спосіб 3 — класичний BufferedReader: найбільш гнучкий, дає контроль над кожним рядком. readLine() повертає null наприкінці файлу. Універсальний підхід для будь-якого розміру файлу.

Зверніть увагу. *Кодування за замовчуванням у методах Files — UTF-8, тому прізвища курсантів кирилицею прочитаються коректно. Зі старими класами FileReader до Java 18 варто явно вказувати кодування.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java прочитати файл та виконати обробку згідно теми варіанту.

№	Що реалізувати
1	Прочитати журнал викликів та вивести лише рядки, довші за 10 символів.
2	Прочитати файл донесення та підрахувати кількість рядків і символів.
3	Знайти найдовшу адресу у файлі адрес виїздів та вивести її.
4	Прочитати CSV-файл «прізвище;взвод;бал» та вивести лише перший стовпець кожного рядка.
5	Прочитати файл особового складу караулу та вивести нумерований список рядків ("1: ...").

ЧАСТИНА 2. БАГАТОПОТОЧНІСТЬ ТА БАЗИ ДАНИХ

Розділ 3. Багатопоточність

Питання 33. Створення та запуск потоку. Пауза. join. Race condition

Теоретична частина

У Java потоки виконання представлені класом java.lang.Thread. Існують два канонічні способи створити потік. Перший — успадкувати Thread і перевизначити run(). Другий — реалізувати Runnable і передати у конструктор Thread: Thread t = new Thread(() -> { ... }). Другий спосіб рекомендований, бо Java підтримує лише одиничне наслідування класів. Принципово розрізняти start() і run(). Виклик t.start() створює новий потік ОС і викликає run() у його контексті. Виклик t.run() напряму НЕ створює нового потоку — метод виконується синхронно у поточному потоці. Це часта пастка співбесід [4].

Пауза реалізується статичним методом Thread.sleep(ms), що призупиняє поточний потік. Кидає checked InterruptedException. Важливо: sleep не звільняє монітори — synchronized-лок залишається заблокованим.

Метод join() — instance-метод, викликається на іншому потоці й блокує поточний до завершення цільового. Шаблон: t.start(); t.join().

Race condition (стан перегонів) — результат залежить від непередбачуваного порядку виконання потоків. Класика: два потоки інкрементують спільну змінну counter++. Ця операція не атомарна (читання, інкремент, запис) — частина інкрементів губиться. Запобігання — синхронізація, AtomicInteger, immutable-дані.

Приклад задачі

Умова. Запустити окремий потік «сирена», що тричі подає сигнал тривоги, дочекатися його завершення через join і вивести повідомлення про виїзд караулу з main.

Демонстрація у файлі Main.java з використанням лямбди як Runnable.

Лістинг 33.1. Реалізація з покроковим розбором

Файл `Main.java` створення потоку, start, join

```
public class Main {
  public static void main(String[] args) {
    Thread t = new Thread(() -> {
        for (int i = 0; i < 3; i++) {
            System.out.print("!");
        }
    });
```

Лямбда (() -> {...}) реалізує функціональний інтерфейс Runnable. Створюємо потік t, передавши йому цю задачу. Поки що потік лише створено — він ще не запущений.

```
    t.start();
    try {
        t.join();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    System.out.println(" караул виїхав");
  }
}
```

t.start() запускає новий потік (саме start, не run!). t.join() блокує main, поки t не завершиться. Тому «караул виїхав» виведеться ПІСЛЯ «!!!». Без join() порядок виводу був би непередбачуваним.

Порада. *Запустіть програму без t.join() кілька разів поспіль. Порядок виводу може змінюватися — це і є недетермінованість багатопотокових програм.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Main.java реалізувати роботу з потоками згідно теми варіанту.

№	Що реалізувати
1	Створити клас SirenThread через успадкування класу Thread і запустити його.
2	Запустити три потоки-«караули», що друкують свої імена з паузою 200 мс через sleep.
3	Продемонструвати race condition на спільному лічильнику викликів без синхронізації.
4	Виправити race condition попереднього варіанту через AtomicInteger.
5	Викликати run() потоку-сирени напряму без start(); пояснити, чому новий потік не створюється.

Питання 34. Як працюють synchronized-методи

Теоретична частина

Ключове слово synchronized забезпечує взаємне виключення (mutual exclusion) між потоками. У будь-який момент лише один потік може виконувати synchronized-блок або метод над одним монітором (внутрішнім локом об’єкта). Інші потоки переходять у стан BLOCKED і чекають [4; 15].

Synchronized instance method блокується на моніторі this. Два потоки, що викликають synchronized-метод на ОДНОМУ екземплярі, чекають один одного; на РІЗНИХ — виконуються паралельно. Synchronized static method блокується на моніторі об’єкта Class (ClassName.class). Це окремий монітор від екземплярного.

Synchronized-блок (synchronized(obj) {...}) гнучкіший — явно вказуємо монітор. Найкраща практика — приватний final Object lock = new Object().

Окрім виключення synchronized гарантує видимість пам’яті (happens-before): усі записи перед виходом із блоку видимі іншому потоку, що входить у блок на тому ж моніторі. Synchronized також re-entrant: потік, що володіє монітором, може повторно увійти.

Приклад задачі

Умова. Реалізувати потокобезпечний лічильник викликів чергової частини через synchronized-методи та продемонструвати коректний підсумок, коли виклики реєструють паралельно два диспетчери (два потоки).

Клас CallCounter у файлі CallCounter.java; запуск двох потоків — у Main.java.

Лістинг 34.1. Реалізація з покроковим розбором

Файл `CallCounter.java` synchronized-методи inc та get

```
class CallCounter {
    private int v = 0;

    public synchronized void inc() {
        v++;
    }
    public synchronized int get() {
        return v;
    }
}
```

Обидва методи позначені synchronized — вони блокуються на моніторі this. Якщо два потоки одночасно викликають inc() на одному об’єкті, вони серіалізуються: один чекає, поки інший завершить. Це усуває race condition.

Файл `Main.java` паралельна реєстрація двома диспетчерами

```
public class Main {
  public static void main(String[] args)
          throws InterruptedException {
    CallCounter c = new CallCounter();
    Runnable task = () -> {
        for (int i = 0; i < 1000; i++) c.inc();
    };
    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);
```

Два потоки-диспетчери виконують однакову задачу: по 1000 реєстрацій викликів у спільному лічильнику. Без synchronized тут виникла б гонка і втрата частини записів.

```
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println(c.get());
  }
}
```

Запускаємо обидва потоки, чекаємо завершення через join. Завдяки synchronized підсумок ЗАВЖДИ 2000. Без синхронізації результат був би менший і непередбачуваний.

Зверніть увагу. *Спробуйте прибрати synchronized з методу inc() і запустити програму 5–10 разів. Найімовірніше, ви побачите різні числа, менші за 2000.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати потокобезпечний клас із synchronized у окремому файлі та продемонструвати його роботу з кількох потоків у Main.java.

№	Що реалізувати
1	Stack донесень з synchronized-методами push і pop; використати з двох потоків.
2	synchronized-блок з власним приватним об’єктом-локом для журналу чергування замість this.
3	Продемонструвати deadlock: два караули (потоки) захоплюють lockPump та lockHose у різному порядку.
4	Порівняти час synchronized-методу та AtomicInteger.incrementAndGet для лічильника викликів.
5	Потокобезпечний журнал чергування з synchronized для уникнення перемішування рядків.

Розділ 4. Реляційні бази даних і мова SQL

Питання 35. Структура RDBMS. Нормалізація. Цілісність даних

Теоретична частина

Реляційна СУБД (RDBMS) — програмний сервер, що зберігає дані у вигляді пов’язаних таблиць. Представники: MySQL, PostgreSQL, Oracle, MS SQL Server. Сервер працює як процес (IP + порт), приймає мережні з’єднання і виконує SQL-запити. Логічна структура: сервер містить одну чи кілька баз даних (database). База — набір таблиць та інших об’єктів (індекси, представлення, процедури). Таблиця має стовпці (з типом і обмеженнями) та рядки (записи) [24; 25].

Нормалізація — проектування схеми для мінімізації дублювання й аномалій. 1НФ — атомарні значення, унікальні рядки; 2НФ — кожен неключовий атрибут залежить від усього ключа; 3НФ — відсутність транзитивних залежностей. На практиці зазвичай досягають 3НФ. Денормалізація — свідоме порушення заради продуктивності читання.

Цілісність даних — гарантії коректності з боку СУБД: сутнісна (PRIMARY KEY), посилкова (FOREIGN KEY), доменна (NOT NULL, CHECK, тип), користувацька (тригери).

Перевага декларативних обмежень: помилки виявляються на рівні БД незалежно від клієнта — сильніша гарантія, ніж лише валідація у коді.

Приклад задачі

Умова. Створити нормалізовану схему для обліку техніки ДСНС Львівщини: окремі таблиці fire_station (пожежно-рятувальна частина) та fire_truck (автомобіль) зі зв'язком за FOREIGN KEY.

SQL-скрипт у файлі schema.sql.

Лістинг 35.1. Реалізація з покроковим розбором

Файл `schema.sql` дві нормалізовані таблиці

```sql
CREATE TABLE fire_station (
    id   INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL
);
```

Таблиця fire_station зберігає частини окремо. PRIMARY KEY id однозначно ідентифікує частину. AUTO_INCREMENT генерує id автоматично. Завдяки окремій таблиці назва частини зберігається в одному місці (3НФ).

```sql
CREATE TABLE fire_truck (
    id         INT PRIMARY KEY AUTO_INCREMENT,
    model      VARCHAR(200) NOT NULL,
    station_id INT,
    FOREIGN KEY (station_id)
        REFERENCES fire_station(id)
);
```

Таблиця fire_truck посилається на fire_station через FOREIGN KEY station_id. Це гарантує посилкову цілісність: не можна вставити автомобіль зі station_id, якого немає у fire_station. Зміна назви частини потребує оновлення лише одного рядка.

Зверніть увагу. *Імена таблиць ми записали англійською в однині (fire_station, fire_truck). Конкретна конвенція не так важлива, як її послідовне дотримання в усій схемі.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі schema.sql спроєктувати нормалізовану схему згідно теми варіанту з дотриманням 3НФ та обмежень цілісності.

№	Предметна область
1	Бібліотека ЛДУБЖД: книги, читачі (курсанти, студенти), видачі.
2	Університет: інститут, кафедра, викладач.
3	Облік спорядження: предмет, категорія, видача рятувальнику.
4	Навчальний процес: курсант, навчальна група, оцінки.
5	Чергова частина: виклик, підрозділ, виїзд.

Питання 36. PRIMARY KEY та FOREIGN KEY

Теоретична частина

PRIMARY KEY (первинний ключ) — стовпець або комбінація, що однозначно ідентифікує кожен рядок. Має обмеження NOT NULL та UNIQUE одночасно. У таблиці лише один PRIMARY KEY. Найчастіше — цілі числа з AUTO_INCREMENT (MySQL) або SERIAL (PostgreSQL). Два типи ключів: сурогатний (штучний, без бізнес-сенсу — id) та

натуральний (з бізнес-даних — ISBN, номер паспорта). На практиці частіше сурогатний: бізнес-дані можуть змінюватися, id стабільний [26].

FOREIGN KEY (зовнішній ключ) — стовпець, значення якого посилається на PRIMARY KEY іншої (або тієї ж) таблиці. Створює зв'язок і дає посилкову цілісність: не можна вставити рядок з FK без відповідного PK; не можна видалити PK, на який є посилання (без CASCADE/SET NULL).

Поведінка налаштовується ON DELETE / ON UPDATE: NO ACTION (заборонити), CASCADE (каскадно видалити), SET NULL (записати NULL), RESTRICT. Вибір залежить від семантики.

FOREIGN KEY вимагає, щоб цільовий стовпець був PRIMARY KEY або мав UNIQUE-обмеження.

Приклад задачі

Умова. Створити дві пов'язані таблиці: department (кафедра університету) та lecturer (викладач) з FOREIGN KEY та поведінкою ON DELETE SET NULL.

SQL-скрипт у файлі schema.sql.

Лістинг 36.1. Реалізація з покроковим розбором

Файл `schema.sql` PRIMARY KEY та FOREIGN KEY зі SET NULL

```
CREATE TABLE department (
    id   INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL
);
```

Таблиця department з первинним ключем id. Це батьківська таблиця — на її id посилатимуться викладачі. PRIMARY KEY гарантує унікальність та NOT NULL.

```
CREATE TABLE lecturer (
    id      INT PRIMARY KEY AUTO_INCREMENT,
    name    VARCHAR(100) NOT NULL,
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES department(id)
        ON DELETE SET NULL
);
```

lecturer.dept_id — FOREIGN KEY на department.id. Не можна вставити lecturer з неіснуючим dept_id. При видаленні (реорганізації) кафедри dept_id у пов'язаних викладачів стане NULL завдяки ON DELETE SET NULL.

З досвіду викладачів. *Вибір між SET NULL, CASCADE і RESTRICT — не технічне, а змістове рішення. Для кафедри логічно SET NULL: викладач не зникає разом зі структурним підрозділом.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі schema.sql створити дві пов'язані таблиці з PRIMARY KEY та FOREIGN KEY згідно теми варіанту; продемонструвати поведінку при видаленні.

№	Таблиці та зв'язок
1	fire_station та fire_call (FK station_id); ON DELETE CASCADE.
2	category та equipment (FK category_id); ON DELETE SET NULL.
3	incident та incident_unit (FK incident_id); ON DELETE CASCADE.
4	study_group та cadet (FK group_id); ON DELETE RESTRICT.
5	unit з self-reference (FK parent_id: частина → загін → управління).

Питання 37. Що таке індекс. Переваги і недоліки

Теоретична частина

Індекс (index) — окрема структура даних, що зберігає відсортовані значення стовпців разом із покажчиками на рядки таблиці. Призначення — прискорення пошуку, фільтрації, сортування, з'єднання [29; 30].

Найпоширеніший тип — B-tree (O(log n)). Є також хеш-індекси, повнотекстові.

Переваги: SELECT з WHERE по індексованому стовпцю — O(log n) замість O(n) повного сканування. JOIN по індексованих стовпцях швидший. ORDER BY індексованих стовпців не вимагає сортування. Унікальний індекс служить обмеженням UNIQUE.

Недоліки: 1) додаткове місце на диску; 2) уповільнення INSERT/UPDATE/DELETE (індекс треба оновлювати); 3) не для кожного запиту корисний — для малих таблиць повне сканування може бути швидшим.

PRIMARY KEY автоматично створює унікальний індекс. UNIQUE теж. FOREIGN KEY у MySQL InnoDB автоматично індексується. Складений індекс на (a, b, c) обслуговує запити по a, (a, b), (a, b, c), але не по b чи c окремо — правило «лівого префіксу». Аналіз — команда EXPLAIN.

Приклад задачі

Умова. Створити різні типи індексів для таблиці cadet (курсант) та журналу fire_call: звичайний, унікальний та складений.

SQL-скрипт у файлі indexes.sql.

Лістинг 37.1. Реалізація з покроковим розбором

Файл `indexes.sql` три типи індексів

```sql
-- Звичайний індекс
CREATE INDEX idx_cadet_email
    ON cadet(email);

-- Унікальний індекс
CREATE UNIQUE INDEX idx_cadet_login
    ON cadet(login);
```

Звичайний індекс прискорює WHERE email = ... до O(log n). Унікальний індекс додатково гарантує, що login не повторюється — служить одночасно обмеженням UNIQUE.

```sql
-- Складений індекс
CREATE INDEX idx_call_date_station
    ON fire_call(date, station_id);
```

Складений індекс ефективний для запитів по date або по (date, station_id), але НЕ самостійно по station_id (правило лівого префіксу). Команда EXPLAIN SELECT покаже, чи план використовує індекс.

Зверніть увагу. *Не створюйте індекси «про запас» на кожен стовпець. Кожен індекс уповільнює вставку, а в журналі викликів вставок дуже багато.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі indexes.sql створити індекс згідно теми варіанту та через EXPLAIN перевірити його використання.

№	Що реалізувати
1	Звичайний індекс на cadet.email; перевірити EXPLAIN для SELECT за email.
2	Унікальний індекс на cadet.record_book (номер залікової книжки); спробувати вставити дублікат.
3	Складений індекс (category, model) на equipment; перевірити лівий префікс.
4	Індекс на FOREIGN KEY fire_call.station_id; пояснити навіщо.
5	Порівняти час SELECT за неіндексованим стовпцем на великій таблиці викликів.

Питання 38. Синтаксис SELECT

Теоретична частина

SELECT — основна інструкція вибірки. Синтаксис: SELECT [DISTINCT] стовпці FROM таблиця [JOIN ...] [WHERE умова] [GROUP BY ...] [HAVING ...] [ORDER BY ...] [LIMIT n [OFFSET m]]. Логічний порядок виконання інакший за написання: FROM/JOIN → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT. Псевдонім зі SELECT не можна використати у WHERE, але можна в ORDER BY [27].

SELECT * повертає всі стовпці — зручно для дослідження, небажано в продакшені. DISTINCT видаляє дублікати (дорога операція).

WHERE приймає логічний вираз: =, <>, <, >, BETWEEN, IN, LIKE (% та _), IS NULL, AND, OR, NOT. Порівняння з NULL через = не працює — для NULL використовується IS NULL.

ORDER BY сортує (ASC/DESC). LIMIT n обмежує кількість, OFFSET m пропускає перші m рядків — основа пагінації.

Приклад задачі

Умова. Виконати запит з кількома клаузами: знайти топ-10 рятувальників, які перебувають на службі та мають понад 5 виїздів, за спаданням кількості виїздів.

SQL-запит у файлі query.sql.

Лістинг 38.1. Реалізація з покроковим розбором

Файл `query.sql` SELECT з усіма основними клаузами

```
SELECT u.name,
       COUNT(o.id) AS trips_count
FROM   rescuer u
LEFT JOIN trip o
    ON o.rescuer_id = u.id
WHERE  u.active = 1
```

Спочатку FROM + JOIN об’єднує rescuer та trip. WHERE відсікає рятувальників, які не перебувають на службі, ДО групування. COUNT(o.id) рахує виїзди кожного рятувальника (псевдонім trips_count).

```
GROUP BY u.id, u.name
HAVING COUNT(o.id) > 5
ORDER BY trips_count DESC
LIMIT 10;
```

GROUP BY групує по рятувальнику. HAVING залишає лише групи з понад 5 виїздами (фільтр після агрегації). ORDER BY сортує за спаданням. LIMIT 10 обмежує результат топ-десяткою.

Зверніть увагу. *Зверніть увагу, що в GROUP BY вказано і u.id, і u.name. Групування лише за прізвищем об'єднало б двох рятувальників-однофамільців в одного.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі query.sql скласти SELECT-запит згідно теми варіанту.

№	Що вибрати
1	Усі виклики з area > 100 (площа пожежі, м²), відсортовані за площею за спаданням.
2	Унікальні (DISTINCT) райони виїздів, відсортовані за алфавітом.
3	Топ-5 найтриваліших гасінь через ORDER BY + LIMIT.
4	Курсантів, прізвище яких починається з 'К', через LIKE.
5	Виклики, де phone IS NULL (анонімні), та порахувати їх кількість.

Питання 39. Синтаксис INSERT

Теоретична частина

INSERT — інструкція вставки рядка. Синтаксис: INSERT INTO таблиця (стовпці) VALUES (значення). Перелік стовпців краще вказувати явно — інакше при зміні схеми код мовчки зламається. Пакетна вставка — кілька рядків в одному INSERT: VALUES (v1, v2), (v3, v4). Значно ефективніше за окремі INSERT [31].

INSERT ... SELECT — вставка з результату запиту. Для міграцій, копіювання в архів. Upsert — «вставити або оновити». MySQL: INSERT ... ON DUPLICATE KEY UPDATE. PostgreSQL: INSERT ... ON CONFLICT DO UPDATE.

При AUTO_INCREMENT значення PK не вказується — БД генерує сама. Згенерований ID: LAST_INSERT_ID() (MySQL) або RETURNING id (PostgreSQL).

Приклад задачі

Умова. Продемонструвати три варіанти INSERT для бази даних університету: один рядок, пакетна вставка та INSERT з SELECT.

SQL-скрипт у файлі inserts.sql.

Лістинг 39.1. Реалізація з покроковим розбором

Файл `inserts.sql` одиничний та пакетний INSERT

```
-- Один рядок
INSERT INTO cadet(name, email)
VALUES ('Петренко', 'petrenko@ldubgd.edu.ua');

-- Пакетна вставка
INSERT INTO alarm_log(level, msg) VALUES
    ('INFO',  'датчик ОК'),
    ('WARN',  'задимлення'),
    ('ERROR', 'пожежа');
```

Перший INSERT додає один рядок з явним переліком стовпців. Другий — три рядки за один запит (пакетна вставка). Це значно швидше за три окремі INSERT, бо менше мережних звернень.

```
-- INSERT з SELECT
INSERT INTO graduate (id, name)
SELECT id, name FROM cadet
WHERE active = 0;
```

INSERT ... SELECT копіює курсантів, що завершили навчання, в таблицю випускників за одну операцію. Корисно для міграцій та перенесення даних між таблицями.

Порада. *Пакетну вставку зручно використовувати для початкового заповнення навчальної бази даних. На лабораторних заняттях це економить час на підготовку тестових даних.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі inserts.sql реалізувати вставку згідно теми варіанту.

№	Що реалізувати
1	Вставити п’ять рядків у таблицю equipment одним пакетним INSERT.
2	Створити таблицю fire_call з AUTO_INCREMENT та отримати згенерований ID виклику.
3	Скопіювати закриті виклики з fire_call у fire_call_archive через INSERT ... SELECT.
4	Реалізувати upsert через ON DUPLICATE KEY UPDATE для лічильника виїздів частини.
5	Спробувати INSERT виклику з пропуском NOT NULL стовпця address; пояснити помилку.

Питання 40. UPDATE, DELETE. JOIN — навіщо потрібен

Теоретична частина

UPDATE — оновлення рядків. Синтаксис: UPDATE таблиця SET стовпець = значення [WHERE умова]. Без WHERE оновлює ВСІ рядки — класична катастрофа продакшену. Захисна практика: спочатку SELECT з тією ж умовою. Кожен, хто хоч раз випадково оновив усю таблицю, пам’ятає цей момент назавжди [27; 28].

DELETE — видалення рядків. Синтаксис: DELETE FROM таблиця [WHERE умова]. Без WHERE видаляє ВСЕ. Альтернатива для всіх рядків — TRUNCATE (швидша, але не транзакційна).

UPDATE та DELETE можуть використовувати JOIN для модифікації за умовами з пов’язаних таблиць. JOIN потрібен, бо в нормалізованій схемі дані розкидані по таблицях. Щоб отримати інформацію, що поєднує сутності (замовлення з даними клієнта), треба з’єднати таблиці за пов’язуючим ключем.

Альтернатива JOIN — підзапит у WHERE. Але JOIN зазвичай ефективніший і читабельніший, для агрегації з різних таблиць — практично безальтернативний.

Приклад задачі

Умова. Продемонструвати UPDATE з умовою, DELETE з умовою та SELECT з JOIN на пов’язаних таблицях бази даних чергової частини.

SQL-скрипт у файлі queries.sql.

Лістинг 40.1. Реалізація з покроковим розбором

Файл `queries.sql` UPDATE та DELETE з умовою	

`-- Позначити прострочену повірку`
`UPDATE extinguisher`
`SET    valid = 0`
`WHERE  checked < '2024-01-01';`
 	
`-- Видалити старі записи сигналізації`
`DELETE FROM alarm_log`
`WHERE  created < NOW() - INTERVAL 30 DAY;`
	UPDATE оновлює лише вогнегасники, повірка яких була до 2024 р. (завдяки WHERE). DELETE видаляє записи журналу сигналізації старше 30 днів. КРИТИЧНО мати WHERE — без нього UPDATE/DELETE зачепили б усі рядки.

`-- Поєднати дані через JOIN`
`SELECT u.name, o.area`
`FROM   fire_station u`
`JOIN   fire_call o`
`    ON o.station_id = u.id`
`WHERE  o.created > '2024-01-01';`
	JOIN з'єднує fire_station та fire_call за station_id, повертаючи поля з обох таблиць. Це потрібно, бо дані нормалізовані: назва частини в fire_station, площа пожежі в fire_call. WHERE фільтрує свіжі виклики.

Зверніть увагу. *Перед кожним UPDATE чи DELETE корисно виконати SELECT з тією ж умовою WHERE. Так ви побачите, які саме рядки зміните, ще до того, як буде пізно.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі queries.sql реалізувати запит згідно теми варіанту.

№	Що реалізувати
1	Збільшити норму витрати води на 10% для всіх об'єктів категорії 'склад' через UPDATE.
2	Видалити облікові записи курсантів, відрахованих понад два роки тому.
3	UPDATE з JOIN: оновити fire_call.district за даними з fire_station.
4	DELETE з підзапитом для видалення сирітських рядків (виїздів без виклику).
5	SELECT з JOIN трьох таблиць (fire_station, fire_call, trip).

Питання 41. INNER JOIN, OUTER JOIN, LEFT/RIGHT JOIN

Теоретична частина

JOIN — операція з'єднання таблиць за умовою. Чотири основні типи. INNER JOIN (або просто JOIN) — лише рядки, де є збіг в обох таблицях за умовою ON. Якщо в одній немає відповідного рядка — пара виключається [25].

LEFT OUTER JOIN (LEFT JOIN) — ВСІ рядки з лівої таблиці плюс збіги з правої. Де немає збігу — поля правої NULL. Класика: «усі клієнти і їхні замовлення, навіть ті, хто нічого не замовляв».

RIGHT OUTER JOIN — дзеркальна версія LEFT. Рідко на практиці — переписують як LEFT зі зміною порядку. FULL OUTER JOIN — усі рядки з обох; MySQL не підтримує (емуляція через UNION).

CROSS JOIN — декартів добуток без ON. SELF JOIN — з'єднання таблиці з собою через псевдоніми (ієрархії).

Приклад задачі

Умова. Порівняти результати INNER JOIN та LEFT JOIN на парі таблиць fire_station (частини) та fire_call (виклики).

SQL-запити у файлі joins.sql.

Лістинг 41.1. Реалізація з покроковим розбором

Файл `joins.sql` INNER JOIN vs LEFT JOIN

```
-- INNER: лише частини з викликами
SELECT u.name, o.id AS call_id
FROM   fire_station u
INNER JOIN fire_call o
    ON o.station_id = u.id;
```

INNER JOIN повертає лише частини, що мають хоча б один виклик. Частини без викликів повністю виключаються з результату — це найжорсткіший тип з'єднання.

```
-- LEFT: усі частини
SELECT u.name, o.id AS call_id
FROM   fire_station u
LEFT JOIN fire_call o
    ON o.station_id = u.id;
```

LEFT JOIN зберігає ВСІ частини. Для тих, що не мали викликів, o.id буде NULL. За цією ознакою (WHERE o.id IS NULL) можна знаходити частини без жодного виїзду.

Зверніть увагу. *Порівняйте кількість рядків у результатах обох запитів. Різниця дорівнює кількості частин, які не мали жодного виклику.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі joins.sql скласти запит із з'єднанням таблиць згідно теми варіанту.

№	Що реалізувати
1	Знайти курсантів без жодного складеного заліку через LEFT JOIN + WHERE o.id IS NULL.
2	Емулювати FULL OUTER JOIN у MySQL через UNION (LEFT + RIGHT) для таблиць rescuer та equipment.
3	CROSS JOIN між shift (зміна) та post (пост) для генерації всіх комбінацій графіка чергування.
4	SELF JOIN для пар рятувальник–начальник караулу з rescuer(id, chief_id).
5	З'єднати три таблиці (fire_station, fire_call, trip) через два INNER JOIN.

Питання 42. ORDER BY та GROUP BY

Теоретична частина

ORDER BY — сортування результату. Синтаксис: ORDER BY стовпець [ASC| DESC], ... За замовчуванням ASC. Можна за псевдонімом зі SELECT. NULL у MySQL сортуються першими при ASC, у PostgreSQL — останніми (регулюється NULLS FIRST/LAST) [27].

Сортування — O(n log n). За наявності індексу з потрібним порядком СУБД може уникнути окремої фази сортування.

GROUP BY — групування рядків за значеннями стовпців. Кожен унікальний набір — один рядок результату. Усі НЕ-агрегатні стовпці у SELECT мають бути в GROUP BY. З GROUP BY використовуються агрегатні функції: COUNT(*), COUNT(стовпець), SUM, AVG, MIN, MAX. Без GROUP BY охоплюють весь результат як одну групу.

Порівняння: SELECT category GROUP BY category дає унікальні категорії, але дозволяє додати агрегацію COUNT(*) — кількість товарів у кожній. Для фільтрації груп за агрегацією існує HAVING.

Приклад задачі

Умова. Згрупувати виклики за типом події (пожежа, ДТП, рятування на воді), обчислити кількість та середню площу, відсортувати за кількістю.

SQL-запит у файлі report.sql.

Лістинг 42.1. Реалізація з покроковим розбором

Файл `report.sql` GROUP BY з агрегацією та ORDER BY

```
SELECT   type,
         COUNT(*)   AS cnt,
         AVG(area)  AS avg_area
FROM     fire_call
GROUP BY type
ORDER BY cnt DESC;
```

GROUP BY type формує по одному рядку на кожен унікальний тип події. COUNT(*) рахує виклики в групі, AVG(area) — середню площу. ORDER BY cnt DESC сортує за кількістю; псевдонім cnt доступний, бо ORDER BY обчислюється після SELECT.

З досвіду викладачів. *Псевдонім cnt можна використати в ORDER BY, але не в WHERE. Це прямий наслідок логічного порядку виконання запиту.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі report.sql скласти запит з групуванням та сортуванням згідно теми варіанту.

№	Що реалізувати
1	Підрахувати кількість курсантів за навчальними групами через GROUP BY.
2	Обчислити сумарну кількість викликів за кожен місяць.
3	Знайти топ-3 райони за середнім часом прибуття підрозділів.
4	Відсортувати rescuer за прізвищем, потім за ім’ям за зростанням.
5	GROUP BY за двома стовпцями (station_id, type); кількість автомобілів кожного типу.

Питання 43. Що таке HAVING

Теоретична частина

HAVING — клауза фільтрації груп ПІСЛЯ GROUP BY. На відміну від WHERE (фільтр окремих рядків ДО групування), HAVING фільтрує сформовані групи на основі агрегатних значень. Правило: умова на сире значення поля — WHERE; умова на агрегацію (COUNT, SUM, AVG) — HAVING. Часто обидві клаузи в одному запиті: WHERE відсікає рядки до групування, HAVING — групи за агрегаціями [26; 31].

У WHERE НЕ можна використати агрегатну функцію — помилка. Причина у порядку виконання: WHERE спрацьовує до GROUP BY, агрегації ще не обчислено. HAVING працює після.

HAVING без GROUP BY — вся таблиця як одна група. Наприклад, SELECT COUNT(*) FROM user HAVING COUNT(*) > 100. На рівні плану WHERE застосовується раніше і може використати індекси. Тому WHERE-фільтр ефективніший — використовуйте його завжди, коли семантично можна.

Приклад задачі

Умова. Знайти пожежно-рятувальні частини, у яких більше 10 справних автомобілів, відсортувати за кількістю спадно.

SQL-запит у файлі having.sql.

Лістинг 43.1. Реалізація з покроковим розбором

Файл `having.sql` WHERE + GROUP BY + HAVING

```
SELECT   station_id,
         COUNT(*) AS cnt
FROM     fire_truck
WHERE    ready = 1
GROUP BY station_id
HAVING   COUNT(*) > 10
ORDER BY cnt DESC;
```

WHERE ready = 1 відсікає несправну техніку ДО групування — дешевша робота для СУБД. GROUP BY групує по частині. HAVING COUNT(*) > 10 залишає лише групи з понад 10 автомобілями. Порядок: FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY.

Зверніть увагу. *Перевірте себе: що станеться, якщо перенести умову ready = 1 з WHERE у HAVING? Запит не виконається — у HAVING можна посилатися лише на стовпці групування та агрегати.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі having.sql скласти запит з фільтрацією груп через HAVING згідно теми варіанту.

№	Що знайти
1	Навчальні групи, де середній бал перевищує 80.
2	Рятувальників, що здійснили понад 5 виїздів за місяць.
3	Місяці, у яких сумарна площа пожеж перевищила 10 000 м².
4	Райони, у яких більше однієї пожежно-рятувальної частини.
5	Спробувати написати WHERE COUNT(*) > 5; пояснити, чому помилка.

Питання 44. Синтаксис CREATE TABLE

Теоретична частина

CREATE TABLE — оголошення нової таблиці. Синтаксис: CREATE TABLE [IF NOT EXISTS] назва (стовпець тип [обмеження], ..., [табличні обмеження]). Типові типи: INT, BIGINT, TINYINT, SMALLINT; DECIMAL(p,s) (для грошей), FLOAT, DOUBLE; CHAR(n), VARCHAR(n), TEXT; DATE, TIME, DATETIME, TIMESTAMP; BOOLEAN, ENUM, JSON, BLOB [31; 32].

Обмеження стовпця: NOT NULL, DEFAULT значення, UNIQUE, PRIMARY KEY, AUTO_INCREMENT/SERIAL, CHECK(умова).

Табличні обмеження: PRIMARY KEY (col1, col2), FOREIGN KEY (col) REFERENCES other(id), UNIQUE (col1, col2), INDEX idx_name (col1, col2).

MySQL: ENGINE=InnoDB, CHARSET=utf8mb4. PostgreSQL не має AUTO_INCREMENT — використовує SERIAL або GENERATED ALWAYS AS IDENTITY.

Приклад задачі

Умова. Створити таблицю report (донесення про пожежу) з усіма основними елементами: PK з автоінкрементом, FK з каскадним видаленням, складеним індексом, DEFAULT-значенням.

SQL-скрипт у файлі create_report.sql.

Лістинг 44.1. Реалізація з покроковим розбором

Файл `create_report.sql` таблиця з PK, FK, індексом, DEFAULT

```
CREATE TABLE report (
    id      BIGINT PRIMARY KEY
                   AUTO_INCREMENT,
    call_id INT          NOT NULL,
    title   VARCHAR(200) NOT NULL,
    content TEXT,
```

PRIMARY KEY id типу BIGINT з AUTO_INCREMENT — сурогатний ключ. call_id NOT NULL — донесення завжди стосується виклику. title NOT NULL — заголовок обов'язковий. content TEXT — необмежений текст, може бути NULL.

```
    created DATETIME
        DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (call_id)
        REFERENCES fire_call(id)
        ON DELETE CASCADE,
    INDEX idx_call_created
        (call_id, created)
);
```

DEFAULT CURRENT_TIMESTAMP заповнює дату створення автоматично. FK на fire_call з ON DELETE CASCADE: при видаленні виклику видаляться його донесення. Складений індекс прискорить пошук донесень щодо виклику за датою.

Зверніть увагу. *Тип TEXT для поля content обрано свідомо: донесення про пожежу може бути довгим, а обмеження VARCHAR(200) його обрізало б.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі create.sql створити таблицю згідно теми варіанту з відповідними обмеженнями.

№	Що створити
1	Таблицю cadet з полями id, name, group_id (FK на study_group).
2	Таблицю duty_schedule з композитним PRIMARY KEY (rescuer_id, date).
3	Таблицю rescuer з CHECK-обмеженням age >= 18.
4	Таблицю unit з self-reference (parent_id як FK на власний id).
5	Таблицю equipment з DEFAULT-значеннями для полів created та updated.

Розділ 5. Технологія JDBC

Питання 45. Як вибрати всі рядки з таблиці та вивести у System.out через JDBC

Теоретична частина

JDBC (Java Database Connectivity) — стандартний API Java для взаємодії з реляційними БД. Базовий потік: 1) отримати Connection через DriverManager.getConnection; 2) створити PreparedStatement; 3) виконати executeQuery() і отримати ResultSet; 4) проітерувати через while(rs.next()) і витягти значення; 5) закрити всі ресурси у зворотному порядку [33].

Сучасний стиль — try-with-resources (Java 7+): усі ресурси оголошуються у заголовку try, і JVM автоматично викликає close() у правильному порядку. Без цього легко допустити витік ресурсів, що призводить до вичерпання пулу з'єднань.

URL з'єднання: jdbc:підпротокол://хост:порт/база?параметри. До JDBC 4.0 (Java 6) драйвер потрібно було завантажувати вручну через Class.forName("...Driver"); сучасні драйвери реєструються автоматично через ServiceLoader. PreparedStatement слід надавати перевагу над Statement: він захищає від SQL-ін'єкцій, дозволяє кешувати план виконання і працює швидше при повторних викликах. Звичайний Statement — лише для DDL без параметрів.

Доступ до значень: rs.getXxx("назва") або rs.getXxx(індекс). Індексація 1-based — часте джерело багів.

Приклад задачі

Умова. Прочитати всі рядки з таблиці cadet (id, name) бази даних ЛДУБЖД та вивести у консоль, використовуючи try-with-resources.

Реалізація у файлі CadetDao.java з обробкою SQLException.

Лістинг 45.1. Реалізація з покроковим розбором

Файл `CadetDao.java` вибірка всіх рядків через JDBC

```
String url = "jdbc:mysql://localhost:3306/ldubgd";

try (Connection c =
        DriverManager.getConnection(url, "user",
"pass");
     PreparedStatement ps =
        c.prepareStatement("SELECT id, name FROM
cadet");
     ResultSet rs = ps.executeQuery()) {
```

Три ресурси (Connection, PreparedStatement, ResultSet) оголошено у заголовку try-with-resources — вони будуть автоматично закриті у зворотному порядку. executeQuery() виконує SELECT і повертає ResultSet.

```
    while (rs.next()) {
        int    id   = rs.getInt("id");
        String name = rs.getString("name");
        System.out.println(id + " " + name);
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

rs.next() повертає false наприкінці, завершуючи цикл. Значення витягуються за іменем стовпця. SQLException — checked-виняток, ловимо у catch. Закриття ресурсів — автоматичне.

Порада. *Пароль до бази даних у прикладі вказано прямо в коді лише для наочності. У реальних проєктах облікові дані зберігають у змінних середовища або конфігураційних файлах поза репозиторієм.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі CadetDao.java реалізувати вибірку з БД через JDBC згідно теми варіанту.

№	Що реалізувати
1	Вибрати рядки з fire_call, де area > 100 (через ?-параметр).
2	Підрахувати кількість курсантів через SELECT COUNT(*) AS cnt.
3	Метод List<Cadet> findAll(Connection c), що повертає список об’єктів.
4	Перехопити SQLException та вивести getSQLState і getErrorCode.
5	Посторінкове отримання журналу викликів через LIMIT ? OFFSET ?.

Питання 46. Як вставити рядок у БД через JDBC

Теоретична частина

Для INSERT використовується PreparedStatement з параметрами (?) та метод executeUpdate() (повертає кількість змінених рядків). Параметри підставляються через setInt(індекс, значення), setString тощо. Індексація 1-based. При AUTO_INCREMENT для PK значення не вказується. Щоб отримати згенерований ID, при підготовці передають Statement.RETURN_GENERATED_KEYS і викликають ps.getGeneratedKeys() [3; 33].

Для масової вставки — batch: ps.addBatch() накопичує рядки, ps.executeBatch() виконує одним запитом. Драматично швидше за послідовні executeUpdate.

Транзакції: за замовчуванням autocommit. Для атомарного блоку: setAutoCommit(false), потім commit() або rollback() при помилці. Параметризовані запити повністю захищають від SQL-ін’єкцій — значення інтерпретується як значення, не як SQL.

Приклад задачі

Умова. Вставити одного курсанта та отримати згенерований ID; виконати batch-вставку записів журналу сигналізації.

Реалізація у файлі CadetDao.java.

Лістинг 46.1. Реалізація з покроковим розбором

Файл `CadetDao.java` INSERT з отриманням згенерованого ID

```
String sql =
    "INSERT INTO cadet(name, email) VALUES
(?, ?)";
try (PreparedStatement ps =
c.prepareStatement(sql,
         Statement.RETURN_GENERATED_KEYS)) {
    ps.setString(1, "Петренко");
    ps.setString(2, "petrenko@ldubgd.edu.ua");
    ps.executeUpdate();
```

PreparedStatement з ?-параметрами. RETURN_GENERATED_KEYS активує отримання згенерованого ID. setString(1,...) та setString(2,...) підставляють значення (індексація 1-based). executeUpdate() виконує вставку.

```
    try (ResultSet keys =
             ps.getGeneratedKeys()) {
        if (keys.next())
            System.out.println("id = "
                + keys.getLong(1));
    }
}
```

getGeneratedKeys() повертає ResultSet зі згенерованими ключами. keys.next() переходить до першого, keys.getLong(1) читає згенерований AUTO_INCREMENT id. Це стандартний спосіб дізнатися ID нового запису.

Зверніть увагу. *Якщо таблиця не має AUTO_INCREMENT-ключа, getGeneratedKeys() поверне порожній ResultSet. Це нормальна поведінка, а не помилка.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Dao.java реалізувати вставку через JDBC згідно теми варіанту.

№	Що реалізувати
1	Вставити рядок у equipment з параметрами name, quantity; обробити SQLException.
2	Метод save(FireCall call) : Long, що повертає згенерований ID виклику.
3	Вставити виклик з NULL у nullable-стовпець phone через setNull(індекс, тип).
4	Пакетна вставка 1000 показів датчиків через batch; порівняти час з послідовним.
5	INSERT виклику та виїзду в одній транзакції: setAutoCommit(false), commit/rollback.

Питання 47. Різниця Statement vs PreparedStatement

Теоретична частина

Statement приймає готовий SQL-рядок без параметрів. Кожен виклик — повна компіляція плану на СУБД. Значення зазвичай через конкатенацію — це створює два недоліки: вразливість до SQL-ін’єкцій та неможливість повторного використання плану. PreparedStatement — параметризований запит зі знаками питання (?). Працює у два етапи: компіляція запиту на СУБД, потім передача значень окремо. Захист від ін’єкцій: значення параметра ніколи не інтерпретується як SQL-код [28; 59].

Друга перевага PreparedStatement — кешування плану виконання. При повторному виклику з різними параметрами СУБД повторно використовує план. Особливо помітно в циклах і батчах.

CallableStatement — для виклику stored procedure: c.prepareCall("{call sp(?, ?)}"). Підтримує IN, OUT та INOUT-параметри.

Правило: у виробничому коді ЗАВЖДИ використовуйте PreparedStatement для запитів з динамічними значеннями. Сирий Statement — лише для DDL.

Приклад задачі

Умова. Продемонструвати на формі входу до системи електронного журналу університету вразливість Statement до SQL-ін'єкції та її відсутність у PreparedStatement.

Порівняння двох підходів у файлі AuthDao.java.

Лістинг 47.1. Реалізація з покроковим розбором

Файл AuthDao.java вразливий Statement

```
String userInput = "' OR '1'='1";

// ВРАЗЛИВО: SQL-ін'єкція
Statement s = c.createStatement();
ResultSet rs = s.executeQuery(
    "SELECT * FROM cadet WHERE login = '"
    + userInput + "'");
```

Конкатенація вмонтовує зловмисний фрагмент у SQL-текст. Виконається: SELECT * FROM cadet WHERE login = '' OR '1'='1' — умова завжди істинна, повернуться ВСІ курсанти. Це класична SQL-ін'єкція.

```
// БЕЗПЕЧНО: параметризація
PreparedStatement ps = c.prepareStatement(
    "SELECT * FROM cadet WHERE login = ?");
ps.setString(1, userInput);
ResultSet rs2 = ps.executeQuery();
```

PreparedStatement з ?-параметром. Драйвер JDBC екранує значення на рівні протоколу — userInput сприймається буквально як рядок-значення, не як SQL. Ін'єкція неможлива в принципі.

Зверніть увагу. *Корисний експеримент: виведіть у консоль SQL-рядок, зібраний конкатенацією, і подивіться на нього очима зловмисника. Вразливість стає очевидною.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Dao.java продемонструвати роботу зі Statement / PreparedStatement згідно теми варіанту.

№	Що реалізувати
1	Продемонструвати SQL-ін'єкцію в пошуку виклику за адресою через Statement та її блокування через PreparedStatement.
2	Виміряти час 1000 однакових SELECT викликів через Statement та PreparedStatement.
3	PreparedStatement з трьома ?-параметрами (район, тип, дата); викликати з різними значеннями.
4	CallableStatement для виклику stored procedure підрахунку виїздів з OUT-параметром.
5	Динамічна побудова WHERE з опційними фільтрами пошуку викликів через PreparedStatement.

Питання 48. Що таке ResultSet, як отримати, як з ним працювати

Теоретична частина

ResultSet — об'єкт-курсор, що зберігає результат SELECT. Отримується викликом executeQuery(). Концептуально це таблиця у пам'яті (дані можуть підвантажуватися порціями). Початково курсор стоїть ПЕРЕД першим рядком; rs.next() переміщує і повертає

true, поки рядки є. Доступ до значень: rs.getInt("col"), rs.getString("col"), rs.getTimestamp, rs.getObject. За іменем читабельніше, за індексом трохи швидше. Індексація 1-based [33].

Для nullable-стовпців нюанс: rs.getInt повертає 0 для NULL. Щоб розрізнити «0» від «NULL», після getInt викликається rs.wasNull(). Альтернатива — rs.getObject("col", Integer.class).

Три типи ResultSet: TYPE_FORWARD_ONLY (за замовчуванням, лише вперед); TYPE_SCROLL_INSENSITIVE (навігація absolute, relative, first, last); TYPE_SCROLL_SENSITIVE. Чим складніший — тим повільніше. Здебільшого FORWARD_ONLY достатньо. ResultSet тримає курсор на стороні СУБД. КРИТИЧНО закривати до закриття Statement і Connection. try-with-resources розв’язує автоматично.

Приклад задачі

Умова. Прочитати результат SELECT з таблиці rescuer, обробити nullable-стовпець стажу через wasNull.

Реалізація у файлі RescuerDao.java.

Лістинг 48.1. Реалізація з покроковим розбором

Файл `RescuerDao.java` обробка nullable-стовпця через wasNull

```
try (PreparedStatement ps = c.prepareStatement(
         "SELECT id, name, exp FROM rescuer");
     ResultSet rs = ps.executeQuery()) {

    while (rs.next()) {
        int    id   = rs.getInt("id");
        String name = rs.getString("name");
        int    exp  = rs.getInt("exp");
```

executeQuery() повертає ResultSet. rs.next() рухає курсор по рядках. Значення читаються за іменем стовпця. Поле exp (стаж, років) може бути NULL у БД — тут rs.getInt поверне 0, що неоднозначно.

```
        if (rs.wasNull()) {
            System.out.println(id + " "
                + name + " exp=null");
        } else {
            System.out.println(id + " "
                + name + " exp=" + exp);
        }
    }
}
```

rs.wasNull() повертає true, якщо останнє getInt прочитало SQL NULL. Без цієї перевірки NULL виглядав би як 0 у Java — некоректно. Так розрізняємо «стаж 0 років» (новачок) і «стаж невідомий».

З досвіду викладачів. *Метод getObject("exp", Integer.class) повертає null для порожнього стажу без додаткового виклику wasNull(). Такий запис часто зручніший.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Dao.java реалізувати роботу з ResultSet згідно теми варіанту.

№	Що реалізувати
1	Прочитати ResultSet викликів з п’ятьма стовпцями різних типів і вивести кожен.
2	Використати rs.getObject(col) для невідомого наперед типу.
3	Створити scrollable ResultSet курсантів і повернутися на rs.first().
4	Прочитати ResultSet спорядження у List<Map<String, Object>>.
5	Спробувати rs.getInt() без попереднього rs.next(); пояснити поведінку.

Питання 49. Виклик stored procedure через JDBC. Закриття ресурсів. Connection у кількох потоках

Теоретична частина

Stored procedure — підпрограма, що зберігається й виконується на сервері БД. Виклик через CallableStatement: c.prepareCall("{call sp(?, ?)}"). Параметри: IN (setXxx), OUT (registerOutParameter перед виконанням, getXxx після), INOUT. Послідовність: підготувати виклик; задати IN-параметри; зареєструвати OUT-параметри (Types.INTEGER тощо); викликати execute(); прочитати OUT через getInt(індекс) [31].

Закриття ресурсів — критичне. Порядок: ResultSet → Statement → Connection. Використання ResultSet після закриття Statement дасть SQLException. try-with-resources закриває у зворотному порядку відкриття.

Connection НЕ потокобезпечний. JDBC гарантує лише послідовне використання у межах одного потоку. Передача одного Connection між потоками призведе до плутанини транзакцій і ResultSet-ів.

Правильний шлях — connection pool (HikariCP, DBCP). Кожен потік запитує з'єднання через DataSource.getConnection(), повертає через close(). У пулі close() не закриває фізичне з'єднання, а повертає його у пул.

Приклад задачі

Умова. Викликати stored procedure get_call_count з одним OUT-параметром та отримати загальну кількість викликів за добу.

Реалізація у файлі CallStatsDao.java.

Лістинг 49.1. Реалізація з покроковим розбором

Файл `CallStatsDao.java` виклик stored procedure з OUT-параметром

```
// На боці БД:
// CREATE PROCEDURE get_call_count(OUT total INT)

try (CallableStatement cs =
         c.prepareCall("{call
get_call_count(?)}")) {
    cs.registerOutParameter(1, Types.INTEGER);
```

CallableStatement готується через prepareCall з синтаксисом {call ...}. registerOutParameter(1, Types.INTEGER) ОБОВ'ЯЗКОВО викликається ДО виконання — повідомляє драйверу про тип і позицію OUT-параметра.

```
    cs.execute();
    int total = cs.getInt(1);
    System.out.println("Викликів: " + total);
}
```

execute() виконує процедуру. Після цього значення OUT-параметра доступне через getInt(1). try-with-resources автоматично закриє CallableStatement. У реальному коді Connection береться з пулу.

Зверніть увагу. *На лабораторних заняттях процедуру get_call_count спочатку створюють у MySQL Workbench, а вже потім викликають із Java — так легше локалізувати помилку.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі Dao.java реалізувати роботу зі stored procedure / пулом з'єднань згідно теми варіанту.

№	Що реалізувати
1	Stored procedure get_cadet_by_id(IN id, OUT name) та виклик через CallableStatement.
2	CallableStatement з INOUT-параметром (залишок води в цистерні).
3	Спробувати закрити Connection до закриття ResultSet; пояснити проблему.
4	Простий connection pool для системи чергової частини на основі BlockingQueue з 5 з'єднаннями.
5	Використати HikariCP DataSource та порівняти з прямим DriverManager.

Питання 50. Чи можна виконати CREATE TABLE через JDBC

Теоретична частина

Так, через JDBC можна виконувати будь-яку SQL-команду, включно з DDL: CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE INDEX, CREATE VIEW. Використовується execute() (повертає boolean) або executeUpdate() (для DDL зазвичай 0) [32].

Параметри (?) у DDL НЕ підтримуються — структурні елементи (імена таблиць, стовпців) не параметризуються. Якщо ім'я приходить ззовні — обов'язково валідуйте його (білий список) перед конкатенацією, інакше SQL-ін'єкція.

У більшості СУБД DDL виконується поза транзакцією — неявно фіксується відразу (implicit commit). В Oracle і MySQL DDL не може бути відкочений; у PostgreSQL деякі DDL транзакційні.

Для управління схемами рідко використовують прямий JDBC — беруть Flyway або Liquibase (versioned-міграції). Але під капотом вони використовують той самий JDBC. Тому розуміння, як виконати CREATE TABLE через JDBC, залишається фундаментальним.

Приклад задачі

Умова. Виконати CREATE TABLE журналу сигналізації через JDBC, потім ALTER TABLE для додавання стовпця.

Реалізація у файлі SchemaInit.java.

Лістинг 50.1. Реалізація з покроковим розбором

Файл `SchemaInit.java` виконання DDL через Statement

```
try (Statement s = c.createStatement()) {
    s.execute(
        "CREATE TABLE IF NOT EXISTS alarm_log ("
        + "id  INT PRIMARY KEY AUTO_INCREMENT, "
        + "msg VARCHAR(500) NOT NULL)");
```

Для DDL достатньо звичайного Statement (параметрів немає). Метод execute() виконує будь-який SQL. IF NOT EXISTS захищає від помилки при повторному запуску скрипта міграції.

```
    s.execute(
        "ALTER TABLE alarm_log"
        + " ADD COLUMN level VARCHAR(10)");
}
```

ALTER TABLE додає новий стовпець level. У MySQL ця DDL-команда неявно фіксується одразу (implicit commit) і не може бути відкочена — це треба враховувати при міграціях.

Зверніть увагу. *Пам'ятайте, що в MySQL кожна DDL-команда фіксується одразу. Помилка в середині скрипта міграції залишить схему у «напівзміненому» стані.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу у файлі SchemaInit.java виконати DDL-операцію через JDBC згідно теми варіанту.

№	Що реалізувати
1	CREATE TABLE fire_call з PRIMARY KEY, FOREIGN KEY та індексом через JDBC.
2	DROP TABLE IF EXISTS перед CREATE для чистої міграції бази даних курсантів.
3	ALTER TABLE ADD COLUMN для додавання стовпця district у таблицю fire_call.
4	CREATE VIEW active_calls через JDBC і виконати SELECT з нього.
5	TRUNCATE TABLE alarm_log через JDBC; пояснити різницю з DELETE.

ЧАСТИНА 3. ВЕБ-РОЗРОБКА

Розділ 6. Об’єктно-реляційне відображення Hibernate / JPA

Питання 51. JPA vs Hibernate

Теоретична частина

JPA (Jakarta Persistence API, раніше Java Persistence API) — це специфікація (стандарт) об’єктно-реляційного відображення (ORM) у Java. Сама по собі JPA — лише набір інтерфейсів та анотацій (EntityManager, @Entity, @Id, JPQL), вона не виконує жодного SQL. Для роботи потрібна конкретна реалізація. Hibernate — найпопулярніша реалізація JPA. Він існував ще до появи JPA і значною мірою вплинув на її дизайн. Hibernate має два API: JPA-сумісний (через EntityManager) та власний, ширший (через Session), що пропонує додаткові можливості поза стандартом [34; 37].

Інші реалізації JPA: EclipseLink (еталонна реалізація), Apache OpenJPA. Якщо код написаний лише через JPA-API, його можна перенести між реалізаціями майже без змін — у цьому головна перевага стандарту.

Коли обирати JPA-API: потрібна переносимість, можливість змінити реалізацію. Коли Hibernate-API: потрібні специфічні можливості (розширений HQL, фільтри, певні стратегії кешування), а переносимість не критична. На практиці у Spring-проєктах зазвичай використовують Spring Data JPA — ще вищий рівень абстракції над JPA, що генерує реалізації репозиторіїв автоматично.

Приклад задачі

Умова. Знайти курсанта університету за ідентифікатором двома способами: через стандартний JPA EntityManager та через Hibernate Session.

Сутність Cadet у файлі Cadet.java; два способи пошуку — у файлі CadetService.java.

Лістинг 51.1. Реалізація з покроковим розбором

Файл `Cadet.java` JPA-сутність

```
import jakarta.persistence.*;

@Entity
@Table(name = "cadets")
public class Cadet {
    @Id
    private Long id;
    private String name;
    // геттери / сеттери
}
```

Анотація @Entity позначає клас як JPA-сутність, що відображається на таблицю. @Table задає ім’я таблиці. @Id позначає первинний ключ. Ці анотації — частина стандарту JPA і працюють на будь-якій реалізації.

Файл `CadetService.java` JPA-API та Hibernate-API

```
// Стандартний JPA
EntityManager em = emf.createEntityManager();
Cadet u1 = em.find(Cadet.class, 1L);

// Власний API Hibernate
Session session = sessionFactory.openSession();
Cadet u2 = session.get(Cadet.class, 1L);
```

em.find(...) — стандартний метод JPA, працює на будь-якій реалізації. session.get(...) — метод Hibernate-API з тією ж семантикою. Обидва завантажують сутність за первинним ключем; перший варіант переносимий.

Порада. *У прикладі EntityManager і Session не закриваються — це спрощення. У робочому коді обидва об’єкти обов’язково закривають після використання.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити JPA-сутність у окремому файлі та реалізувати операцію згідно теми варіанту обома способами (JPA EntityManager і Hibernate Session), де це доречно.

№	Сутність	Що реалізувати
1	Equipment	знайти спорядження за id через EntityManager.find та Session.get
2	Rescuer	зберегти нового рятувальника через persist та save
3	FireCall	оновити статус виклику через merge та update
4	FireTruck	видалити списаний автомобіль через remove та delete
5	Lecturer	налаштувати persistence.xml та знайти викладача за id

Питання 52. Вибірка всіх рядків через JPA/Hibernate

Теоретична частина

Для вибірки всіх сутностей через JPA використовується мова запитів JPQL (Jakarta Persistence Query Language): em.createQuery("SELECT u FROM User u", User.class).getResultList(). Важливо: JPQL оперує СУТНОСТЯМИ та їхніми полями, а не таблицями та стовпцями. Запитуємо User (ім’я класу), не users (ім’я таблиці). Через Hibernate-API: session.createQuery("FROM User", User.class).list(). HQL (Hibernate Query Language) — діалект JPQL з додатковими можливостями [35].

Зі Spring Data JPA найпростіше: інтерфейс UserRepository extends JpaRepository<User, Long> вже має готовий метод findAll() — реалізацію згенерує Spring автоматично.

Обережно з лінивою ініціалізацією: якщо в User є lazy-колекція @OneToMany, доступ до неї поза відкритою сесією/транзакцією призведе до LazyInitializationException.

Для великих таблиць варто використовувати пагінацію (setFirstResult/setMaxResults) замість завантаження всіх рядків одразу — інакше ризик вичерпання пам’яті.

Приклад задачі

Умова. Вибрати всіх рятувальників частини з бази даних через JPQL і вивести їхні прізвища.

Сутність Rescuer у файлі Rescuer.java; вибірка — у файлі RescuerRepository.java.

Лістинг 52.1. Реалізація з покроковим розбором

Файл `Rescuer.java` проста сутність

```
@Entity
public class Rescuer {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    // геттери / сеттери
}
```

Сутність Rescuer з первинним ключем id. @GeneratedValue вмикає автоматичну генерацію значення ключа (аналог AUTO_INCREMENT). Поле name відображається на однойменний стовпець.

Файл `RescuerRepository.java` вибірка через JPQL

```
List<Rescuer> users = em.createQuery(
        "SELECT u FROM Rescuer u", Rescuer.class)
    .getResultList();

for (Rescuer u : users) {
    System.out.println(u.getName());
}
```

JPQL-запит «SELECT u FROM Rescuer u» запитує саме сутність Rescuer (не таблицю). getResultList() повертає List<Rescuer>. Hibernate сам генерує відповідний SQL (SELECT ... FROM rescuer). Цикл виводить прізвища всіх рятувальників.

Зверніть увагу. *Для таблиці з тисячами записів getResultList() завантажить усе в пам'ять. Для журналу викликів обов'язково використовуйте пагінацію.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити JPA-сутність та реалізувати вибірку через JPQL згідно теми варіанту.

№	Сутність	Що вибрати
1	Equipment	усе спорядження; вивести назви
2	FireTruck	лише моделі через SELECT t.model
3	Cadet	курсантів, допущених до сесії, через WHERE c.admitted = true
4	FireCall	усі виклики, відсортовані за датою (ORDER BY)
5	Report	донесення з пагінацією через setFirstResult / setMaxResults

Питання 53. JPA-еквівалент SessionFactory

Теоретична частина

У Hibernate-API центральним об'єктом є SessionFactory — важкий, потокобезпечний об'єкт, що створюється один раз на весь додаток і слугує фабрикою для створення Session. JPA-еквівалентом SessionFactory є EntityManagerFactory (EMF). Це теж важкий потокобезпечний об'єкт, зазвичай один на додаток (точніше — один на persistence unit). Створюється: Persistence.createEntityManagerFactory("назва-unit") [36].

З EntityManagerFactory отримують EntityManager — легкий, НЕ потокобезпечний об'єкт, що відповідає одній одиниці роботи (зазвичай один на запит або транзакцію).

Аналогія повна: SessionFactory ↔ EntityManagerFactory (важкі, один на додаток), Session ↔ EntityManager (легкі, короткоживучі). У Hibernate EntityManagerFactory фактично обгортає SessionFactory.

У Spring EntityManagerFactory зазвичай конфігурується контейнером (LocalContainerEntityManagerFactoryBean) і впроваджується автоматично — розробник рідко створює його вручну.

Приклад задачі

Умова. Створити EntityManagerFactory для інформаційної системи чергової частини, отримати з нього EntityManager, виконати роботу та коректно закрити обидва.

Конфігурація у файлі persistence.xml; код роботи — у файлі App.java.

Лістинг 53.1. Реалізація з покроковим розбором

Файл `persistence.xml` конфігурація persistence unit

```
<persistence-unit name="dispatch">
    <class>ua.ldubgd.FireCall</class>
    <properties>
        <property
            name="jakarta.persistence.jdbc.url"
            value="jdbc:h2:mem:dispatch"/>
    </properties>
</persistence-unit>
```

Файл persistence.xml описує persistence unit з ім'ям «dispatch». Тут перелічуються сутності та властивості з'єднання з БД. Саме це ім'я передається у createEntityManagerFactory.

Файл `App.java` життєвий цикл EMF та EntityManager

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("dispatc
h");
EntityManager em = emf.createEntityManager();

// ... робота з em ...

em.close();
emf.close();
```

EntityManagerFactory створюється один раз (важкий об'єкт). З нього отримуємо легкий EntityManager. Після роботи закриваємо em (короткоживучий), а emf — лише при завершенні додатку. Порядок закриття: спочатку em, потім emf.

Зверніть увагу. *EntityManagerFactory створюють один раз на весь застосунок. Створення її в кожному методі — поширена помилка, що різко уповільнює програму.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу налаштувати EntityManagerFactory та продемонструвати роботу з EntityManager згідно теми варіанту.

№	Що реалізувати
1	Створити persistence.xml з persistence-unit "ldubgd" для бази даних курсантів та отримати EMF.
2	Створити EMF, отримати кілька EntityManager (для кожного диспетчера), закрити кожен окремо.
3	Налаштувати EMF системи обліку спорядження програмно через Map властивостей.
4	Зробити EMF singleton-ом у власному утилітному класі JpaUtil.
5	Реалізувати graceful shutdown системи чергової частини: закрити всі EM, потім EMF.

Питання 54. Що таке Session

Теоретична частина

Session (у Hibernate) — короткоживучий об'єкт, що представляє одиницю роботи (unit of work) з базою даних. Це головний інтерфейс для збереження, завантаження, оновлення та видалення сутностей у Hibernate-API [34].

Session тримає першорівневий кеш (Persistence Context): усі сутності, з якими працюємо в межах цієї сесії, кешуються в ньому. Завдяки цьому діє гарантія ідентичності: для одного первинного ключа в межах однієї Session повертається той самий Java-об'єкт. session.get(User.class, 1L) двічі поверне той самий екземпляр.

Життєвий цикл сутності в Session: transient (новий об'єкт, не пов'язаний з БД), persistent (керований сесією, синхронізується з БД), detached (був керований, але сесія

закрита), removed (помічений на видалення). Session відстежує зміни керованих сутностей (dirty checking): якщо змінити поле persistent-об’єкта, Hibernate автоматично згенерує UPDATE при commit, без явного виклику update().

Session не потокобезпечний — кожен потік повинен мати власну сесію. Закривається через close().

Приклад задачі

Умова. Завантажити сутність FireCall через Session, змінити статус виклику та переконатися, що Hibernate автоматично згенерує UPDATE завдяки dirty checking.

Робота з Session у файлі CallService.java у межах транзакції.

Лістинг 54.1. Реалізація з покроковим розбором

Файл `CallService.java` dirty checking у межах сесії	

`Session session = factory.openSession();`
`Transaction tx = session.beginTransaction();`
 	
`FireCall u = session.get(FireCall.class, 1L);`
`u.setStatus("ЛОКАЛІЗОВАНО");`
	Відкриваємо сесію та транзакцію. session.get завантажує FireCall за ключем — тепер це persistent-об’єкт, керований сесією. Змінюємо поле status. Явного виклику update() не потрібно.

`tx.commit();` `session.close();`	При commit Hibernate виконує dirty checking — порівнює поточний стан керованих об’єктів з початковим. Виявивши зміну status, автоматично генерує UPDATE. Це ключова перевага керованих сутностей.

З досвіду викладачів. *Додайте до persistence.xml властивість hibernate.show_sql = true — і ви побачите той самий UPDATE, який Hibernate згенерував автоматично.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу з Hibernate Session згідно теми варіанту у файлі сервісу.

№	Що реалізувати
1	Завантажити Cadet, змінити навчальну групу; переконатися, що commit генерує UPDATE.
2	Витягнути FireTruck двічі в одній Session; переконатися, що це той самий об’єкт.
3	Зберегти новий виклик через save(); вивести згенерований id.
4	Видалити списане спорядження через delete().
5	Витягнути Rescuer, закрити сесію; продемонструвати роботу з detached-об’єктом.

Питання 55. JPA-еквівалент Session

Теоретична частина

JPA-еквівалентом Hibernate Session є EntityManager (EM). Семантика повністю аналогічна: він представляє одиницю роботи, тримає Persistence Context (першорівневий кеш), забезпечує ідентичність сутностей у своїх межах [37].

Основні методи EntityManager: persist(entity) — зробити transient-об’єкт керованим (аналог save); find(Class, id) — завантажити за ключем (аналог get); merge(entity) — приєднати detached-об’єкт; remove(entity) — видалити; flush() — примусово синхронізувати з БД; clear() — очистити Persistence Context; close() — закрити.

Відмінність від Session переважно термінологічна: JPA-API більш формалізований і стандартизований. У Hibernate EntityManager фактично делегує роботу внутрішній Session.

У Spring/Jakarta EE EntityManager отримують через анотацію @PersistenceContext — контейнер сам керує його життєвим циклом і прив’язкою до транзакції, тому розробник не створює і не закриває його вручну. Як і Session, EntityManager не потокобезпечний — один на одиницю роботи.

Приклад задачі

Умова. Зареєструвати нового курсанта та зберегти його через EntityManager.persist у межах транзакції.

Робота з EntityManager у файлі CadetService.java.

Лістинг 55.1. Реалізація з покроковим розбором

Файл `CadetService.java` збереження через persist	

`EntityManager em = emf.createEntityManager();`
`em.getTransaction().begin();`
 	
`Cadet u = new Cadet();`
`u.setName("Петренко");`
`em.persist(u);`
	Отримуємо EntityManager та починаємо транзакцію. Створюємо новий об’єкт (transient-стан). em.persist(u) робить його керованим (managed) — Hibernate заплановує INSERT. Це JPA-аналог Session.save.

`em.getTransaction().commit();` `em.close();`	commit фіксує транзакцію — виконується запланований INSERT. close() звільняє EntityManager та його Persistence Context. Порядок класичний: begin → операції → commit → close.

Зверніть увагу. *Якщо викликати persist() поза транзакцією, INSERT не виконається. Ця помилка часто трапляється в перших лабораторних роботах з JPA.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу з JPA EntityManager згідно теми варіанту.

№	Що реалізувати
1	Зареєструвати новий виклик через em.persist у межах транзакції.
2	Знайти рятувальника через em.find; пояснити результат для невідомого id.
3	Оновити detached-сутність Cadet через em.merge; пояснити відмінність від persist.
4	Видалити запис про списаний вогнегасник через em.remove.
5	Викликати em.flush() та em.clear() під час масового імпорту викликів; пояснити їх дію.

Питання 56. HQL/JPQL із прикладом WHERE. Типи асоціацій

Теоретична частина

JPQL (Jakarta Persistence Query Language) та HQL (Hibernate Query Language) — мови запитів, синтаксично схожі на SQL, але оперують СУТНОСТЯМИ та їхніми полями, а не таблицями та стовпцями. Псевдоніми обов'язкові. Параметри задаються через :name (named) або ?1 (positional). Приклад з WHERE: SELECT p FROM Report p WHERE p.author.name = :author. Тут p.author.name — навігація через асоціацію: Hibernate автоматично згенерує необхідний JOIN [35; 36].

Типи асоціацій між сутностями: @OneToOne (один до одного), @OneToMany (один до багатьох), @ManyToOne (багато до одного), @ManyToMany (багато до багатьох). Найчастіша — @ManyToOne (наприклад, багато донесень належать одному рятувальнику-автору).

Стратегія завантаження (fetch type): за замовчуванням EAGER для *-ToOne асоціацій та LAZY для *-ToMany. Найкраща практика — явно вказувати LAZY скрізь і використовувати JOIN FETCH у конкретних запитах, де дані потрібні.

@ManyToMany реалізується через проміжну таблицю; на практиці часто замінюють явною сутністю-зв'язком з двома @ManyToOne для більшого контролю.

Приклад задачі

Умова. Створити асоціацію @ManyToOne між Report (донесення про пожежу) та Rescuer (автор донесення), а потім вибрати всі донесення конкретного рятувальника через JPQL з параметром.

Сутність Report з асоціацією у файлі Report.java; запит — у файлі ReportRepository.java.

Лістинг 56.1. Реалізація з покроковим розбором

Файл `Report.java` асоціація @ManyToOne

```
@Entity
public class Report {
    @Id @GeneratedValue
    private Long id;
    private String title;

    @ManyToOne(fetch = FetchType.LAZY)
    private Rescuer author;
}
```

Анотація @ManyToOne означає: багато донесень можуть належати одному автору. fetch = LAZY відкладає завантаження author до моменту звернення. У БД це реалізується стовпцем-зовнішнім ключем author_id у таблиці донесень.

Файл `ReportRepository.java` JPQL з навігацією через асоціацію

```
List<Report> posts = em.createQuery(
    "SELECT p FROM Report p"
    + " WHERE p.author.name = :author",
    Report.class)
  .setParameter("author", "Коваль")
  .getResultList();
```

Вираз p.author.name — навігація через асоціацію @ManyToOne. Hibernate автоматично згенерує JOIN між таблицями донесень та рятувальників. Параметр :author підставляється через setParameter — захист від ін'єкцій.

Зверніть увагу. *Параметр :author у JPQL — це не просто зручність, а захист від ін'єкцій, так само як «?» у PreparedStatement.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити сутності з вказаною асоціацією у окремих файлах та реалізувати JPQL-запит з фільтром згідно теми варіанту.

№	Асоціація	Сутності / запит
1	@ManyToOne	Report → Rescuer; донесення автора за прізвищем
2	@OneToMany	FireStation → FireTrucks (mappedBy); автомобілі частини
3	@ManyToMany	Cadet ↔ Course; курсанти, записані на дисципліну
4	@OneToOne	Rescuer ↔ Certificate; посвідчення за прізвищем рятувальника
5	@ManyToOne	FireCall → FireStation з JOIN FETCH (уникнути N+1)

Питання 57. Lazy initialisation. Рівні кешування Hibernate

Теоретична частина

Lazy initialization (відкладене завантаження) — стратегія, за якої асоційовані дані не завантажуються одразу разом із власником, а лише при першому зверненні до них. Це економить пам’ять і запити, коли пов’язані дані не потрібні [35].

Класична проблема — LazyInitializationException: якщо звернутися до lazy-асоціації після того, як сесія (Persistence Context) вже закрита, Hibernate не може дозавантажити дані і кидає виняток. Розв’язання: JOIN FETCH у запиті, @EntityGraph, або тримати сесію відкритою довше (OpenSessionInView, хоча це антипатерн).

Hibernate має три рівні кешування. Перший рівень (first-level cache) — це Persistence Context самої Session/EntityManager. Увімкнений завжди, обов’язковий, діє в межах однієї сесії. Гарантує ідентичність об’єктів.

Другий рівень (second-level cache) — спільний між сесіями, на рівні SessionFactory. Опційний, потребує провайдера (Ehcache, Hazelcast, Infinispan). Вмикається анотацією @Cacheable. Доцільний лише для рідко змінюваних сутностей (довідники). Третій — кеш запитів (query cache) — кешує результати запитів за їх параметрами. Вмикається окремо, працює разом із second-level cache.

Приклад задачі

Умова. Відтворити LazyInitializationException при зверненні до lazy-колекції виїздів рятувальника після закриття сесії.

Демонстрація проблеми у файлі Demo.java.

Лістинг 57.1. Реалізація з покроковим розбором

Файл `Demo.java` доступ до lazy-колекції поза сесією	

`Rescuer u;`
`try (Session s = factory.openSession()) {`
`  u = s.get(Rescuer.class, 1L);`
`}`
`// сесія вже закрита!`
	Завантажуємо Rescuer у межах try-with-resources. Колекція u.getTrips() оголошена як LAZY — вона ще НЕ завантажена. Після виходу з блоку try сесія закривається разом із Persistence Context.

`// LazyInitializationException:` `u.getTrips().size();`	Спроба звернутися до lazy-колекції getTrips() поза відкритою сесією. Hibernate не може виконати запит для дозавантаження — кидає LazyInitializationException. Розв'язання: JOIN FETCH у запиті або @EntityGraph.

Порада. *Найпростіше виправлення — додати до запиту JOIN FETCH: SELECT r FROM Rescuer r JOIN FETCH r.trips WHERE r.id = :id.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу продемонструвати роботу з lazy-завантаженням або кешуванням Hibernate згідно теми варіанту.

№	Що реалізувати
1	Відтворити LazyInitializationException на lazy-колекції автомобілів FireStation.
2	Виправити попередній варіант через JOIN FETCH у JPQL.
3	Виправити через @EntityGraph.
4	Увімкнути second-level cache та позначити довідникову сутність FireClass @Cacheable.
5	Пояснити N+1 проблему на прикладі «частини → виклики» та усунути її через JOIN FETCH.

Питання 58. Напрямки асоціацій (unidirectional / bidirectional)

Теоретична частина

Асоціації між сутностями можуть бути односпрямованими (unidirectional) або двоспрямованими (bidirectional). Unidirectional: навігація можлива лише в один бік. Наприклад, Report (донесення) знає свого автора Rescuer (через @ManyToOne), але Rescuer не має посилання на свої донесення. Простіше у налаштуванні, менше мапінгів [34].

Bidirectional: обидві сторони знають одна одну. Rescuer має колекцію донесень (@OneToMany), а Report має посилання на Rescuer (@ManyToOne). У коді можна переходити між сутностями в обидва боки.

У bidirectional-зв'язку одна сторона є власником (owner) — саме вона тримає зовнішній ключ і визначає, як Hibernate генерує SQL для збереження зв'язку. Інша сторона — інверсна (inverse), позначається атрибутом mappedBy. У парі @OneToMany/@ManyToOne власником завжди є сторона @ManyToOne (там, де ЗК).

Поширена пастка: якщо у bidirectional-зв’язку додати елемент лише з одного боку, об’єкти в пам’яті будуть неузгоджені. Тому реалізують helper-метод (наприклад, addReport), що синхронізує обидві сторони одночасно.

Приклад задачі

Умова. Створити двоспрямовану асоціацію між Rescuer та Report (донесення) із helper-методом, що синхронізує обидві сторони.

Сутності Rescuer та Report у відповідних файлах.

Лістинг 58.1. Реалізація з покроковим розбором

Файл `Rescuer.java` інверсна сторона (mappedBy) + helper

```
@Entity
public class Rescuer {
    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "author")
    private Set<Report> posts = new HashSet<>();

    public void addReport(Report p) {
        posts.add(p);
        p.setAuthor(this);
    }
}
```

Rescuer — інверсна сторона: mappedBy = "author" вказує, що власником зв’язку є поле author у Report. Helper-метод addReport синхронізує обидві сторони: додає донесення в колекцію ТА встановлює автора в донесенні.

Файл `Report.java` сторона-власник (тримає ЗК)

```
@Entity
public class Report {
    @Id @GeneratedValue
    private Long id;
    private String title;

    @ManyToOne
    private Rescuer author;
    // геттер / сеттер author
}
```

Report — власник зв’язку: саме тут @ManyToOne і фізично зберігається зовнішній ключ author_id. Hibernate орієнтується на стан цього поля при генерації SQL для збереження зв’язку.

Зверніть увагу. *Спробуйте закоментувати рядок p.setAuthor(this) у helper-методі. Донесення потрапить у колекцію, але в базі даних author_id залишиться порожнім.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати асоціацію вказаного напрямку між сутностями та helper-метод синхронізації, де доречно.

№	Напрямок	Сутності
1	unidirectional @OneToMany	Department → Lecturers
2	bidirectional @OneToMany/@ManyToOne	FireStation ↔ FireTrucks з helper-методом
3	bidirectional @OneToOne	Rescuer ↔ MedicalCard
4	self-reference @OneToMany	Unit (управління ↔ загони ↔ частини)
5	unidirectional @ManyToOne	FireCall → FireStation

Розділ 7. Протокол HTTP

Питання 59. Як працює HTTP

Теоретична частина

HTTP (HyperText Transfer Protocol) — текстовий протокол прикладного рівня типу запит-відповідь (request-response), що працює поверх TCP (зазвичай порт 80 для HTTP та 443 для HTTPS). Клієнт встановлює TCP-з'єднання із сервером, надсилає запит, отримує відповідь та інтерпретує її. HTTP-запит містить: метод (GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS), шлях (URL), версію протоколу, заголовки та опційне тіло. Відповідь містить: статус-код (200 OK, 404 Not Found, 500 Internal Server Error), заголовки та тіло [44; 45; 55].

Ключова властивість HTTP — він stateless (без стану): сервер сам по собі не пам'ятає попередніх запитів від клієнта. Стан між запитами реалізують через cookies, серверні сесії або токени (JWT).

Еволюція: HTTP/1.0 — одне з'єднання на запит; HTTP/1.1 — постійні з'єднання (keep-alive), конвеєризація; HTTP/2 — мультиплексування багатьох запитів в одному з'єднанні, бінарний формат; HTTP/3 — поверх QUIC (UDP).

Методи мають семантику: GET (отримати, безпечний, ідемпотентний), POST (створити, не ідемпотентний), PUT (замінити, ідемпотентний), DELETE (видалити, ідемпотентний).

Приклад задачі

Умова. Розглянути структуру типового HTTP-запиту GET до API чергової частини та відповіді сервера у форматі JSON.

Текстове представлення HTTP-обміну (request.txt — запит, response.txt — відповідь).

Лістинг 59.1. Реалізація з покроковим розбором

Файл `request.txt` сирий HTTP-запит

```
GET /api/calls HTTP/1.1
Host: dispatch.ldubgd.edu.ua
Accept: application/json
User-Agent: curl/8.0

(тіло відсутнє для GET)
```

Перший рядок: метод GET, шлях /api/calls, версія HTTP/1.1. Далі заголовки: Host (обов'язковий, який віртуальний хост), Accept (бажаний формат відповіді), User-Agent (клієнт). Порожній рядок відділяє заголовки від тіла.

Файл `response.txt` сира HTTP-відповідь

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 30

[{"id":1,"type":"пожежа"}]
```

Перший рядок: версія, статус-код 200 та його опис OK. Заголовки: Content-Type (тип тіла), Content-Length (розмір у байтах). Після порожнього рядка йде тіло відповіді — JSON-масив викликів.

Зверніть увагу. *HTTP-обмін зручно спостерігати в браузері: відкрийте інструменти розробника (F12), вкладка Network, і перегляньте заголовки будь-якого запиту.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу описати або виконати HTTP-обмін згідно теми варіанту (можна через curl або текстовим представленням запиту/відповіді).

№	Що реалізувати
1	GET-запит до публічного API (наприклад, прогнозу погоди для оцінки пожежної небезпеки) через curl; розглянути заголовки відповіді.
2	POST-запит з JSON-тілом нового виклику через curl; пояснити Content-Type.
3	Перерахувати 5 статус-кодів різних класів (2xx, 3xx, 4xx, 5xx) та їх значення.
4	HEAD-запит до сайту університету та порівняти з GET; пояснити різницю.
5	Пояснити, чому HTTP stateless, та як cookies додають стан (на прикладі входу в електронний журнал).

Питання 60. Заголовки запиту та відповіді (навести по два)

Теоретична частина

HTTP-заголовки — це пари «ім’я: значення», що передають метаінформацію про запит чи відповідь. Вони йдуть після стартового рядка і відділяються від тіла порожнім рядком. Поширені заголовки запиту: Host (обов’язковий — який віртуальний хост), Accept (які формати приймає клієнт), Authorization (токен або basic-аутентифікація), Content-Type (тип тіла при POST/PUT), Content-Length, Cookie (надсилає збережені cookie серверу), User-Agent [44].

Поширені заголовки відповіді: Content-Type (тип повернутого вмісту), Content-Length, Set-Cookie (просить клієнт зберегти cookie), Cache-Control (інструкції кешування), Location (для редиректів зі статусом 3xx), Server, Date.

Заголовок Content-Type визначає формат тіла: application/json, text/html, application/x-www-form-urlencoded, multipart/form-data. Він критичний для правильної інтерпретації даних обома сторонами.

Кастомні заголовки історично починалися з префікса X- (наприклад, X-Request-ID); RFC 6648 визнав цю практику застарілою, але вона досі поширена.

Приклад задачі

Умова. Розглянути HTTP-обмін мобільного застосунку рятувальника з сервером чергової частини: типові заголовки запиту (Authorization, Accept) та відповіді (Content-Type, Cache-Control).

Текстове представлення обміну з заголовками.

Лістинг 60.1. Реалізація з покроковим розбором

Файл `request.txt` заголовки запиту

```
GET /api/v1/incidents HTTP/1.1
Host: dispatch.ldubgd.edu.ua
Authorization: Bearer eyJhbGci...
Accept: application/json
```

Authorization несе токен доступу (схема Bearer) — сервер за ним аутентифікує рятувальника. Accept повідомляє, що клієнт очікує JSON. Host вказує цільовий хост — обов’язковий у HTTP/1.1.

Файл `response.txt` заголовки відповіді	

`HTTP/1.1 200 OK`
`Content-Type: application/json;`
`  charset=utf-8`
`Cache-Control: no-cache`
`X-Request-ID: a1b2c3`
	Content-Type повідомляє формат і кодування тіла. Cache-Control: no-cache забороняє кешувати відповідь — дані про пожежі мають бути завжди актуальними. X-Request-ID — кастомний заголовок для трасування запиту в логах сервера.

З досвіду викладачів. *Токен у заголовку Authorization — це фактично пароль. Ніколи не публікуйте його у звітах, скріншотах чи на GitHub.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу продемонструвати або описати використання HTTP-заголовків згідно теми варіанту.

№	Що реалізувати
1	Запит до електронного журналу з Authorization: Basic; пояснити кодування base64.
2	Пояснити Content-Type: application/json vs x-www-form-urlencoded на прикладі форми реєстрації виклику.
3	Редирект зі сторінки входу до кабінету курсанта через заголовок Location зі статусом 302.
4	Заголовки кешування Cache-Control: max-age=3600 для довідника пожежних гідрантів; пояснити дію.
5	CORS-заголовок Access-Control-Allow-Origin для карти викликів; пояснити призначення.

Розділ 8. Технологія сервлетів

Питання 61. Що таке Servlet, методи інтерфейсу, мапінг до URL

Теоретична частина

Servlet — серверний Java-компонент, що обробляє HTTP-запити в межах web-контейнера (Tomcat, Jetty, GlassFish, WildFly). Базовий інтерфейс jakarta.servlet.Servlet має п’ять методів: init(ServletConfig) — ініціалізація, service(req, resp) — обробка запиту, destroy() — звільнення ресурсів, getServletConfig(), getServletInfo() [38].

На практиці успадковують абстрактний клас HttpServlet, який реалізує Servlet та додає методи для HTTP: doGet, doPost, doPut, doDelete, doHead, doOptions. Розробник перевизначає лише потрібні.

Контейнер створює зазвичай ОДИН екземпляр сервлета на додаток і викликає його методи з різних потоків — тому сервлети мають бути потокобезпечними (не зберігати стан запиту в полях). Мапінг до URL — три способи: 1) анотація @WebServlet("/path") (Servlet 3.0+); 2) дескриптор web.xml (елементи <servlet> та <servlet-mapping>); 3) програмно через ServletContext.addServlet.

Життєвий цикл: контейнер створює екземпляр → init() (один раз) → service() (на кожен запит) → destroy() (при вивантаженні).

Приклад задачі

Умова. Створити простий Servlet для інформаційної системи чергової частини, що відповідає на GET-запит текстом, та замапити його на URL /status.

Servlet у файлі StatusServlet.java з анотацією @WebServlet.

Лістинг 61.1. Реалізація з покроковим розбором

Файл `StatusServlet.java` Servlet з мапінгом через анотацію

```
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.*;
import java.io.IOException;

@WebServlet("/status")
public class StatusServlet
        extends HttpServlet {
```

Клас успадковує HttpServlet. Анотація @WebServlet("/status") мапить його на URL /status — контейнер сам зареєструє сервлет без web.xml. Це найзручніший спосіб з Servlet 3.0+.

```
    protected void doGet(
            HttpServletRequest req,
            HttpServletResponse resp)
            throws IOException {
        resp.setContentType("text/plain");
        resp.getWriter().write("Карaул на
зв'язку");
    }
}
```

Перевизначаємо doGet для обробки GET-запитів. setContentType задає тип відповіді. getWriter().write(...) записує тіло. На запит GET /status контейнер викличе саме цей метод.

Зверніть увагу. *Сервлет зручно перевіряти командою curl http://localhost:8080/status або просто у браузері — для GET-запиту цього досить.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу створити Servlet згідно теми варіанту у окремому файлі та замапити його на URL.

№	Що реалізувати
1	Servlet, що відповідає на GET /time поточним часом чергування.
2	Servlet, що приймає POST з параметрами виклику (адреса, тип) і повертає JSON.
3	Servlet, замаплений через web.xml замість анотації.
4	Servlet з ініціалізацією довідника пожежно-рятувальних частин через init().
5	Servlet, що читає параметр з URL (?name=...) і вітає курсанта.

Питання 62. Сигнатура методу Servlet.service

Теоретична частина

Базовий метод service інтерфейсу Servlet має сигнатуру: void service(ServletRequest req, ServletResponse resp) throws ServletException, IOException. Він приймає узагальнені (протоколо-незалежні) інтерфейси ServletRequest та ServletResponse. Метод оголошує два checked-винятки: ServletException (загальна помилка обробки) та IOException (помилка введення-виведення при роботі з потоками запиту/відповіді) [38; 39].

Клас HttpServlet перевизначає service з HTTP-специфічними типами: protected void service(HttpServletRequest req, HttpServletResponse resp). Ця реалізація аналізує HTTP-метод запиту (getMethod) і делегує виклик до відповідного doGet, doPost, doPut тощо.

У переважній більшості випадків розробник НЕ перевизначає service, а лише потрібні doXxx-методи. Перевизначення service у HttpServlet — рідкісний випадок, коли треба обробити всі методи однаково або реалізувати власну диспетчеризацію.

Метод нічого не повертає (void) — результат пишеться у переданий об'єкт HttpServletResponse через його потік виводу.

Приклад задачі

Умова. Створити Servlet диспетчерської системи з власною реалізацією service, що самостійно диспетчеризує запит за HTTP-методом.

Servlet у файлі RouterServlet.java з перевизначеним service.

Лістинг 62.1. Реалізація з покроковим розбором

Файл `RouterServlet.java` власна реалізація service

`@WebServlet("/router")`
`public class RouterServlet`
`  extends HttpServlet {`
 	
`  @Override`
`  protected void service(`
`  HttpServletRequest req,`
`  HttpServletResponse resp)`
`  throws ServletException, IOException {`
	Перевизначаємо service у HttpServlet з HTTP-типами. Сигнатура оголошує ServletException та IOException — це частина контракту, обидва checked. Зазвичай так НЕ роблять, а перевизначають doGet/doPost.

`  String method = req.getMethod();`
`  resp.getWriter().write("Method: "`
`  + method);`
`  }`
`}`
	Замість делегування до doXxx ми самі читаємо HTTP-метод через getMethod() і обробляємо запит вручну. Результат пишемо у response (метод нічого не повертає). Це власна диспетчеризація.

Зверніть увагу. *Перевизначення service() — рідкість у реальних проєктах. Тут воно показано, щоб ви розуміли механізм диспетчеризації, на якому працює HttpServlet.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати обробку запиту через service або doXxx згідно теми варіанту.

№	Що реалізувати
1	Servlet з власним service для ресурсу /incidents, що диспетчеризує за HTTP-методом.
2	Пояснити, чому service оголошує ServletException та IOException.
3	Servlet, що обробляє і GET, і POST форми реєстрації курсанта однаково через спільний метод.
4	Перевірити, що при відсутності doPost контейнер повертає 405.
5	Базовий клас з логуванням часу обробки запитів диспетчера у перевизначеному service.

Питання 63. Ієрархія Servlet

Теоретична частина

Ієрархія сервлетів складається з трьох рівнів. На вершині — інтерфейс Servlet (jakarta.servlet.Servlet), що визначає базовий контракт: init, service, destroy, getServletConfig, getServletInfo [39].

Наступний рівень — абстрактний клас GenericServlet, що реалізує інтерфейс Servlet протоколо-незалежним чином. Він надає реалізації init, destroy, getServletConfig, getServletInfo, залишаючи абстрактним лише service. Підходить для не-HTTP протоколів (рідкісний випадок).

Третій рівень — клас HttpServlet, що успадковує GenericServlet та додає HTTP-специфіку. Він перевизначає service для розбору HTTP-методу та делегування до doGet, doPost, doPut, doDelete, doHead, doOptions, doTrace.

Таким чином ланцюг: Servlet (інтерфейс) ← GenericServlet (абстрактний) ← HttpServlet ← ваш клас. Більшість реальних сервлетів успадковують саме HttpServlet. Завдяки GenericServlet ваш клас безкоштовно отримує реалізації init/destroy, а завдяки HttpServlet — зручні doXxx-методи замість ручного розбору HTTP-методу.

Приклад задачі

Умова. Створити власний сервлет системи моніторингу датчиків пожежної сигналізації, що наслідує HttpServlet, з логуванням у методах життєвого циклу init і destroy.

Servlet у файлі LifecycleServlet.java з методами життєвого циклу.

Лістинг 63.1. Реалізація з покроковим розбором

Файл `LifecycleServlet.java` наслідування HttpServlet та життєвий цикл

```
@WebServlet("/lifecycle")
public class LifecycleServlet
        extends HttpServlet {

    @Override
    public void init() {
        System.out.println("init: датчики
підключено");
    }
```

Клас успадковує HttpServlet (а той — GenericServlet, а той — Servlet). Метод init() викликається контейнером ОДИН раз при створенні екземпляра — успадкований від GenericServlet. Тут зручно ініціалізувати ресурси.

```
    @Override
    protected void doGet(
            HttpServletRequest req,
            HttpServletResponse resp)
            throws IOException {
        resp.getWriter().write("Датчики в нормі");
    }

    @Override
    public void destroy() {
        System.out.println("destroy: датчики
відключено");
    }
}
```

doGet — з рівня HttpServlet. destroy() викликається один раз при вивантаженні сервлета — успадкований від GenericServlet. Так клас використовує методи з усіх трьох рівнів ієрархії.

Порада. *Перегляньте журнал Tomcat після зупинки сервера: повідомлення з destroy() з'явиться саме там, а не у вікні браузера.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати сервлет із використанням методів ієрархії згідно теми варіанту.

№	Що реалізувати
1	Сервлет, що наслідує GenericServlet безпосередньо (не HttpServlet).
2	Перерахувати методи кожного рівня ієрархії у коментарях.
3	Сервлет журналу чергування з логуванням у init, doGet та destroy.
4	Сервлет, що при OPTIONS відповідає списком підтримуваних методів.
5	Перевірити, що doGet за замовчуванням повертає 405 Method Not Allowed.

Питання 64. ServletRequest та HttpServletRequest. Яку інформацію з них отримати

Теоретична частина

ServletRequest — базовий протоколо-незалежний інтерфейс із методами, спільними для будь-якого протоколу: getParameter(name) — параметр запиту, getAttribute/setAttribute — атрибути request-scope, getInputStream/getReader — тіло запиту, getContentType, getContentLength, getRemoteAddr — IP клієнта [38].

HttpServletRequest успадковує ServletRequest і додає HTTP-специфічні методи: getMethod() — HTTP-метод (GET/POST), getRequestURI(), getQueryString(), getHeader(name) — HTTP-заголовок, getHeaders(name), getCookies(), getSession(), getContextPath(), getPathInfo().

Концептуально важливо розрізняти три речі. Параметр (parameter) — дані з рядка запиту (?key=value) або з тіла форми. Заголовок (header) — HTTP-метаінформація. Атрибут (attribute) — внутрішнє сховище request-scope для передачі даних між фільтрами та сервлетами у межах одного запиту.

getParameter повертає String (або null), getParameterValues — масив (для кількох значень одного імені). Атрибути зберігають будь-який Object. Атрибути живуть лише в межах одного запиту і не передаються клієнту — на відміну від параметрів і заголовків, що приходять ззовні.

Приклад задачі

Умова. Створити Servlet пошуку викликів, що читає параметр запиту, HTTP-заголовок та метод, а також зберігає атрибут для передачі далі.

Servlet у файлі CallSearchServlet.java.

Лістинг 64.1. Реалізація з покроковим розбором

Файл `CallSearchServlet.java` читання параметра, заголовка, методу

```
protected void doGet(
        HttpServletRequest req,
        HttpServletResponse resp)
        throws IOException {
    String q = req.getParameter("district");
    String ua = req.getHeader("User-Agent");
    String method = req.getMethod();
```

getParameter("district") читає параметр із рядка запиту (?district=...) або тіла форми. getHeader("User-Agent") читає HTTP-заголовок. getMethod() повертає HTTP-метод (тут «GET»). Це три різні джерела даних.

```
    req.setAttribute("dispatcher", "Коваль");
    resp.getWriter().write(
        "district=" + q + ", method=" + method);
}
```

setAttribute("dispatcher", "Коваль") зберігає атрибут у request-scope — він доступний фільтрам та іншим сервлетам у межах ЦЬОГО запиту, але не надсилається клієнту. У відповідь пишемо прочитані значення.

Зверніть увагу. *Метод getParameter завжди повертає String або null. Перед використанням значення району обов'язково перевіряйте його на null.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати Servlet, що отримує дані із запиту згідно теми варіанту.

№	Що реалізувати
1	Прочитати параметр ?callId=123 та вивести дані виклику.
2	Прочитати всі заголовки запиту через getHeaderNames.
3	Витягнути IP клієнта (робочого місця диспетчера) через getRemoteAddr.
4	Прочитати тіло POST з донесенням про пожежу через getReader.
5	Зберегти атрибут «черговий» у request та прочитати його у Filter.

Питання 65. Що таке ServletContext

Теоретична частина

ServletContext — об'єкт, що представляє весь web-додаток (один deployment unit) у контейнері. Він один на додаток і спільний для всіх сервлетів, фільтрів та JSP цього додатку [39].

ServletContext надає доступ до: init-параметрів додатку (елементи context-param у web.xml), атрибутів application-scope (доступних усьому додатку), ресурсів додатку (getResourceAsStream), логування (log), інформації про сервер (getServerInfo).

Атрибути application-scope, збережені через setAttribute, видимі всім компонентам додатку протягом усього часу його роботи — на відміну від request-scope (один запит) та session-scope (одна сесія користувача).

Типове застосування: зберігання спільних ресурсів — конфігурації, кешів, single-instance сервісів. Однак у сучасних додатках цю роль зазвичай виконує DI-контейнер (Spring). Отримати ServletContext можна через getServletContext() у сервлеті або через request.getServletContext().

Приклад задачі

Умова. Зберегти спільний довідник пожежних гідрантів у ServletContext (application-scope) та прочитати його з того ж або іншого сервлета.

Servlet у файлі HydrantServlet.java.

Лістинг 65.1. Реалізація з покроковим розбором

Файл `HydrantServlet.java` робота з application-scope атрибутами	

`protected void doGet(`
`  HttpServletRequest req,`
`  HttpServletResponse resp) {`
`  ServletContext ctx =`
`  req.getServletContext();`
`  ctx.setAttribute("hydrants",`
`  new HashMap<String, String>());`
	Отримуємо ServletContext із запиту. setAttribute("hydrants", ...) зберігає об'єкт у application-scope — він стає доступним усім сервлетам, фільтрам та JSP цього додатку протягом усього часу роботи.

`  String mode =`
`  ctx.getInitParameter("app.mode");`
`  Object cache = ctx.getAttribute("hydrants");`
`  ctx.log("hydrants initialized");`
`}`
	getInitParameter читає context-param із web.xml (конфігурація додатку). getAttribute повертає раніше збережений спільний об'єкт. ctx.log пише у журнал контейнера. Усе це — на рівні всього додатку.

Зверніть увагу. *ServletContext спільний для всіх користувачів. Не зберігайте в ньому дані конкретного диспетчера чи курсанта — для цього є сесія.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу зі ServletContext згідно теми варіанту.

№	Що реалізувати
1	Зберегти Map «частина → телефон» у ServletContext та прочитати з іншого сервлета.
2	Створити context-param region=Львівська у web.xml та прочитати його з коду.
3	Прочитати файл плану евакуації з ресурсів додатку через getResourceAsStream.
4	ServletContextListener для завантаження довідника частин при старті.
5	Порівняти application-scope та session-scope на прикладі лічильника звернень диспетчерів.

Питання 66. HttpSession та як її отримати

Теоретична частина

HttpSession — об'єкт, що зберігає стан, пов'язаний з конкретним користувачем (точніше — з його браузером) між кількома запитами. Це серверне сховище, що розв'язує проблему stateless-природи HTTP. Прив'язка до клієнта відбувається через cookie JSESSIONID: сервер створює сесію та надсилає її ідентифікатор у cookie; браузер повертає цей cookie з кожним наступним запитом, і сервер впізнає сесію [45].

Отримати сесію: req.getSession() — повертає наявну або створює нову; req.getSession(false) — повертає наявну або null (не створює). У сесію кладуть і беруть атрибути через setAttribute/getAttribute.

Життєвий цикл: сесія створюється, живе до виклику invalidate() або до закінчення таймауту бездіяльності (session-timeout у web.xml або setMaxInactiveInterval). Типове застосування — зберігання даних авторизованого користувача, кошика покупок.

У сучасних REST-API серверні сесії часто не використовують, надаючи перевагу stateless-автентифікації через токени (JWT), що краще масштабується.

Приклад задачі

Умова. Реалізувати збереження курсанта у сесії при вході до електронного журналу та читання його з сесії у наступному запиті.

Servlet у файлі LoginServlet.java.

Лістинг 66.1. Реалізація з покроковим розбором

Файл `LoginServlet.java` збереження користувача в сесії

```
protected void doPost(
        HttpServletRequest req,
        HttpServletResponse resp) {
    HttpSession session = req.getSession();
    String user =
        req.getParameter("username");
    session.setAttribute("user", user);
}
```

req.getSession() повертає наявну сесію або створює нову (з cookie JSESSIONID). setAttribute("user", ...) зберігає логін курсанта в сесії — він буде доступний у наступних запитах того ж браузера.

```
// В іншому запиті:
HttpSession s = req.getSession(false);
if (s != null) {
    String user =
        (String) s.getAttribute("user");
}
```

req.getSession(false) повертає наявну сесію або null (НЕ створює нову) — так перевіряють, чи курсант увійшов до системи. getAttribute("user") читає збережене значення. Сесія впізнається за cookie JSESSIONID.

З досвіду викладачів. *Після виклику session.invalidate() об'єкт сесії використовувати не можна: наступне звернення до нього кине IllegalStateException.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу з HttpSession згідно теми варіанту.

№	Що реалізувати
1	Servlet входу до кабінету курсанта, що кладе його в сесію.
2	Servlet виходу (logout), що викликає session.invalidate().
3	«Кошик» замовлення спорядження в сесії; додавання предметів та підрахунок кількості.
4	Налаштувати session-timeout 30 хвилин для системи чергової частини.
5	Пояснити, чому в REST-API часто не використовують серверні сесії.

Питання 67. Filter, його методи, мапінг до URL

Теоретична частина

Filter (фільтр) — компонент, що перехоплює запити до того, як вони досягнуть сервлета, і відповіді — перед відправкою клієнту. Фільтри реалізують наскрізні обов'язки (cross-cutting concerns): автентифікацію, логування, стиснення, CORS, кодування [38].

Інтерфейс Filter має три методи: init(FilterConfig) — ініціалізація, doFilter(req, resp, chain) — основна обробка, destroy() — звільнення ресурсів.

Ключовий момент — у doFilter обов'язково викликати chain.doFilter(req, resp), щоб передати управління далі по ланцюгу (наступному фільтру або сервлету). Якщо цього не зробити, обробка запиту зупиниться. Код ДО chain.doFilter виконується на вході (перед сервлетом), код ПІСЛЯ — на виході (перед відправкою клієнту).

Кілька фільтрів утворюють ланцюг (FilterChain); порядок визначається мапінгом. Це дозволяє нанизувати обробники один на одного. Мапінг: анотація @WebFilter("/path") або web.xml. Можна мапити за URL-патерном, за іменем сервлета чи за типом диспетчеризації (REQUEST/FORWARD/INCLUDE/ERROR).

Приклад задачі

Умова. Створити фільтр для системи чергової частини, що логує час обробки кожного запиту, демонструючи код до та після chain.doFilter.

Filter у файлі LogFilter.java з мапінгом на всі запити.

Лістинг 67.1. Реалізація з покроковим розбором

Файл `LogFilter.java` логування часу обробки

```
@WebFilter("/*")
public class LogFilter implements Filter {

    public void doFilter(
            ServletRequest req,
            ServletResponse resp,
            FilterChain chain)
            throws IOException, ServletException {
        long t = System.currentTimeMillis();
```

@WebFilter("/*") мапить фільтр на ВСІ запити додатку. Код до chain.doFilter виконується на вході. Тут фіксуємо час початку обробки — це відбувається ще ДО того, як запит досягне сервлета.

```
        chain.doFilter(req, resp);

        long dt = System.currentTimeMillis() - t;
        System.out.println("оброблено за " + dt +
" мс");
    }
}
```

chain.doFilter передає управління далі (наступному фільтру або сервлету) — ОБОВ'ЯЗКОВО, інакше запит зупиниться. Код після нього виконується на виході: рахуємо тривалість обробки. Так фільтр «обгортає» виконання сервлета.

Зверніть увагу. *Порядок виконання кількох фільтрів визначається web.xml або анотацією @Order у Spring. Без явного налаштування порядок фільтрів з @WebFilter не гарантується.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати фільтр згідно теми варіанту у окремому файлі.

№	Що реалізувати
1	Фільтр, що логує URL кожного запиту до системи викликів.
2	AuthFilter, що перевіряє наявність диспетчера в сесії.
3	Фільтр, що встановлює кодування UTF-8 (адреси українською) для запиту та відповіді.
4	CORS-фільтр для карти викликів, що додає заголовок Access-Control-Allow-Origin.
5	Фільтр, що додає заголовок X-Request-ID до кожної відповіді.

Питання 68. Структура WAR. Як контейнер деплоїть

Теоретична частина

WAR (Web Application Archive) — це ZIP-архів зі стандартизованою структурою для розгортання web-додатку в сервлет-контейнері. Структура WAR: у корені та поза каталогом WEB-INF розташовані статичні ресурси (HTML, CSS, JS, зображення), доступні клієнту напряму за URL. Каталог WEB-INF — приватний, його вміст недоступний клієнту через HTTP. У ньому: web.xml (дескриптор розгортання, опційний з Servlet 3.0+), classes/ (скомпільовані .class-файли) та lib/ (JAR-залежності) [39; 40].

Те, що WEB-INF недоступний ззовні, — це функція безпеки: туди кладуть класи, конфігурацію, бібліотеки, які не повинні бути завантажені клієнтом.

Процес деплою: контейнер копіює WAR у свій каталог (webapps у Tomcat), розпаковує архів, читає web.xml та анотації, створює ServletContext, ініціалізує компоненти у порядку: listeners → filters → servlets. URL додатку: http://host/context-path/, де context-path зазвичай збігається з ім'ям WAR-файлу.

Сучасна альтернатива WAR — Spring Boot fat-JAR із вбудованим контейнером (Tomcat/Jetty), що запускається як звичайний java -jar app.jar.

Приклад задачі

Умова. Розглянути типову структуру WAR-архіву веб-застосунку чергової частини та призначення його основних частин.

Дерево файлів WAR-архіву (dispatch.war).

Лістинг 68.1. Реалізація з покроковим розбором

Файл `dispatch.war` структура архіву (дерево)

```
dispatch.war
├─ index.html
├─ css/style.css
├─ js/app.js
└─ WEB-INF/
   ├─ web.xml
   ├─ classes/
   │  └─ ua/ldubgd/
   │     └─ StatusServlet.class
   └─ lib/
      └─ jdbc-driver.jar
```

У корені — статичні ресурси (index.html, css, js), доступні клієнту за URL. WEB-INF — приватний каталог: web.xml (дескриптор), classes/ (скомпільовані сервлети), lib/ (JAR-залежності). Вміст WEB-INF недоступний клієнту напряму.

Зверніть увагу. *Відкрийте будь-який WAR-файл архіватором — це звичайний ZIP. Побачити структуру власними очима корисніше, ніж прочитати про неї.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу описати або підготувати елемент розгортання web-додатку згідно теми варіанту.

№	Що реалізувати
1	Зібрати WAR системи обліку курсантів через Maven (war packaging) та розгорнути на Tomcat.
2	Створити web.xml із servlet-mapping для сервлета /incidents.
3	Пояснити, чому WEB-INF недоступний клієнту через HTTP.
4	Запакувати JDBC-драйвер у WEB-INF/lib та використати його у сервлеті журналу викликів.
5	Порівняти класичний WAR із Spring Boot fat-JAR.

ЧАСТИНА 4. ДОПОМІЖНІ ТЕХНОЛОГІЇ

Розділ 9. Алгоритми, структури даних і безпека

Питання 69. Що таке рекурсія

Теоретична частина

Рекурсія — спосіб розв'язання задачі, при якому функція викликає сама себе для розв'язання меншої підзадачі того ж типу. Класичний приклад — обчислення факторіалу n! = n × (n−1)! з базовим випадком 0! = 1. Кожна рекурсивна функція має містити два обов'язкові елементи: базовий випадок (base case) — умова, при якій рекурсія припиняється і повертається явне значення; рекурсивний крок — виклик функції самої себе зі зменшеним аргументом. Без базового випадку рекурсія буде нескінченною і призведе до StackOverflowError, оскільки JVM має обмежений стек викликів [62; 64].

Кожен рекурсивний виклик додає кадр (frame) у стек: зберігаються локальні змінні, аргументи, точка повернення. Глибина рекурсії обмежена розміром стека (за замовчуванням приблизно 256–1024 КБ у JVM, що дозволяє кілька тисяч викликів). Тому глибока рекурсія може бути небезпечна.

Рекурсія елегантна для задач з природною ієрархічною структурою: обхід дерев, графів, рекурсивних структур даних, divide-and-conquer алгоритми (merge sort, quicksort, бінарний пошук). Для задач з лінійним характером (сума елементів) ітеративний підхід зазвичай ефективніший.

Існує спеціальна оптимізація — tail-call optimization, коли рекурсивний виклик є останньою операцією; компілятор може перетворити його на цикл. Java не підтримує цю оптимізацію — у глибокій рекурсії краще переписувати на цикл.

На практичних заняттях ми радимо курсантам одну вправу: розписати на аркуші стек викликів для factorial(4). Після цього рекурсія перестає здаватися «магією».

Приклад задачі

Умова. Підрахувати рекурсивно кількість можливих способів розташування n рятувальників у шерензі (n!) та продемонструвати роботу для бойового розрахунку з n=5 осіб.

Рекурсивний метод у файлі Demo.java.

Лістинг 69.1. Реалізація з покроковим розбором

Файл `Demo.java` рекурсивне обчислення факторіалу	

`public class Demo {`
`  static long factorial(int n) {`
`  if (n <= 1) return 1;`
`  return n * factorial(n - 1);`
`  }`
	Базовий випадок: при n ≤ 1 повертаємо 1 — це зупиняє рекурсію. Рекурсивний крок: виклик factorial(n - 1) із меншим аргументом. Кожен виклик додає кадр у стек і чекає на результат внутрішнього.

`  public static void main(String[] args) {`
`  System.out.println(factorial(5));`
`  }`
`}`
	factorial(5) → 5 × factorial(4) → 5 × 4 × factorial(3) → ... → 5 × 4 × 3 × 2 × 1 = 120. Отже, відділення з 5 рятувальників можна вишикувати 120 способами. На піку стек містить 5 кадрів методу factorial (плюс кадр main), потім розкручується назад. Вивід: «120».

Порада. *Спробуйте обчислити factorial(25) з типом long. Результат буде від'ємним через переповнення — для великих значень потрібен BigInteger.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати рекурсивну функцію згідно теми варіанту у файлі Demo.java. Перевірити роботу на трьох тестових входах.

№	Що реалізувати рекурсивно
1	Сума цифр номера пожежно-рятувальної частини n.
2	Число Фібоначчі F(n) (умовна модель зростання кількості осередків займання).
3	Степінь pow(x, n) для цілого n ≥ 0 (поширення вогню з коефіцієнтом x).
4	Реверс позивного «ЛЬВІВ-101» через рекурсію.
5	Найбільший спільний дільник gcd(a, b) кількості курсантів двох взводів (алгоритм Евкліда).

Питання 70. Що таке дерево

Теоретична частина

Дерево (tree) — це ієрархічна нелінійна структура даних, що складається з вузлів (nodes), з'єднаних ребрами (edges). Має один спеціальний вузол — корінь (root), з якого починається структура. Кожен вузол може мати дітей (children), а вузол-предок називається батьком (parent). Вузли без дітей — листя (leaves) [60; 63].

Формальне визначення: дерево — це неорієнтований зв'язний ациклічний граф з виділеним коренем. Між будь-якими двома вузлами існує рівно один шлях. Висота дерева — найдовший шлях від кореня до листа.

Найпоширеніший тип — бінарне дерево, де кожен вузол має не більше двох дітей (left і right). Бінарне дерево пошуку (BST) додає інваріант: усі ключі в лівому піддереві менші за ключ вузла, у правому — більші. Це дає O(log n) пошук, вставку, видалення (у збалансованому дереві).

Збалансовані дерева (AVL, red-black) автоматично підтримують малу висоту, гарантуючи O(log n). Стандартні TreeMap і TreeSet у Java реалізовані на red-black tree. Інші

типи: trie (для рядків), B-tree та B+ tree (для баз даних і файлових систем), heap (купа — спеціальне бінарне дерево для пріоритетних черг). Способи обходу: in-order (LNR), pre-order (NLR), post-order (LRN), BFS (за рівнями).

Приклад задачі

Умова. Реалізувати просте бінарне дерево пошуку (BST) для швидкого пошуку номерів викликів у журналі чергової частини з операцією вставки та обходом in-order.

Клас Node та BST у файлі BST.java; демонстрація — у файлі Main.java.

Лістинг 70.1. Реалізація з покроковим розбором

Файл `BST.java` вузол та операції дерева

```
class Node {
    int value;
    Node left, right;
    Node(int v) { value = v; }
}

class BST {
    Node root;
```

Node містить значення та два посилання — на лівого та правого нащадка. Клас BST зберігає корінь. Структура рекурсивна: кожен Node сам є коренем свого піддерева.

```
    void insert(int v) {
        root = insert(root, v);
    }
    Node insert(Node n, int v) {
        if (n == null) return new Node(v);
        if (v < n.value) n.left = insert(n.left,
v);
        else             n.right = insert(n.right,
v);
        return n;
    }
}
```

Рекурсивна вставка: якщо досягли null — створюємо новий Node. Якщо нове значення менше — йдемо ліворуч, інакше — праворуч. Інваріант BST дотримується автоматично. Складність O(log n) для збалансованого дерева, O(n) у найгіршому випадку.

Зверніть увагу. *Якщо вставляти номери у зростаючому порядку (1, 2, 3, ...), дерево виродиться в «список» і пошук стане O(n). Саме тому в бібліотеках використовують збалансовані дерева.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати операцію над деревом згідно теми варіанту у відповідному файлі.

№	Що реалізувати
1	BST номерів пожежно-рятувальних частин з операціями insert та contains.
2	Обхід in-order: вивести номери викликів у відсортованому порядку.
3	Обхід pre-order та post-order дерева підпорядкування підрозділів; порівняти послідовності.
4	Обчислити висоту дерева номерів аудиторій через рекурсію.
5	Підрахувати кількість листя (частин без підпорядкованих постів).

Питання 71. Граф. Алгоритми обходу. O-нотація. SQL-ін’єкція

Теоретична частина

Граф (graph) — структура даних з вузлів (vertices) та ребер (edges), що з’єднують пари вузлів. На відміну від дерева, граф може мати цикли і не має кореня. Графи бувають орієнтованими (directed — ребро має напрямок) та неорієнтованими, зваженими (з вагами

на ребрах) та незваженими. Способи представлення в пам’яті: матриця суміжності (adjacency matrix) — 2D-масив n×n, O(1) перевірка ребра, O(n^2) пам’яті; список суміжності (adjacency list) — для кожного вузла список сусідів, O(V+E) пам’яті, ефективніший для розріджених графів [60; 61].

Два класичні алгоритми обходу: BFS (Breadth-First Search) — обхід у ширину через чергу (Queue), знаходить найкоротший шлях у незваженому графі; DFS (Depth-First Search) — обхід у глибину через стек або рекурсію, корисний для пошуку циклів, топологічного сортування. Складність обох — O(V+E).

O-нотація (Big-O) описує асимптотичну верхню межу зростання витрат алгоритму (часу чи пам’яті) залежно від розміру вхідних даних n. O(1) — константний час, O(log n) — логарифмічний, O(n) — лінійний, O(n log n) — типовий для ефективних сортувань, O(n^2) — квадратичний (вкладені цикли), O(2^n) — експоненційний. Зазвичай її використовують для оцінки найгіршого випадку.

SQL-ін’єкція — атака, при якій зловмисник вставляє SQL-код у параметри запиту через рядки введення. Якщо програма конкатенує введення у SQL-текст, атакувальник може змінити логіку запиту, отримати дані інших користувачів, видалити таблиці. Захист — параметризовані запити (PreparedStatement з ?-параметрами), де значення передаються окремо і не інтерпретуються як SQL.

Приклад задачі

Умова. Реалізувати неорієнтований граф доріг між пожежно-рятувальними частинами та об’єктами району виїзду через список суміжності та обхід BFS від частини, що отримала виклик.

Клас Graph у файлі Graph.java з методом bfs.

Лістинг 71.1. Реалізація з покроковим розбором

Файл `Graph.java` граф через список суміжності

```
import java.util.*;

class Graph {
    Map<Integer, List<Integer>> adj = new
HashMap<>();

    void addEdge(int u, int v) {
        adj.computeIfAbsent(u, k -> new
ArrayList<>()).add(v);
        adj.computeIfAbsent(v, k -> new
ArrayList<>()).add(u);
    }
```

Список суміжності: для кожного вузла зберігаємо список сусідів. addEdge додає ребро з обох боків (граф неорієнтований). computeIfAbsent ініціалізує список при першому зверненні. Складність представлення — O(V+E).

```
    void bfs(int start) {
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
        queue.add(start); visited.add(start);
        while (!queue.isEmpty()) {
            int v = queue.poll();
            System.out.print(v + " ");
            for (int n : adj.getOrDefault(v,
List.of()))
                if (visited.add(n)) queue.add(n);
        }
    }
}
```

BFS використовує чергу (FIFO). Починаємо зі стартового вузла (частини). Кожну ітерацію беремо вузол з черги, виводимо його, додаємо у чергу всіх неотриманих сусідів. visited.add повертає true лише якщо елемент новий — це уникає повторного відвідування. BFS знаходить об’єкти в порядку зростання кількості перегонів від частини. Складність O(V+E).

Зверніть увагу. *BFS гарантує найкоротший маршрут лише за кількістю ребер. Для реальних доріг з різною довжиною потрібен алгоритм Дейкстри.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати алгоритм або захист згідно теми варіанту у файлі Demo.java або Graph.java.

№	Що реалізувати
1	Граф доріг між 6 частинами через список суміжності; знайти найкоротший (за кількістю ребер) маршрут BFS.
2	Граф поверхів і сходових клітин будівлі та обхід DFS рекурсивно (план обстеження).
3	Граф через матрицю суміжності для 5 корпусів університету.
4	Продемонструвати SQL-ін’єкцію в пошуку курсанта та її блокування через PreparedStatement.
5	Виміряти час лінійного та бінарного пошуку номера виклику на масиві 1 000 000; порівняти O(n) vs O(log n).

Питання 72. Authorization vs authentication

Теоретична частина

Це дві принципово різні концепції безпеки, які часто плутають у повсякденній мові. Authentication (автентифікація, AuthN) — процес підтвердження того, ХТО користувач. Відповідає на питання «чи ти справді той, за кого себе видаєш». Методи: пароль (щось, що знаєш); SMS-код, апаратний токен (щось, що маєш); відбиток пальця, обличчя (щось, ким ти є). Багатофакторна автентифікація (MFA) комбінує кілька з цих методів [57; 58].

Authorization (авторизація, AuthZ) — процес визначення, ЩО автентифікований користувач може робити. Відповідає на питання «чи дозволено тобі виконати цю дію». Здійснюється через ролі (RBAC — Role-Based Access Control), політики (ABAC — Attribute-Based Access Control), списки доступу (ACL).

Порядок завжди такий: спочатку автентифікація, потім авторизація. Не можна авторизувати того, хто не автентифікований — система не знає, кому надавати права.

У HTTP заголовок Authorization, попри свою назву, насправді використовується насамперед для автентифікації: він несе токен (Bearer eyJ...) або basic-credentials, за якими сервер встановлює особу користувача, і лише після цього перевіряє його права. Через це виникає термінологічна плутанина.

Приклад задачі

Умова. Реалізувати простий фільтр для системи чергової частини, що виконує автентифікацію (перевірка токена) та авторизацію (перевірка ролі DISPATCHER) на захищеному ендпоінті.

Фільтр у файлі SecurityFilter.java для Servlet-додатку.

Лістинг 72.1. Реалізація з покроковим розбором

Файл `SecurityFilter.java` автентифікація + авторизація

```
@WebFilter("/dispatch/*")
public class SecurityFilter implements Filter {
    public void doFilter(ServletRequest req,
            ServletResponse resp, FilterChain
chain) {
        HttpServletRequest r =
(HttpServletRequest) req;
        String token =
r.getHeader("Authorization");
```

Фільтр захищає всі URL під /dispatch/. Перший крок — взяти токен із заголовка Authorization. Це сировина для АВТЕНТИФІКАЦІЇ — ми ще не знаємо, чи токен валідний і кому належить.

```
        User user = authenticate(token);
        if (user == null) {
            ((HttpServletResponse)
resp).setStatus(401);
            return;
        }
        if (!user.hasRole("DISPATCHER")) {
            ((HttpServletResponse)
resp).setStatus(403);
            return;
        }
        chain.doFilter(req, resp);
    }
}
```

authenticate(token) — АВТЕНТИФІКАЦІЯ: повертає User або null. Невалідний токен → 401 Unauthorized. Далі АВТОРИЗАЦІЯ: hasRole("DISPATCHER") — лише диспетчер може реєструвати виклики. Немає ролі → 403 Forbidden. Усе пройшло — chain.doFilter передає управління далі.

З досвіду викладачів. *Код 401 означає «не знаю, хто ви», а 403 — «знаю, хто ви, але сюди вам не можна». Ця коротка формула добре запам'ятовується.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати або описати елемент безпеки згідно теми варіанту у відповідному файлі.

№	Що реалізувати
1	Фільтр електронного журналу, що повертає 401 при відсутньому токені.
2	Фільтр, що повертає 403, якщо курсант намагається виставити оцінку (роль LECTURER).
3	Описати у коментарях типи MFA-факторів (knowledge, possession, inherence) для доступу до системи ДСНС.
4	Реалізувати клас Role (CADET, LECTURER, DISPATCHER) з методами hasPermission.
5	Пояснити різницю між 401 та 403 у HTTP.

Питання 73. Що таке phishing

Теоретична частина

Phishing (фішинг) — вид соціальної інженерії, при якому зловмисник видає себе за довірену сторону (банк, компанію, колегу) для виманювання конфіденційних даних: паролів, номерів карток, токенів доступу [58; 59].

Типові механізми: підроблені листи від «банку» з посиланням на підроблений сайт; SMS-повідомлення про «заблокований акаунт» (smishing); телефонні дзвінки від «служби безпеки» (vishing). Сайти-підробки часто візуально майже не відрізняються від оригіналу і використовують схожі домени (paypa1.com замість paypal.com — homograph attack).

Spear phishing — цільовий фішинг проти конкретної особи з використанням попередньо зібраної інформації (ім'я, посада, проєкти). Whaling — атака на топ-

менеджмент. BEC (Business Email Compromise) — підробка електронної пошти керівника з проханням терміново перевести гроші.

Захист на стороні користувача: перевіряти URL у адресному рядку (не натискати посилання з листів сліпо), увімкнути MFA (один пароль уже недостатній для входу), не вводити дані з листів-сповіщень. На стороні організації: SPF/DKIM/DMARC для пошти, регулярне навчання співробітників, симуляції фішингу. Технічні заходи у застосунках: HTTPS з валідним сертифікатом, заголовки безпеки (CSP, X-Frame-Options), правильна обробка зовнішніх посилань (атрибут rel="noopener noreferrer").

Приклад задачі

Умова. Реалізувати простий валідатор URL, що позначає посилання як підозрілі за типовими ознаками фішингу (домени, схожі на dsns.gov.ua чи ldubgd.edu.ua, незахищений HTTP, IP-адреса замість домену) — захист працівників і курсантів від фішингових розсилок.

Утилітний клас у файлі PhishingDetector.java.

Лістинг 73.1. Реалізація з покроковим розбором

Файл `PhishingDetector.java` евристики для виявлення фішингу

```
public class PhishingDetector {
    public static List<String> check(String url) {
        List<String> warnings = new ArrayList<>();

        if (!url.startsWith("https://"))
            warnings.add("Не HTTPS");
```

Перша евристика: HTTP замість HTTPS — потенційна небезпека (відсутнє шифрування і перевірка сертифіката). Не означає 100% фішинг, але привід насторожитися.

```
        if (url.matches(".*\\d+\\.\\d+\\.\\d+\\.\
\d+.*"))
            warnings.add("IP-адреса замість
домену");

        if (url.contains("dsn5") ||
url.contains("1dubgd"))
            warnings.add("Підозріла назва
(homograph attack)");

        return warnings;
    }
}
```

Друга евристика: IP замість домену (фішингові сайти часто на сирих IP). Третя: homograph attack — заміна літер на схожі цифри (dsn5 замість dsns, 1dubgd замість ldubgd). Реальні системи безпеки використовують десятки таких правил плюс машинне навчання.

Зверніть увагу. *Евристики з прикладу легко обійти, тому вони лише доповнюють навчання людей, а не замінюють його. Найкращий захист від фішингу — уважний користувач.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати елемент захисту від фішингу або описати тип атаки згідно теми варіанту у відповідному файлі.

№	Що реалізувати або описати
1	Перевірка URL на наявність HTTPS та офіційного домену gov.ua / edu.ua.
2	Описати у коментарях 5 видів phishing-атак на працівників ДСНС.
3	Перевірка домену електронної пошти на підробки адреси університету.
4	Описати роботу SPF/DKIM/DMARC у коментарях.
5	Реалізувати попередження при кліку на зовнішнє посилання в системі електронного журналу.

Питання 74. Offset vs timezone

Теоретична частина

Це дві різні, хоч і пов’язані концепції роботи з часом, які часто плутають. Offset (UTC offset) — фіксована різниця у часі від UTC, виражена як «+03:00» або «-08:00». Це просте число — не залежить від географії, історії, сезону. У Java представлений класами ZoneOffset та OffsetDateTime [16].

Timezone (часовий пояс) — географічний регіон з правилами, які визначають, який offset діє в кожен момент часу. Наприклад, пояс «Europe/Kyiv» — це +02:00 взимку та +03:00 влітку (через перехід на літній час). У Java це ZoneId та ZonedDateTime.

Принципова відмінність: offset «+03:00» — це константа, її значення не змінюється. Timezone «Europe/Kyiv» — це правило, що може давати різні offset'и залежно від дати: 1 січня — +02:00, 1 липня — +03:00. Це робить timezone «розумнішим», але й складнішим: правила можуть змінюватися (наприклад, скасування переходу на літній час).

Дані про правила часових поясів зберігаються в IANA Time Zone Database (tzdata), що оновлюється. JVM використовує її. Коли уряд змінює правила переходу, потрібно оновити tzdata.

Як обирати: для зберігання моменту часу в БД — використовуйте UTC або timestamp з offset. Для відображення користувачу — конвертуйте у його timezone. Уникайте «наївного» LocalDateTime без зони — це джерело багів при перетинах поясів.

Приклад задачі

Умова. Продемонструвати на часі надходження виклику різницю між OffsetDateTime та ZonedDateTime: створити обидва, додати 6 місяців і подивитися, що зміниться (DST).

Демонстрація у файлі TimeDemo.java з java.time.

Лістинг 74.1. Реалізація з покроковим розбором

Файл `TimeDemo.java` Offset не змінюється; ZoneId враховує DST

```
import java.time.*;

public class TimeDemo {
  public static void main(String[] args) {
    OffsetDateTime offsetDt =
        OffsetDateTime.of(2024, 1, 15, 12, 0, 0,
0, ZoneOffset.of("+02:00"));
    ZonedDateTime zoneDt =
        ZonedDateTime.of(2024, 1, 15, 12, 0, 0, 0,
ZoneId.of("Europe/Kyiv"));
```

Обидва представляють момент надходження виклику 15 січня 2024, 12:00. OffsetDateTime жорстко фіксує +02:00. ZonedDateTime використовує правила Europe/Kyiv — взимку це +02:00, так що значення збігається.

```
    System.out.println(offsetDt.plusMonths(6));
    System.out.println(zoneDt.plusMonths(6));
  }
}
```

Додаємо 6 місяців → 15 липня. OffsetDateTime лишається з offset +02:00 (не змінюється). ZonedDateTime через DST автоматично перейшов на +03:00 (літній час). Це принципова різниця: timezone «знає» про переходи, offset — ні.

Зверніть увагу. *Час подій у системі чергової частини завжди зберігайте в UTC. Перехід на літній час посеред нічного чергування — класичне джерело плутанини в журналах.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу з часом згідно теми варіанту у файлі TimeDemo.java.

№	Що реалізувати
1	Створити ZonedDateTime виклику в різних часових поясах (Київ, Варшава, UTC) і вивести їх.
2	Конвертувати поточний UTC-час сервера у локальний час чергової частини.
3	Обчислити тривалість гасіння пожежі (Duration) між часом прибуття та ліквідації.
4	Продемонструвати, як змінюється offset при переході на літній час під час нічного чергування.
5	Зберегти час виклику в БД як UTC та відобразити у локальній зоні.

Розділ 10. Формати обміну даними

Питання 75. Що таке JSON

Теоретична частина

JSON (JavaScript Object Notation) — текстовий формат серіалізації структурованих даних, що походить із синтаксису об'єктних літералів JavaScript. Завдяки простоті та широкій підтримці став де-факто стандартом для обміну даними у веб-API [47; 50].

Має шість типів значень: string (у подвійних лапках), number (без лапок), boolean (true/false), null, object (пари «ключ»: значення у фігурних дужках), array (значення у квадратних дужках). Ключі завжди рядкові.

Переваги: легко читається людиною, компактний у порівнянні з XML, нативно підтримується JavaScript, мінімальний overhead для серіалізації/десеріалізації. Недоліки: не підтримує коментарі, дати, бінарні дані (доводиться кодувати як рядки чи base64), немає чіткої схеми (хоча є JSON Schema).

У Java популярні бібліотеки: Jackson (стандарт де-факто, особливо в Spring), Gson (від Google), org.json (мінімальна). Усі дозволяють серіалізувати Java-об'єкт у JSON-рядок та десеріалізувати назад. Альтернативи для специфічних задач: YAML (для конфігурацій, з відступами), TOML (теж конфігурації), Protocol Buffers, Avro, MessagePack (бінарні, ефективніші для високонавантажених сервісів).

Приклад задачі

Умова. Серіалізувати об'єкт Rescuer у JSON-рядок та десеріалізувати назад через Jackson.

Клас Rescuer у файлі Rescuer.java; робота з Jackson — у файлі JsonDemo.java.

Лістинг 75.1. Реалізація з покроковим розбором

Файл `Rescuer.java` POJO для серіалізації

```
public class Rescuer {
    private String name;
    private int age;

    public Rescuer() {}
    public Rescuer(String name, int age) {
        this.name = name; this.age = age;
    }
    // геттери / сеттери
}
```

Звичайний POJO: приватні поля, конструктор за замовчуванням (потрібен Jackson для десеріалізації) та геттери/сеттери. Жодних анотацій — Jackson сам відобразить поля на JSON-ключі.

Файл `JsonDemo.java` серіалізація та десеріалізація

```
import
com.fasterxml.jackson.databind.ObjectMapper;

ObjectMapper m = new ObjectMapper();
Rescuer u = new Rescuer("Коваль", 25);
String json = m.writeValueAsString(u);
// {"name":"Коваль","age":25}

Rescuer back = m.readValue(json, Rescuer.class);
```

ObjectMapper — центральний клас Jackson. writeValueAsString перетворює об'єкт у JSON-рядок. readValue робить зворотну операцію: з JSON у Java-об'єкт. Поля визначаються автоматично за геттерами/сеттерами.

Порада. *Якщо прибрати конструктор без параметрів, десеріалізація впаде з винятком InvalidDefinitionException. Ця помилка часто трапляється на перших заняттях.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу з JSON через Jackson згідно теми варіанту у відповідному файлі.

№	Що реалізувати
1	Серіалізувати List<FireCall> у JSON-масив.
2	Десеріалізувати JSON-масив у List<Cadet> через TypeReference.
3	Прочитати довідник частин із JSON-файлу через readValue(File, ...).
4	Використати @JsonIgnore для приховування поля password облікового запису курсанта.
5	Налаштувати pretty-print донесення: writerWithDefaultPrettyPrinter().

Питання 76. Чи можна виконати JSON як код

Теоретична частина

Найточніша відповідь: JSON — це формат даних, він не призначений для виконання. JSON-документ сам по собі не є програмою — це лише структурований текст. У Java, Python, C# та інших мовах JSON парситься у структури даних (Map, List, об'єкти), які можна потім використовувати, але не виконувати [47].

Історично JSON виник із підмножини синтаксису об'єктних літералів JavaScript. У ранніх веб-додатках JSON «виконували» через eval() браузера: var obj = eval('(' + jsonString + ')'). Це парсило JSON завдяки тому, що JS-двигун обчислював його як об'єктний літерал.

Цей підхід давно вважається небезпечним і застарілим. eval виконує БУДЬ-ЯКИЙ JavaScript-код, тому якщо у JSON потрапить шкідливий вираз (через XSS, MITM, ненадійне джерело), він буде виконаний з повними правами поточної сторінки. Сучасні браузери надають JSON.parse — спеціальний парсер, що приймає лише валідний JSON і не виконує коду.

У серверних середовищах Node.js та інших eval над JSON — антипатерн. Використовують JSON.parse або еквіваленти. Тому правильна відповідь на співбесіді: JSON — це дані, не код; теоретично можна змусити JS-двигун виконати JSON через eval, але це небезпечно й не використовується. Замість цього — JSON.parse або еквівалентні парсери у кожній мові.

Приклад задачі

Умова. Продемонструвати на даних про рятувальника безпечний JSON.parse та небезпечний eval у JavaScript для розуміння різниці.

Простий приклад у файлі demo.js — текстове пояснення для аудиторії.

Лістинг 76.1. Реалізація з покроковим розбором

Файл `demo.js` JSON.parse — безпечно, eval — небезпечно

```
const safe = '{"name":"Коваль","age":25}';

// БЕЗПЕЧНО: спеціалізований парсер
const obj1 = JSON.parse(safe);
console.log(obj1.name);  // Коваль
```

JSON.parse приймає лише валідний JSON. Якщо у вхідному рядку буде JS-код (виклик функції, оголошення), парсер кине SyntaxError — нічого не виконає. Це коректний інструмент.

```
// НЕБЕЗПЕЧНО: виконання довільного коду
const danger =
    '{"a":1, "b":alert("hacked")}';
// eval('(' + danger + ')');
// → виконає alert('hacked')!
```

eval виконає БУДЬ-ЯКИЙ JavaScript. Якщо у «JSON» з'явиться виклик alert або щось гірше (надсилання токенів зловмиснику), він спрацює. Тому eval над JSON — антипатерн. Завжди використовуйте JSON.parse.

Зверніть увагу. *Функцію eval у JavaScript варто сприймати як «червоний прапорець»: якщо ви бачите її в коді, що обробляє зовнішні дані, — це майже напевно вразливість.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу пояснити різницю між JSON та виконуваним кодом згідно теми варіанту у відповідному файлі (Java або JavaScript).

№	Що реалізувати або описати
1	У JS: показати JSON.parse vs eval на даних виклику та описати небезпеку.
2	У Java: показати, що Jackson.readValue не може виконати код, вбудований у JSON донесення.
3	Описати у коментарях, чому JSON не є мовою програмування.
4	Продемонструвати парсинг JSON з даними курсанта у Python через json.loads.
5	Реалізувати власний міні-парсер JSON у Java (для одного рівня).

Питання 77. Що таке XML. Чи можна виконати XML як код

Теоретична частина

XML (eXtensible Markup Language) — текстовий формат розмітки для зберігання та обміну структурованими даними. Розроблений у 1990-х W3C як універсальний, розширюваний формат. Документ складається з тегів, що утворюють ієрархічне дерево [48].

Базовий синтаксис: відкривальний та закривальний теги <name>...</name>, можливі атрибути <name id="1">, самозакривні теги <br/>, текстовий вміст, коментарі <!-- -->. Документ має один кореневий елемент. На відміну від HTML, XML суворий: усі теги мусять бути закриті, атрибути в лапках.

Переваги: суворо структурований, підтримує схеми (XSD, DTD) для валідації, namespaces для уникнення конфліктів імен, потужні засоби трансформації (XSLT) та запитів (XPath). Недоліки порівняно з JSON: багатослівніший, складніший для парсингу, повільніший. Сфери застосування: SOAP web-сервіси, конфігурація (Maven pom.xml,

web.xml, Spring beans.xml), документообіг (Office Open XML, ODF), карти (KML, GPX). У сучасних REST-API переважно витіснений JSON.

Чи можна виконати XML як код: ні — XML є форматом даних, не мовою програмування. Однак існує XSLT (eXtensible Stylesheet Language Transformations) — Turing-повна мова на основі XML для трансформації XML-документів. XSLT-таблиці виконуються XSLT-процесорами (Saxon, Xalan). Це окрема технологія, не «виконання XML», а інтерпретація XML як інструкцій для трансформації.

Приклад задачі

Умова. Розглянути типовий XML-документ зі списком рятувальників караулу та продемонструвати його розбір у Java через DOM-парсер.

XML-файл rescuers.xml; розбір у файлі XmlDemo.java.

Лістинг 77.1. Реалізація з покроковим розбором

Файл `rescuers.xml` приклад XML-документа

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rescuers>
    <rescuer id="1">
        <name>Коваль</name>
        <age>25</age>
    </rescuer>
</rescuers>
```

Заголовок з версією та кодуванням. Кореневий елемент <rescuers>, що містить <rescuer> з атрибутом id. Дочірні елементи <name> та <age> з текстовим вмістом. Структура чітко ієрархічна.

Файл `XmlDemo.java` розбір через DOM

```java
import javax.xml.parsers.*;
import org.w3c.dom.*;

DocumentBuilder b = DocumentBuilderFactory
    .newInstance().newDocumentBuilder();
Document doc = b.parse("rescuers.xml");
NodeList list = doc.getElementsByTagName("name");
System.out.println(list.item(0).getTextContent());
```

DocumentBuilderFactory створює DOM-парсер. parse читає файл і будує дерево у пам’яті. getElementsByTagName знаходить усі елементи <name>. getTextContent повертає текстовий вміст. Вивід: «Коваль».

Зверніть увагу. *DOM-парсер завантажує в пам’ять увесь документ. Для журналу з мільйонами записів обирайте SAX або StAX (див. питання 85).*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати роботу з XML згідно теми варіанту у відповідному файлі.

№	Що реалізувати
1	Створити XML-файл з переліком пожежних автомобілів частини та прочитати його через DOM.
2	Створити XML донесення про пожежу програмно через DocumentBuilder.
3	Знайти всі елементи <hydrant> з атрибутом id через XPath.
4	Серіалізувати об’єкт FireCall у XML через Jackson XML.
5	Описати у коментарях, що таке XSLT і чим відрізняється від XML.

Розділ 11. Операційна система Linux

Питання 78. Що таке Linux. Дистрибутиви Linux

Теоретична частина

Linux — це Unix-подібне ядро операційної системи з відкритим кодом, створене Лінусом Торвальдсом у 1991 році. Ядро (kernel) — центральна частина ОС, що керує процесами, пам'яттю, файловими системами, мережею та апаратурою. Саме лише ядро не є повною ОС — для роботи потрібні утиліти, оболонка, бібліотеки [51; 52].

Дистрибутив Linux — це повна ОС, що складається з ядра Linux плюс набору програм: GNU coreutils, оболонки (bash, zsh), системи ініціалізації (systemd), пакетного менеджера, графічного середовища (GNOME, KDE), допоміжних застосунків.

Популярні дистрибутиви для серверів: Ubuntu Server (на базі Debian, найпопулярніший у хмарах), Debian (стабільність), Red Hat Enterprise Linux (комерційна підтримка), CentOS Stream та його наступники (Rocky Linux, AlmaLinux), Alpine Linux (мінімальний, популярний у Docker-образах).

Для робочих станцій: Ubuntu, Fedora, Linux Mint, openSUSE, Arch Linux, Manjaro. Принципи дистрибутивів відрізняються: rolling release (постійні оновлення, Arch) vs LTS (довгострокова підтримка фіксованих версій, Ubuntu LTS). Linux домінує на серверах: переважна більшість веб-серверів, хмарних інстансів, контейнерів Docker — це Linux. Також — основа Android (мобільні), вбудованих систем, суперкомп'ютерів.

Приклад задачі

Умова. Перевірити версію ядра та дистрибутиву на сервері інформаційної системи чергової частини через стандартні утиліти.

Команди оболонки у файлі commands.sh.

Лістинг 78.1. Реалізація з покроковим розбором

Файл `commands.sh` перевірка версій

```
# Версія ядра Linux
uname -a
# Linux dispatch-srv 5.15.0-100-generic ...

# Інформація про дистрибутив
cat /etc/os-release
```

Команда uname -a виводить ядро, ім'я хоста, версію, архітектуру. Файл /etc/os-release (стандарт freedesktop.org) містить структуровану інформацію про дистрибутив: NAME, VERSION, ID.

```
# Альтернатива для деяких дистрибутивів
lsb_release -a
# Distributor ID: Ubuntu
# Release: 22.04
```

Утиліта lsb_release дає узагальнену інформацію (Linux Standard Base). На сучасних системах /etc/os-release вважається надійнішим джерелом, оскільки доступний завжди.

З досвіду викладачів. *Команди з прикладу доступні без прав адміністратора, тому їх можна спокійно виконувати на будь-якому навчальному сервері.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати команди Linux згідно теми варіанту та задокументувати їх вивід у коментарях.

№	Що зробити
1	Дізнатися версію ядра та дистрибутиву на сервері навчальної лабораторії кафедри.
2	Назвати 3 серверні та 3 десктопні дистрибутиви.
3	Дізнатися аптайм сервера чергової частини через uptime.
4	Перевірити, скільки оперативної пам'яті є та використано (free -h).
5	Описати у коментарях, чому Alpine популярний для Docker.

Питання 79. Два основні компоненти Linux

Теоретична частина

Найчастіша відповідь на цьому рівні: kernel (ядро) та shell (оболонка). Це не повна архітектура, але правильне базове розрізнення [53].

Kernel (ядро) — найнижчий рівень ОС, що працює в привілейованому режимі процесора (kernel mode). Він керує процесами (планувальник), пам'яттю (віртуальна пам'ять, page tables), файловими системами (VFS), мережею (TCP/IP), пристроями (драйвери). Користувацькі програми звертаються до ядра через системні виклики (syscalls).

Shell (оболонка) — програма, що приймає команди користувача та виконує їх. Найпопулярніша — bash (Bourne Again Shell), також zsh, fish, dash. Shell працює в user mode як звичайний процес. Між shell і ядром стоять стандартні бібліотеки (libc, glibc), що надають зручний API над syscalls.

Повніша архітектура: hardware → kernel → system libraries (libc) → user programs (включно з shell, демонами, графічним середовищем). Між ядром і користувацьким простором — чіткий бар'єр захисту: користувацький код не може напряму торкнутися пам'яті ядра. Інколи відповідь дають у термінах kernel + GNU userland (утиліти GNU, що утворюють зручне середовище над «голим» ядром). Звідси історична назва «GNU/Linux», що підкреслює внесок проєкту GNU поряд із ядром.

Приклад задачі

Умова. Продемонструвати взаємодію shell із kernel через системний виклик: команда ls каталогу журналів викликів робить syscall openat та readdir.

Демонстрація через strace у файлі commands.sh.

Лістинг 79.1. Реалізація з покроковим розбором

Файл `commands.sh` shell викликає kernel через syscalls

```
# Запустити ls і подивитися його syscalls
strace -e openat,read,write ls /var/log/dispatch
2>&1 | head
# openat(AT_FDCWD, "/var/log/dispatch",
O_RDONLY...) = 3
# write(1, "calls.log\nalarm.log\n", ...) = 20
```

strace показує системні виклики, що команда ls робить до ядра. openat відкриває каталог, getdents читає вміст, write виводить результат. Усі ці виклики виконуються ядром у привілейованому режимі.

Зверніть увагу. *Утиліта strace може бути не встановлена за замовчуванням. В Ubuntu її встановлюють командою sudo apt install strace.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати команду та задокументувати взаємодію shell і ядра у коментарях.

№	Що зробити
1	Описати, що таке kernel та shell, у коментарях.
2	Виконати strace на cat /etc/hostname та описати syscalls.
3	Перерахувати 5 системних викликів (read, write, open, close, fork).
4	Описати, що таке user mode та kernel mode.
5	Перерахувати 3 популярні shell та порівняти їх.

Питання 80. Абсолютний та відносний шлях. Що таке ".", "..", "~"

Теоретична частина

У файловій системі Linux шлях (path) ідентифікує файл або каталог. Шлях буває абсолютним (починається з кореня /) або відносним (відштовхується від поточного каталогу) [52].

Абсолютний шлях завжди починається з / і однозначний з будь-якої точки системи: /home/user/projects/app.txt. Відносний шлях коротший, але залежить від того, де ви зараз: якщо ви в /home/user, то projects/app.txt вказує на той же файл.

Спеціальні позначення усередині шляху: «.» — поточний каталог (рідко потрібен у середині шляху, частіше — для запуску виконуваних файлів: ./script.sh); «..» — батьківський каталог; кілька «..» можна комбінувати: ../../other/file.txt.

Символ «~» (тильда) — позначення домашнього каталогу поточного користувача (на більшості систем — /home/<username>). Розширюється самою оболонкою перед передачею команді: cd ~/Downloads → cd /home/user/Downloads. Це не системний концепт, а особливість оболонки. Слеш / у Linux завжди розділяє компоненти шляху (на відміну від Windows, де \). Системний корінь — один: усі диски монтуються в /mnt, /media або інші точки. Цим Linux відрізняється від Windows з C:\, D:\.

Приклад задачі

Умова. Продемонструвати на файловій структурі сервера чергової частини різницю між абсолютним та відносним шляхами і використання ".", "..", "~".

Команди оболонки у файлі commands.sh.

Лістинг 80.1. Реалізація з покроковим розбором

Файл `commands.sh` різні види шляхів

```
pwd
# /home/dispatcher/projects

# Абсолютний шлях
ls /home/dispatcher/reports

# Відносний шлях (від поточного каталогу)
ls ../reports
```

pwd показує поточний каталог. Абсолютний шлях /home/dispatcher/reports однозначний. Відносний ../reports піднімається на рівень вгору (з projects) і заходить у reports (каталог донесень) — той самий каталог, але запис коротший.

```
# Тильда — домашній каталог
ls ~/calls
# те саме, що /home/dispatcher/calls

# Поточний каталог
ls .

# Виконати скрипт у поточному каталозі
./backup_calls.sh
```

~ — домашній каталог поточного користувача. «.» — поточний каталог; ls . еквівалентно ls без аргументів. Префікс ./ для запуску скрипта обов'язковий, бо інакше shell шукав би його у PATH (звичайних місцях для виконуваних файлів).

Зверніть увагу. *Пробіл у шляху — часта причина помилок у скриптах. Беріть шляхи в лапки: cd "/home/dispatcher/my reports".*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати команди з різними видами шляхів згідно теми варіанту.

№	Що зробити
1	Перейти у домашній каталог через cd ~ та через cd /home/<user> на сервері лабораторії кафедри.
2	Виконати ls /etc та ls ../etc з різних каталогів.
3	Запустити скрипт ./backup_calls.sh у поточному каталозі.
4	Піднятися на 3 рівні вгору через cd ../../../.
5	Описати у коментарях різницю між «.» та «..».

Питання 81. Як дізнатися шлях до поточного каталогу. Як змінити каталог

Теоретична частина

Команда pwd (print working directory) виводить абсолютний шлях до поточного робочого каталогу. Це базова орієнтаційна команда. Її використовують у скриптах для логування або для побудови абсолютних шляхів на льоту [54].

Команда cd (change directory) змінює поточний каталог. Синтаксис: cd <path>. Без аргументів cd переходить у домашній каталог користувача. cd ~ робить те саме явно. cd .. піднімається на рівень. cd - переходить у попередній каталог (зручно для переключення між двома).

У shell-скриптах поточний каталог критично впливає на роботу. Класична практика — на початку скрипта переходити у каталог самого скрипта: cd "$(dirname "$0")", щоб

відносні шляхи всередині працювали передбачувано. Змінні середовища: $PWD містить поточний каталог (його встановлює та оновлює shell), $OLDPWD — попередній (звідки була остання cd). $HOME — домашній каталог.

У програмах Java поточний робочий каталог отримується через System.getProperty("user.dir"). У Python — os.getcwd(). Це поточний каталог процесу, з якого його запустили, — він не обов'язково збігається з місцем розташування виконуваного файлу.

Приклад задачі

Умова. Продемонструвати pwd та cd під час роботи з каталогами журналів чергової частини: показати поточний каталог, перейти в інший, повернутися назад.

Команди оболонки у файлі commands.sh.

Лістинг 81.1. Реалізація з покроковим розбором

Файл `commands.sh` pwd та cd для навігації

`pwd`
`# /home/dispatcher`
 	
`cd /var/log/dispatch`
`pwd`
`# /var/log/dispatch`
 	
`cd ~/reports`
`pwd`
`# /home/dispatcher/reports`
	pwd показує абсолютний шлях. Після cd /var/log/dispatch ми у каталозі журналів. cd ~/reports переходить у каталог reports у домівці. Кожна cd змінює поточний каталог процесу shell.

`cd -`
`# повертаємось у /var/log/dispatch (попередній)`
 	
`cd`
`# без аргументів — у домівку`
	cd - переходить у попередній каталог, де ми були до останньої cd, і виводить його. cd без аргументів — у домашній каталог (значення $HOME). Це найшвидші команди для переключення.

Порада. *Команда cd без аргументів повертає в домашній каталог з будь-якого місця. Це найшвидший спосіб «знайти себе», якщо загубилися у файловій системі.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати навігаційні команди згідно теми варіанту.

№	Що зробити
1	Перейти у /var/log, потім назад у домашній через cd -.
2	Вивести значення змінних $PWD та $OLDPWD через echo.
3	У Java вивести System.getProperty("user.dir").
4	Створити скрипт резервного копіювання журналів, який виводить свій каталог через dirname "$0".
5	Перейти на 3 рівні вгору через cd ../../.. та повернутися.

Питання 82. Команди ls, mkdir, cp

Теоретична частина

ls (list) — виводить вміст каталогу. Без аргументів показує поточний; з аргументом — вказаний. Корисні ключі: -l (довгий формат: права, власник, розмір, дата), -a (показати приховані файли, що починаються з крапки), -h (зручний для людини розмір: K, M, G), -t (сортувати за датою змінення), -R (рекурсивно) [52].

mkdir (make directory) — створює каталог. Синтаксис: mkdir <name>. Ключ -p створює всі проміжні каталоги (mkdir -p a/b/c створить a, потім a/b, потім a/b/c одразу) і не помиляється, якщо каталог уже існує. Без -p спроба створити вже наявний — помилка.

cp (copy) — копіює файли та каталоги. cp src dst — копіює файл src у dst. cp -r src dst — рекурсивне копіювання каталогу. cp -i — інтерактивно (запитує перед перезаписом), cp -p — зберігає метадані (права, дату), cp -v — verbose (показує, що копіюється).

Усі команди приймають кілька аргументів: cp a b c target/ — копіює три файли в target. Шаблони (glob): cp *.txt backup/ — копіює всі .txt-файли. Shell сам розгортає глоби перед викликом cp. Споріднені команди: mv (move або rename), rm (remove), cat (вивід вмісту файлу), touch (створити порожній файл або оновити дату). Усе разом — базовий інструментарій для роботи з файловою системою.

Приклад задачі

Умова. Продемонструвати створення структури каталогів навчального проєкту курсанта, перегляд її вмісту та копіювання файлу.

Команди оболонки у файлі commands.sh.

Лістинг 82.1. Реалізація з покроковим розбором

Файл `commands.sh` ls, mkdir, cp у поєднанні

```
# Створюємо вкладені каталоги одразу
mkdir -p ldubgd/dispatch/src

# Створюємо порожній файл
touch ldubgd/dispatch/src/Main.java

# Дивимось вміст
ls -la ldubgd/dispatch/src
```

mkdir -p створює всю ієрархію ldubgd → dispatch → src за одну команду; без -p довелося б три mkdir. touch створює порожній файл. ls -la показує довгий формат разом із прихованими файлами (. і ..).

```
# Копіюємо файл
cp ldubgd/dispatch/src/Main.java backup/

# Копіюємо весь каталог рекурсивно
cp -r ldubgd backup/ldubgd-copy
```

Перша cp копіює файл у каталог backup/ (зберігає ім’я). Друга з прапором -r копіює весь каталог ldubgd рекурсивно з усім вмістом — без -r команда відмовиться копіювати каталог.

Зверніть увагу. *Команда cp без попередження перезаписує файл із таким самим ім’ям. Додавайте ключ -i, якщо не впевнені, що в каталозі призначення немає однойменного файлу.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати послідовність команд із ls, mkdir, cp згідно теми варіанту.

№	Що зробити
1	Створити вкладену структуру каталогів reports/2026/10 через mkdir -p.
2	Скопіювати каталог журналів викликів рекурсивно через cp -r.
3	Вивести вміст каталогу у довгому форматі з прихованими файлами (ls -la).
4	Скопіювати усі .log-файли журналів в архівний каталог через glob (cp *.log archive/).
5	Виконати ls -lh та пояснити значення стовпців виводу.

Розділ 12. Серіалізація в Java

Питання 83. Серіалізація та десеріалізація у JSON

Теоретична частина

Серіалізація — процес перетворення Java-об’єкта у формат, придатний для зберігання чи передачі (файл, мережа, БД). Десеріалізація — зворотний процес: відновлення об’єкта з формату. У контексті JSON це означає Java-об’єкт ↔ JSON-рядок [50].

Найпопулярніша бібліотека у Java — Jackson (com.fasterxml.jackson). Її центральний клас ObjectMapper. Альтернативи: Gson (Google), JSON-B (стандарт Jakarta EE), org.json. Базові операції Jackson: writeValueAsString(obj) — об’єкт у JSON-рядок, writeValue(File, obj) — у файл, readValue(json, Class) — десеріалізація. Для колекцій із generic-типом — TypeReference: readValue(json, new TypeReference<List<User>>(){}).

Кастомізація через анотації: @JsonProperty("name") — змінити ім’я поля, @JsonIgnore — пропустити поле, @JsonFormat — формат дати, @JsonInclude(NON_NULL) — не серіалізувати null. У Spring Web Jackson інтегрований автоматично — контролери повертають об’єкти, Spring сам серіалізує у JSON.

Поширені проблеми: циклічні посилання (батько→дитина→батько) — розв’язуються через @JsonManagedReference/@JsonBackReference; lazy-колекції Hibernate, що несподівано тригерять завантаження; різні timezone-формати дат.

Приклад задачі

Умова. Серіалізувати список рятувальників караулу List<Rescuer> у JSON-рядок та десеріалізувати назад через Jackson із використанням TypeReference.

POJO Rescuer та робота з Jackson у двох файлах.

Лістинг 83.1. Реалізація з покроковим розбором

Файл `Rescuer.java` клас даних

```
public class Rescuer {
    private String name;
    private int age;
    public Rescuer() {}
    public Rescuer(String name, int age) {
        this.name = name; this.age = age;
    }
    // геттери/сеттери
}
```

Звичайний POJO з конструктором за замовчуванням (Jackson потребує його для десеріалізації) та публічними методами доступу. Жодних анотацій — Jackson за замовчуванням працює з полями через геттери/сеттери.

Файл JsonDemo.java серіалізація та десеріалізація списку	

`ObjectMapper m = new ObjectMapper();`
`List<Rescuer> users = List.of(`
`  new Rescuer("Коваль", 25),`
`  new Rescuer("Мельник", 30));`
 	
`String json = m.writeValueAsString(users);`
`// [{"name":"Коваль","age":25},`
`{"name":"Мельник","age":30}]`
	ObjectMapper — центральний клас Jackson. writeValueAsString перетворює список об'єктів у JSON-масив. Для серіалізації Jackson достатньо публічних геттерів — анотацій не потрібно.

`List<Rescuer> back = m.readValue(`
`  json,`
`  new TypeReference<List<Rescuer>>(){});`
`System.out.println(back.size());`
	Для десеріалізації generic-типу потрібен TypeReference — інакше через type erasure Jackson не знає, який тип у List. TypeReference зберігає інформацію про тип через анонімний клас. Результат — список з двох об'єктів Rescuer.

Зверніть увагу. *TypeReference виглядає незвично, але без нього Jackson поверне список LinkedHashMap замість списку Rescuer — і помилка проявиться значно пізніше.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати JSON-серіалізацію згідно теми варіанту через Jackson.

№	Що реалізувати
1	Серіалізувати об'єкт FireCall у JSON та записати у файл.
2	Прочитати JSON-файл та десеріалізувати у List<Equipment>.
3	Використати @JsonIgnore для приховування поля password облікового запису диспетчера.
4	Використати @JsonProperty для перейменування поля stationNumber у JSON.
5	Налаштувати pretty-print донесення через writerWithDefaultPrettyPrinter().

Питання 84. Серіалізація та десеріалізація в XML

Теоретична частина

Серіалізація в XML — перетворення об'єкта у структурований XML-документ. У Java є три популярні підходи: JAXB (стандарт), Jackson XML (через ObjectMapper), власні DOM-маніпуляції [48].

JAXB (Jakarta XML Binding) — стандарт серіалізації Java↔XML. До Java 8 входив до JDK; з Java 11 — окрема залежність (jakarta.xml.bind-api). Класи позначаються анотаціями @XmlRootElement, @XmlElement, @XmlAttribute. Центральні класи — Marshaller (об'єкт→XML) і Unmarshaller (XML→об'єкт).

Jackson XML — той самий API, що для JSON, але інша реалізація: XmlMapper замість ObjectMapper. Зручний, якщо проєкт уже використовує Jackson. Анотації Jackson

доповнюються XML-специфічними (@JacksonXmlElementWrapper, @JacksonXmlProperty).

Чисто ручний підхід — DocumentBuilder для парсингу, Transformer для виводу. Гнучкий, але багатослівний. Підходить для нестандартних схем. XML-серіалізація історично переважала у корпоративних додатках і SOAP-сервісах. У сучасних REST-API майже всюди витіснена JSON. Залишається актуальною в SOAP, обміні з legacy-системами, документообігу (ODF, OOXML).

Приклад задачі

Умова. Серіалізувати об’єкт Rescuer у XML через JAXB та десеріалізувати назад.

Клас Rescuer з JAXB-анотаціями; демонстрація — у файлі XmlDemo.java.

Лістинг 84.1. Реалізація з покроковим розбором

Файл `Rescuer.java` POJO з JAXB-анотаціями

```
import jakarta.xml.bind.annotation.*;

@XmlRootElement(name = "rescuer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Rescuer {
    private String name;
    private int age;
    public Rescuer() {}
    public Rescuer(String n, int a) {
        name = n; age = a;
    }
}
```

@XmlRootElement позначає клас як кореневий елемент XML — буде <rescuer>. @XmlAccessorType(FIELD) каже використовувати поля напряму, не геттери. JAXB вимагає публічного конструктора без параметрів.

Файл `XmlDemo.java` Marshaller та Unmarshaller

```
JAXBContext ctx =
JAXBContext.newInstance(Rescuer.class);
Marshaller m = ctx.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
true);
m.marshal(new Rescuer("Коваль", 25), System.out);
//
<rescuer><name>Коваль</name><age>25</age></rescuer>
```

JAXBContext — стартова точка, обчислює метадані класу. Marshaller серіалізує об’єкт; JAXB_FORMATTED_OUTPUT вмикає відступи. marshal записує XML у потік (тут System.out).

```
Unmarshaller u = ctx.createUnmarshaller();
Rescuer back = (Rescuer) u.unmarshal(new
StringReader(
    "<rescuer><name>Мельник</name><age>30</age></res
cuer>"));
System.out.println(back);
```

Unmarshaller робить зворотну операцію. unmarshal приймає Reader, InputStream або File з XML-документом. Повертає Object — потрібен каст. Тут створюємо об’єкт Rescuer з рядка XML.

З досвіду викладачів. *З Java 11 JAXB не входить до JDK. Якщо компілятор не знаходить анотацію @XmlRootElement, додайте до pom.xml залежність jakarta.xml.bind-api.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати XML-серіалізацію згідно теми варіанту.

№	Що реалізувати
1	Серіалізувати FireTruck у XML через JAXB та записати у файл.
2	Прочитати XML-файл та десеріалізувати у List<FireTruck>.
3	Використати @XmlAttribute для серіалізації номера частини як атрибута.
4	Серіалізувати донесення про пожежу через Jackson XmlMapper.
5	Описати у коментарях, коли обирати XML, а коли JSON.

Питання 85. DOM vs SAX

Теоретична частина

DOM (Document Object Model) та SAX (Simple API for XML) — два принципово різні підходи до парсингу XML. DOM-парсер читає весь XML-документ у пам’ять і будує деревоподібну структуру об’єктів (Document → Element → Node). Доступ до будь-якого елемента — через навігацію по дереву (getChildNodes, getElementsByTagName), модифікація — через прямі виклики. Перевага: зручний, дозволяє повторні читання, XPath. Недолік: ВЕСЬ документ у пам’яті — для гігабайтних файлів неприйнятно [3].

SAX-парсер — потоковий (event-based): читає XML послідовно і викликає колбеки (startElement, endElement, characters) на кожну подію. Розробник реалізує ContentHandler і обробляє події. Перевага: майже не використовує пам’ять, обробляє файли будь-якого розміру. Недолік: тільки one-pass, складніше для нелінійних запитів, треба тримати стан вручну.

StAX (Streaming API for XML) — компроміс: потоковий, але pull-based (програма сама запитує наступну подію через cursor або iterator). У більшості випадків зручніший за SAX.

Правило вибору: малі-середні документи з частим доступом до різних елементів — DOM. Великі файли з лінійною обробкою — SAX/StAX. У сучасній практиці часто беруть JAXB/Jackson, що під капотом використовують один з парсерів.

Приклад задачі

Умова. Реалізувати читання XML-файлу з особовим складом караулу обома способами: DOM-парсером та SAX-парсером — і порівняти стиль роботи.

Окремі реалізації у файлах DomDemo.java та SaxDemo.java.

Лістинг 85.1. Реалізація з покроковим розбором

Файл `DomDemo.java` DOM: усе дерево в пам’яті

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

DocumentBuilder b = DocumentBuilderFactory
    .newInstance().newDocumentBuilder();
Document doc = b.parse("rescuers.xml");
NodeList list = doc.getElementsByTagName("name");
for (int i = 0; i < list.getLength(); i++)
    System.out.println(list.item(i).getTextContent
());
```

DOM завантажує весь документ у дерево. Потім getElementsByTagName знаходить усі елементи <name>. Можемо звертатися до будь-якого елемента в будь-якому порядку. Швидко й зручно, але вся структура — у пам’яті.

Файл `SaxDemo.java` SAX: обробка подій

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.DefaultHandler;

SAXParser p =
SAXParserFactory.newInstance().newSAXParser();
p.parse("rescuers.xml", new DefaultHandler() {
    boolean inName;
    public void startElement(String u, String l,
            String q, Attributes a) {
        inName = "name".equals(q);
    }
    public void characters(char[] ch, int s, int
n) {
        if (inName) System.out.println(new
String(ch, s, n));
    }
});
```

SAX викликає колбеки startElement, characters, endElement у міру читання документа. Доводиться тримати стан вручну (поле inName). Жодного дерева в пам'яті — обробка йде «на льоту», що дозволяє читати гігантські файли.

Зверніть увагу. *Обидва парсери дадуть однаковий результат на маленькому файлі. Різниця стане помітною лише на файлах у сотні мегабайтів.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати читання XML згідно теми варіанту.

№	Що реалізувати
1	Прочитати rescuers.xml через DOM і вивести всі прізвища.
2	Прочитати великий журнал викликів calls.xml через SAX і підрахувати кількість елементів.
3	Знайти рятувальника за атрибутом id через DOM.
4	Описати у коментарях, для якого розміру файлу обирати DOM, а для якого SAX.
5	Використати StAX (XMLStreamReader) для лінивого парсингу журналу показів датчиків.

Розділ 13. Логування

Питання 86. Що таке логування

Теоретична частина

Логування (logging) — запис повідомлень про події під час роботи програми у журнал (файл, консоль, мережне сховище). Призначення: діагностика, моніторинг, аудит, відлагодження після факту (post-mortem). Логи дозволяють відповісти на питання «що відбувалося в системі», коли немає прямого доступу до запущеного процесу. На продакшені це часто єдине джерело правди про збої [66].

У Java є кілька стандартів. SLF4J (Simple Logging Facade for Java) — фасадний API: код пишеться один раз, реалізацію (backend) можна підмінити. Реалізації: Logback (де-факто стандарт у нових проєктах), Log4j 2, java.util.logging (вбудований у JDK).

Базова ідея: розробник пише log.info("Запит обробився за {} мс", duration); фасад робить рендеринг повідомлення лише якщо рівень INFO увімкнений. Це дозволяє писати багато логів і вмикати/вимикати їх через конфігурацію без зміни коду. Конфігурація задається у файлі (logback.xml, log4j2.xml): які повідомлення куди писати (console, file, syslog), у якому форматі, з якої категорії. Це називається appender і pattern.

Приклад задачі

Умова. Налаштувати SLF4J з Logback та використати логування у сервісі реєстрації викликів.

Залежності в pom.xml; код у файлі CallService.java; конфігурація у файлі logback.xml.

Лістинг 86.1. Реалізація з покроковим розбором

Файл `CallService.java` виклики логера через SLF4J

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class CallService {
    private static final Logger log =
        LoggerFactory.getLogger(CallService.class)
;
```

Імпортуємо інтерфейси SLF4J. LoggerFactory.getLogger створює (або повертає кешований) логер з категорією — зазвичай ім'я класу. Це стандартний шаблон, повторюється у кожному класі.

```
    public void registerCall(long id) {
        log.info("Виклик {} зареєстровано", id);
        try {
            // ...
        } catch (Exception e) {
            log.error("Помилка обробки {}", id,
e);
        }
    }
}
```

log.info використовує параметризоване повідомлення з {} — заміна на id виконується лише якщо рівень INFO увімкнений (економія CPU). log.error приймає Throwable останнім аргументом — він буде роздрукований зі стеком.

Зверніть увагу. *Не використовуйте System.out.println для журналювання в робочих застосунках: такі повідомлення неможливо відфільтрувати за рівнем, а у файл вони потрапляють лише випадково.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати логування у простому класі згідно теми варіанту.

№	Що реалізувати
1	Налаштувати SLF4J + Logback у Maven-проєкті системи чергової частини.
2	Логувати початок і завершення обробки виклику з тривалістю.
3	Логувати помилку з'єднання з датчиком зі стеком виключення (Throwable).
4	Налаштувати запис журналу чергування у файл через RollingFileAppender.
5	Використати java.util.logging (без зовнішніх залежностей).

Питання 87. Рівні логування

Теоретична частина

Логи у SLF4J/Logback/Log4j мають п'ять стандартних рівнів за зменшенням важливості: ERROR, WARN, INFO, DEBUG, TRACE. Java util logging має дещо іншу шкалу: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST [67].

Семантика рівнів. ERROR — серйозна помилка, що потребує уваги: впав сервіс, недоступна БД, втрачено дані. WARN — щось підозріле, але робота триває: повільний запит, fallback на дефолтні значення, тимчасові проблеми з мережею. INFO — нормальні події бізнес-логіки: створено замовлення, користувач залогінився, запущено сервіс. Кількість INFO має бути помірною — це повідомлення продакшен-моніторингу.

DEBUG — детальна інформація для розробників: вхідні параметри методів, проміжні стани, прийняті рішення. На продакшені зазвичай вимкнено. TRACE — найдрібніша деталізація: вхід у кожен метод, кожна ітерація циклу. Майже ніколи не вмикається на продакшені, лише при цільовому пошуку бага.

Конфігурація задає мінімальний рівень: якщо встановлено INFO, тоді ERROR/WARN/INFO виводяться, DEBUG/TRACE мовчать. На продакшені зазвичай INFO; для аналізу інциденту тимчасово вмикають детальніший рівень DEBUG.

Приклад задачі

Умова. Продемонструвати використання різних рівнів логування в модулі моніторингу пожежної сигналізації та показати, як рівень у конфігурації фільтрує повідомлення.

Код у файлі Demo.java; конфігурація у logback.xml.

Лістинг 87.1. Реалізація з покроковим розбором

Файл `Demo.java` виклики всіх п’яти рівнів

```
log.trace("Опитування датчика params={}", params);
log.debug("Температура датчика: {}", value);
log.info("Виклик {} передано караулу", id);
log.warn("Перевищено час відповіді датчика: {}
мс", ms);
log.error("Не вдалося передати виклик {}", id,
ex);
```

П’ять рівнів від найдрібнішого до найважливішого. TRACE/DEBUG — для розробника, не вмикають на проді. INFO — нормальні бізнес-події. WARN — щось підозріле. ERROR — серйозна проблема з Throwable.

Файл `logback.xml` налаштування рівня

```
<configuration>
    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
    </root>
</configuration>
```

Кореневий рівень INFO: повідомлення ERROR, WARN, INFO виводяться, DEBUG і TRACE мовчать. Під час розслідування інциденту можна тимчасово ввімкнути детальніший рівень DEBUG, перезапустити застосунок (або скористатися hot-reload конфігурації) й отримати детальні журнали.

Порада. *Типова порада: якщо повідомлення знадобиться черговому адміністратору о третій ночі — це INFO або WARN; якщо лише розробнику під час налагодження — DEBUG.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати або налаштувати рівні логування згідно теми варіанту.

№	Що реалізувати
1	Викликати лог усіх п’яти рівнів у сервісі сигналізації та переконатися, які з них видно.
2	Налаштувати logback.xml з рівнем DEBUG лише для пакета ua.ldubgd.dispatch.
3	Описати у коментарях, на якому рівні логувати кожну подію чергової частини (5 прикладів).
4	Динамічно перевірити, чи рівень DEBUG увімкнено (log.isDebugEnabled()).
5	Налаштувати різні рівні для різних appender'ів (console=INFO, file=DEBUG).

Розділ 14. Мережне програмування в Java

Питання 88. Як надіслати HTTP-запит із Java

Теоретична частина

У Java є кілька способів надіслати HTTP-запит. До Java 11 використовували застарілий HttpURLConnection (низькорівневий, незручний) або сторонні бібліотеки. З Java 11 у JDK є сучасний HTTP-клієнт java.net.http.HttpClient, що повністю замінює старі підходи [56].

HttpClient підтримує HTTP/1.1 та HTTP/2, синхронні (send) та асинхронні (sendAsync) виклики, реактивний BodyHandler, повторні спроби. API лаконічний і зручний.

Базовий потік: створити HttpClient (можна перевикористовувати), сформувати HttpRequest через builder, надіслати через client.send(request, BodyHandlers.ofString()), отримати HttpResponse зі статусом, заголовками, тілом.

Популярні зовнішні бібліотеки: Apache HttpClient (зрілий, багато можливостей), OkHttp (Square, легкий, добре для Android), Spring WebClient (реактивний, у Spring-проєктах), Retrofit (декларативний клієнт, побудований на OkHttp). Вибір залежить від проєкту: чистий JDK без залежностей — java.net.http.HttpClient; Spring-додаток — WebClient або RestTemplate (legacy); складні сценарії з кастомним SSL, проксі — Apache HttpClient.

Приклад задачі

Умова. Виконати GET-запит до API інформаційної системи чергової частини через стандартний HttpClient з Java 11+ та вивести тіло відповіді (перелік активних викликів).

Реалізація у файлі HttpDemo.java.

Лістинг 88.1. Реалізація з покроковим розбором

Файл `HttpDemo.java` GET-запит через HttpClient

```
import java.net.URI;
import java.net.http.*;

HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://dispatch.ldubgd.edu.ua/api/c
alls"))
    .GET()
    .header("Accept", "application/json")
    .build();
```

HttpClient.newHttpClient() створює клієнт з налаштуваннями за замовчуванням (HTTP/2, без проксі). HttpRequest будується через builder: URI, метод (GET), заголовки. Об’єкт незмінний — можна повторно використовувати.

```
HttpResponse<String> resp = client.send(
    request,
    HttpResponse.BodyHandlers.ofString());
System.out.println(resp.statusCode());
System.out.println(resp.body());
```

client.send виконує запит синхронно. BodyHandlers.ofString читає тіло як String. HttpResponse містить statusCode (наприклад, 200), headers, body. Для асинхронного виклику — sendAsync, що повертає CompletableFuture.

Зверніть увагу. *Адреса API в прикладі умовна. Для лабораторної роботи можна використати будь-який відкритий API, наприклад сервіс прогнозу погоди.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати HTTP-запит згідно теми варіанту через HttpClient.

№	Що реалізувати
1	GET-запит до публічного API погоди та вивід тіла відповіді (оцінка пожежної небезпеки).
2	POST-запит з JSON-тілом нового виклику через BodyPublishers.ofString.
3	Асинхронне опитування трьох частин через sendAsync та CompletableFuture.
4	Запит з таймаутом 5 секунд через .timeout(Duration.ofSeconds(5)).
5	Прочитати заголовки відповіді через resp.headers().map().

Розділ 15. Веб-сервіси REST і SOAP, контроль версій, збірка, CI

Питання 89. Реалізація REST-клієнта у Java

Теоретична частина

REST-клієнт — компонент, що звертається до зовнішнього REST API (виконує GET/POST/PUT/DELETE), отримує дані у форматі JSON або XML і перетворює їх на Java-об’єкти. Класичний підхід — стандартний HttpClient (Java 11+) у поєднанні з Jackson для десеріалізації JSON. Гнучко, без зайвих залежностей, повний контроль [42; 46].

У Spring-проєктах традиційно використовували RestTemplate (синхронний; починаючи зі Spring 5 переведений у режим підтримки на користь WebClient). WebClient (зі Spring WebFlux) — реактивний, підтримує асинхронні запити, підходить і для блокуючого коду.

Декларативний підхід — Feign (Spring Cloud OpenFeign) або Retrofit: оголошуєте інтерфейс з анотаціями (@GET, @POST, @Path), бібліотека генерує реалізацію автоматично. Це найзручніший варіант для роботи з кількома API.

Типова логіка клієнта: побудувати URL з параметрами, надіслати запит з потрібними заголовками (Authorization, Accept), обробити статус відповіді (200 — успіх, 4xx — помилка клієнта, 5xx — сервера), десеріалізувати тіло у DTO. Не забути про retry/timeout/circuit breaker для надійності.

Приклад задачі

Умова. Реалізувати REST-клієнт, що отримує дані пожежно-рятувальної частини за id з API чергової служби та десеріалізує JSON у Java-об’єкт.

DTO у файлі Station.java; клієнт у файлі StationClient.java.

Лістинг 89.1. Реалізація з покроковим розбором

Файл `Station.java` DTO для десеріалізації

```
public class Station {
    private Long id;
    private String name;
    private String phone;
    public Station() {}
    // геттери/сеттери
}
```

Простий DTO, що повторює структуру JSON-відповіді API. Jackson десеріалізуватиме JSON у такий об’єкт автоматично. Конструктор без параметрів обов’язковий.

Файл `StationClient.java` клієнт із HttpClient + Jackson

```
HttpClient http = HttpClient.newHttpClient();
ObjectMapper json = new ObjectMapper();

HttpRequest req = HttpRequest.newBuilder()
    .uri(URI.create("https://dispatch.ldubgd.edu.ua/api/stat
ions/1"))
    .header("Accept", "application/json")
    .GET()
    .build();
```

Створюємо HttpClient та ObjectMapper (обидва дорогі — варто перевикористовувати). Будуємо GET-запит до endpoint'у з заголовком Accept, що повідомляє серверу про бажаний формат відповіді.

```
HttpResponse<String> resp = http.send(req,
    HttpResponse.BodyHandlers.ofString());
if (resp.statusCode() == 200) {
    Station u = json.readValue(resp.body(), Station.class);
    System.out.println(u.getName());
}
```

Виконуємо запит синхронно. Перевіряємо статус 200 — успіх. Тіло (JSON-рядок) парсимо через Jackson у Java-об'єкт Station. У продакшені варто додати обробку інших статусів (4xx, 5xx) та винятків.

Зверніть увагу. *Перевіряйте статус-код до десеріалізації. Сервер, що повернув 500, зазвичай надсилає в тілі HTML-сторінку помилки, а не очікуваний JSON.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати REST-клієнт згідно теми варіанту.

№	Що реалізувати
1	GET-клієнт, що отримує список частин і десеріалізує у List<Station>.
2	POST-клієнт, що реєструє новий виклик (JSON-тіло).
3	Клієнт API розкладу занять університету з обробкою помилок 4xx та 5xx.
4	Клієнт довідника гідрантів через RestTemplate (Spring).
5	Декларативний клієнт API чергової частини через інтерфейс і Spring OpenFeign.

Питання 90. Реалізація REST-сервісу у Java

Теоретична частина

REST-сервіс (REST-API) — серверний компонент, що приймає HTTP-запити, обробляє їх і повертає дані (зазвичай у JSON). Базується на принципах REST: ресурси, HTTP-методи з семантикою (GET — читання, POST — створення, PUT — заміна, DELETE — видалення), stateless-взаємодія, статус-коди [41; 43].

Найпопулярніший спосіб у Java — Spring Web (через Spring Boot). Контролер позначається @RestController, методи — анотаціями @GetMapping, @PostMapping тощо.

Spring сам серіалізує об’єкти-результати у JSON через Jackson і десеріалізує JSON-тіла запитів у параметри методів.

Стандарт Jakarta EE — JAX-RS (анотації @Path, @GET, @POST). Реалізації: Jersey, RESTEasy. Менш популярний за Spring у нових проєктах, але класично використовується в WildFly, Quarkus.

Базові принципи проектування REST-API: іменники у шляхах (/users/123, не /getUser/123), HTTP-методи передають дію (POST /users — створити), використання статус-кодів (201 Created при створенні, 404 Not Found, 400 Bad Request, 500 Internal Server Error), пагінація для списків (page, size), фільтрація через query-параметри. Сучасні практики: версіонування (/v1/users), HATEOAS (посилання у відповідях), OpenAPI/Swagger для документації, JWT/OAuth для автентифікації, rate limiting, CORS.

Приклад задачі

Умова. Реалізувати простий REST-контролер на Spring Boot для інформаційної системи чергової частини, що повертає список викликів та один виклик за id.

Контролер у файлі CallController.java.

Лістинг 90.1. Реалізація з покроковим розбором

Файл `CallController.java` Spring REST-контролер

```
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api/calls")
public class CallController {
```

@RestController = @Controller + @ResponseBody — Spring автоматично серіалізує повернені об’єкти у JSON. @RequestMapping("/api/calls") задає базовий шлях для всіх методів контролера.

```
    @GetMapping
    public List<FireCall> all() {
        return callService.findAll();
    }

    @GetMapping("/{id}")
    public FireCall byId(@PathVariable Long id) {
        return callService.findById(id);
    }
}
```

@GetMapping без шляху обробляє GET /api/calls — повертає список усіх викликів (Spring перетворить у JSON-масив). @GetMapping("/{id}") з @PathVariable обробляє GET /api/calls/5 — id візьметься зі шляху.

З досвіду викладачів. *Запустіть застосунок і відкрийте у браузері http://localhost:8080/api/calls — Spring поверне JSON без жодного додаткового коду серіалізації.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу реалізувати endpoint REST-сервісу згідно теми варіанту.

№	Що реалізувати
1	@PostMapping для реєстрації нового виклику з @RequestBody.
2	@PutMapping для оновлення статусу виклику за id.
3	@DeleteMapping для видалення помилково створеного виклику з поверненням ResponseEntity та статусом 204.
4	@GetMapping списку курсантів із пагінацією через @RequestParam (page, size).
5	Обробити випадок «частину не знайдено» з поверненням 404.

Питання 91. Що таке SOAP web-сервіс

Теоретична частина

SOAP (Simple Object Access Protocol) — протокол обміну структурованою інформацією через web-сервіси. Базується виключно на XML. Розроблений у 1998 році, був домінуючим стандартом корпоративних інтеграцій у 2000-х. Кожне SOAP-повідомлення — це XML-документ з фіксованою структурою: <Envelope> містить опційний <Header> (метадані, автентифікація, маршрутизація) та обов'язковий <Body> (корисне навантаження або помилка <Fault>) [49].

Контракт сервісу описується у WSDL (Web Services Description Language) — XML-файлі з переліком операцій, типів параметрів, форматом відповідей, точкою доступу (endpoint URL). За WSDL клієнт може автоматично згенерувати proxy-класи (wsimport у Java).

Переваги SOAP: суворо типізований, з контрактом WSDL, багатий стандартний стек (WS-Security, WS-AtomicTransaction, WS-ReliableMessaging) для корпоративних сценаріїв, не залежить від HTTP (може поверх SMTP, JMS).

Недоліки: багатослівний (XML overhead), складніше тестувати ніж REST, вищий поріг входу. У сучасній розробці майже повсюдно витіснений REST/JSON. SOAP залишається в legacy-системах, банкінгу, держустановах, обміні з SAP.

Приклад задачі

Умова. Розглянути приклад SOAP-конверту із запитом до гідравлічного сервісу: обчислити сумарну витрату води двох рукавних ліній (л/с).

Приклади SOAP-повідомлень у файлах request.xml та response.xml.

Лістинг 91.1. Реалізація з покроковим розбором

Файл `request.xml` SOAP-запит

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelop
e/">
  <soap:Body>
    <totalFlow xmlns="http://ldubgd.edu.ua/hydraulics">
      <a>3</a>
      <b>5</b>
    </totalFlow>
  </soap:Body>
</soap:Envelope>
```

Envelope — кореневий елемент кожного SOAP-повідомлення. Body містить виклик операції totalFlow із двома параметрами — витратами двох рукавних ліній. Заголовок Header опущений, бо не потрібен. xmlns задає namespace SOAP.

Файл `response.xml` SOAP-відповідь

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelop
e/">
  <soap:Body>
    <totalFlowResponse
xmlns="http://ldubgd.edu.ua/hydraulics">
      <result>8</result>
    </totalFlowResponse>
  </soap:Body>
</soap:Envelope>
```

Відповідь має ту саму структуру Envelope/Body, але всередині — totalFlowResponse з результатом (8 л/с). У разі помилки тут був би елемент <soap:Fault> з кодом і описом. Усе обов'язково у XML.

Зверніть увагу. *SOAP-повідомлення зручно надсилати за допомогою SoapUI або Postman: обидва інструменти мають безкоштовні версії, достатні для навчання.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу описати або реалізувати елемент SOAP-сервісу згідно теми варіанту.

№	Що реалізувати або описати
1	Описати структуру SOAP-конверта (Envelope, Header, Body).
2	Описати призначення WSDL та основні елементи.
3	Розглянути приклад SOAP Fault при помилці (від'ємна витрата води).
4	Порівняти SOAP і REST у таблиці: формат, контракт, стандарти.
5	Описати, в яких випадках SOAP сьогодні актуальний (міжвідомчий обмін даними державних служб).

Питання 92. Що таке система контролю версій (VCS)

Теоретична частина

Система контролю версій (Version Control System, VCS) — програмне забезпечення, що відстежує зміни у файлах із часом і дозволяє повернутися до будь-якої попередньої версії, побачити історію, узгодити зміни між кількома розробниками. Базові завдання VCS: збереження повної історії змін, ідентифікація автора кожної зміни, паралельна робота кількох розробників, злиття змін, відкат до попередніх станів [68].

Класифікація. Локальні (RCS) — застарілі, лише на одній машині. Централізовані (Subversion, SVN, CVS, Perforce) — один сервер містить історію, клієнти лише workspace. Розподілені (Git, Mercurial) — кожен клієнт має повну копію репозиторію з історією; робота офлайн, локальні коміти.

De facto стандарт сьогодні — Git, створений Лінусом Торвальдсом для розробки ядра Linux у 2005. Усі великі платформи (GitHub, GitLab, Bitbucket) базуються на Git.

Базові концепції Git: репозиторій (.git/ з усією історією), commit (атомарний зріз стану з повідомленням), branch (паралельна лінія розробки), merge/rebase (об'єднання гілок), remote (віддалена копія репозиторію), pull request / merge request (запит на злиття змін).

Приклад задачі

Умова. Створити локальний Git-репозиторій для курсового проєкту курсанта, додати файл, зробити перший коміт.

Послідовність команд у файлі init.sh.

Лістинг 92.1. Реалізація з покроковим розбором

Файл `init.sh` ініціалізація та перший коміт	

`mkdir dispatch-system && cd dispatch-system`
`git init`
`# Initialized empty Git repository in .git/`
 	
`echo "# Dispatch System" > README.md`
`git status`
`# Untracked files: README.md`
	mkdir + cd створюють каталог. git init ініціалізує порожній репозиторій (створюється прихована теку .git/). Створюємо файл і дивимось статус — README.md «untracked», тобто Git його ще не відстежує.

`git add README.md`
`# або git add . для всіх файлів`
 	
`git commit -m "Initial commit"`
`# [master ...] Initial commit`
 	
`git log --oneline`
	git add додає файл у staging area (індекс) — підготовча зона перед комітом. git commit фіксує знімок зі staging як новий коміт у історії. git log показує історію — наш єдиний коміт з повідомленням.

Зверніть увагу. *Повідомлення коміту пишіть так, щоб через пів року ви самі зрозуміли, що змінилося. «fix», «update» чи «123» — погані приклади.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати операції з VCS згідно теми варіанту.

№	Що зробити
1	Ініціалізувати репозиторій лабораторної роботи, додати 3 файли, зробити коміт.
2	Створити .gitignore та переконатися, що вказані файли не відстежуються.
3	Переглянути історію через git log та git log --oneline.
4	Описати у коментарях різницю між централізованим (SVN) та розподіленим (Git) VCS.
5	Налаштувати user.name та user.email (корпоративна пошта ldubgd.edu.ua) через git config.

Питання 93. Базові команди git: add, commit, pull, push

Теоретична частина

git add <files> — додає зміни у staging area (індекс). Це підготовча зона: файли, які підуть у наступний коміт. git add . додає всі змінені файли у поточному каталозі рекурсивно. Файли можна додавати частково (git add -p) [68].

git commit -m "message" — фіксує вміст staging area як новий коміт у локальній історії з повідомленням. Кожен коміт має унікальний SHA-1 хеш, автора, дату. git commit --amend дозволяє змінити останній коміт (повідомлення або вміст). Конвенція повідомлень: коротке (50 символів) перше речення в наказовому способі, далі — порожній рядок і детальний опис.

git pull — забирає зміни з віддаленого репозиторію та зливає їх з локальною гілкою. Фактично = git fetch + git merge. Альтернатива: git pull --rebase замість злиття перебазовує локальні коміти на верхівку віддалених — лінійніша історія.

git push — публікує локальні коміти у віддалений репозиторій. git push origin main надсилає гілку main у remote origin. Якщо віддалена гілка просунулася (хтось push'нув першим), Git відмовить — потрібен pull/rebase перед повторним push. Типовий цикл

роботи: pull → редагувати → add → commit → push. Перед push зазвичай ще раз pull (на випадок змін від колег) і вирішення конфліктів, якщо вони з’явилися.

Приклад задачі

Умова. Виконати типовий цикл роботи з Git над модулем реєстрації викликів: змінити файл, додати, закомітити та відправити у віддалений репозиторій.

Послідовність команд у файлі workflow.sh.

Лістинг 93.1. Реалізація з покроковим розбором

Файл `workflow.sh` повний цикл змін

`git pull origin main`
`# отримуємо останні зміни`
 	
`echo "call registration" >> feature.txt`
`git status`
`# modified: feature.txt`
	Перед роботою — pull, щоб синхронізувати локальну копію з віддаленою. Робимо зміну у файлі. git status показує, що файл модифікований, але ще не у staging area.

`git add feature.txt`
`git commit -m "Add call registration"`
`# [main 7a1b2c3] Add call registration`
 	
`git push origin main`
`# To github.com:ldubgd/dispatch.git`
`#   3f4d5e6..7a1b2c3  main -> main`
	git add переносить файл у staging. commit створює локальний коміт із повідомленням. push надсилає коміт у віддалений репозиторій (origin) у гілку main. Тепер інші можуть забрати ці зміни через pull.

Порада. *Перед git push завжди переглядайте git status і git diff. Це зберігає від випадкового публікування паролів, тимчасових файлів і чужих змін.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати послідовність команд Git згідно теми варіанту.

№	Що зробити
1	Цикл add → commit → push для нового файлу звіту лабораторної роботи.
2	Виконати git pull --rebase та пояснити різницю від git pull.
3	Виправити повідомлення останнього коміту через git commit --amend.
4	Створити .gitignore, додати у нього *.log, перевірити, що .log-файли ігноруються.
5	Скасувати ще не закомічені зміни через git restore (або git checkout).

Питання 94. git stash vs git commit

Теоретична частина

Обидві команди зберігають поточні зміни, але мають різне призначення. git commit фіксує зміни як постійний запис у історії гілки. Коміт має SHA-1 хеш, автора, повідомлення; стає частиною лінійної історії, push'ається у віддалений репозиторій. Це найкращий спосіб зберегти будь-яку завершену або проміжну роботу [68].

git stash тимчасово відкладає зміни у спеціальний «карман» (стек), повертаючи робочу копію до останнього коміту. Зміни не потрапляють у історію, не push'аться. Корисно, коли треба тимчасово переключитися на іншу задачу, але поточні зміни ще не готові до коміту.

Команди stash: git stash (зберегти зміни і очистити робочу копію), git stash list (список збережених), git stash pop (відновити останнє і видалити зі стеку), git stash apply (відновити, але залишити у стеку), git stash drop (видалити зі стеку).

Типовий сценарій: працюєш над фічею, прийшла термінова баг-фіксація → git stash → переключитися на main → виправити баг → закомітити → повернутися у фічу-гілку → git stash pop. Альтернатива — створити «WIP commit» (Work In Progress), потім git reset --soft HEAD~1 для розпакування.

Приклад задачі

Умова. Виконати типовий сценарій: відкласти незавершені зміни модуля карти гідрантів через stash, переключитися на терміновий баг у реєстрації викликів, потім повернути зміни.

Послідовність команд у файлі stash.sh.

Лістинг 94.1. Реалізація з покроковим розбором

Файл `stash.sh` відкладення та відновлення

```
# працюємо над фічею
echo "WIP" > feature.txt
git status
# modified: feature.txt

# терміновий баг у реєстрації викликів
git stash
# Saved working directory and index state WIP on
main
```

Маємо незавершену роботу у feature.txt. Хочемо переключитися на іншу гілку, але не хочемо комітити сире. git stash зберігає зміни у стек і повертає робочу копію до чистого стану (останній коміт).

```
# виправляємо баг
git checkout -b hotfix
# ... зміни, commit, push, merge ...

git checkout main
git stash pop
# зміни повернулися у feature.txt
```

Працюємо у hotfix-гілці, фіксимо баг. Повертаємось у main. git stash pop повертає відкладені зміни і видаляє запис зі стеку — продовжуємо роботу над фічею з місця, де зупинилися.

Зверніть увагу. *Не накопичуйте десятки записів у stash. Через тиждень ви вже не згадаєте, що в кожному з них, — краще створіть окрему гілку.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати операцію зі stash або commit згідно теми варіанту.

№	Що зробити
1	Зберегти зміни через stash і відновити через pop.
2	Виконати кілька stash підряд та переглянути список через git stash list.
3	Створити stash з повідомленням: git stash push -m "WIP hydrant map".
4	Видалити конкретний stash через git stash drop stash@{1}.
5	Описати у коментарях, коли краще stash, а коли WIP commit.

Питання 95. Що таке системи збірки (Project Build Tools)

Теоретична частина

Системи збірки — інструменти, що автоматизують компіляцію, тестування, пакування та доставку програмних проєктів. Замість ручних команд компілятора розробник запускає одну команду, яка виконує всі етапи в потрібному порядку [69; 70].

Завдання build tool: керування залежностями (автоматичне завантаження бібліотек із віддалених репозиторіїв), компіляція коду, виконання тестів, пакування у JAR/WAR/EAR, публікація артефактів, генерація документації, інтеграція з IDE та CI.

У світі Java три основні build tools. Apache Maven (з 2004) — декларативний, конфігурація через XML (pom.xml), широка екосистема, конвенція над конфігурацією. Найпоширеніший у корпоративних Java-проєктах.

Gradle (з 2007) — гнучкіший, конфігурація через DSL (Groovy або Kotlin). Швидший за Maven завдяки кешуванню та інкрементальній збірці. Стандарт для Android. Популярний у сучасних JVM-проєктах.

Apache Ant (з 2000) — старіший, імперативний, без керування залежностями (доповнюється Ivy). Сьогодні рідко використовується для нових проєктів, лишається в legacy. Поза JVM: npm/yarn/pnpm (JavaScript), Cargo (Rust), Go modules, pip/poetry (Python), Bundler (Ruby). Концепції схожі: маніфест залежностей, центральний репозиторій, віртуальне середовище.

Приклад задачі

Умова. Розглянути типову структуру Maven-проєкту інформаційної системи чергової частини з мінімальним pom.xml та командою збірки.

Маніфест pom.xml та структура проєкту.

Лістинг 95.1. Реалізація з покроковим розбором

Файл `pom.xml` мінімальний маніфест Maven

```
<project
xmlns="http://maven.apache.org/POM/4.0.0">
    <modelVersion>4.0.0</modelVersion>
    <groupId>ua.ldubgd</groupId>
    <artifactId>dispatch</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>2.0.7</version>
        </dependency>
    </dependencies>
</project>
```

Maven-маніфест задає координати проєкту (groupId, artifactId, version) — це адреса в Maven Central. Секція dependencies перелічує бібліотеки, які Maven автоматично завантажить. packaging задає тип артефакту.

Зверніть увагу. *Координати groupId зазвичай записують як «перевернуте» доменне ім'я організації: для ЛДУБЖД це ua.ldubgd.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу описати або налаштувати елемент системи збірки згідно теми варіанту.

№	Що зробити
1	Створити мінімальний pom.xml для JAR-проєкту обліку спорядження.
2	Описати порівняння Maven vs Gradle у таблиці.
3	Перерахувати у коментарях типові залежності Java-проєкту системи чергової частини.
4	Описати призначення Maven Central та локального кешу ~/.m2.
5	Створити простий build.gradle для аналогічного проєкту.

Питання 96. Maven. Різниця mvn install та mvn package

Теоретична частина

Maven організовує збірку як послідовність фаз життєвого циклу (build lifecycle). Основний цикл (default) має 23 фази, з яких ключові: validate, compile, test, package, verify, install, deploy. Виконання будь-якої фази запускає всі попередні [69].

mvn compile — компілює основний код (src/main/java) у target/classes. Тести не виконуються. mvn test — компілює тести (src/test/java) і запускає їх. Якщо тести не пройдуть — збірка падає (можна обійти прапором -DskipTests або більш агресивним -Dmaven.test.skip=true).

mvn package — пакує проєкт у JAR/WAR відповідно до <packaging> у pom.xml. Артефакт з'являється у target/. Перед цим автоматично виконуються compile та test. Після package є артефакт, але він НЕ доступний іншим проєктам у системі.

mvn install — повне виконання package, плюс копіювання артефакту у локальний Maven-репозиторій (~/.m2/repository). Тепер інші локальні проєкти можуть посилатися на нього як на залежність. Це принципова відмінність: install робить артефакт доступним для інших проєктів на цій машині.

mvn deploy — крок далі: publish у віддалений Maven-репозиторій (Nexus, Artifactory, Maven Central). Доступний усім, хто має доступ до репозиторію. Зазвичай запускається з CI.

Приклад задачі

Умова. Виконати типову збірку проєкту системи чергової частини: компіляція, тести, пакування, встановлення в локальний репозиторій.

Послідовність команд Maven у файлі commands.sh.

Лістинг 96.1. Реалізація з покроковим розбором

Файл `commands.sh` ключові фази Maven	

`mvn clean`
`# видаляє target/`
 	
`mvn compile`
`# компілює код у target/classes`
 	
`mvn test`
`# виконує тести; падає при помилці`
	clean видаляє попередні артефакти (свіжий старт). compile компілює основний код. test запускає JUnit/TestNG — якщо тести «червоні», збірка падає, наступні фази не виконуються.

`mvn package`
`# створює target/dispatch-1.0.0.jar`
`# (виконуються compile + test)`
 	
`mvn install`
`# package + копіює у ~/.m2/repository`
`# тепер інші локальні проєкти можуть залежати`
	package створює артефакт у target/. install робить ще один крок — копіює у локальний Maven-кеш, після чого артефакт доступний як залежність для інших проєктів на цій машині. Це головна різниця двох команд.

З досвіду викладачів. *Ключ -DskipTests зручний, але звичка запускати збірку без тестів рано чи пізно призводить до того, що зламаний код потрапляє в репозиторій.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу виконати команду Maven згідно теми варіанту та задокументувати її роботу.

№	Що зробити
1	Виконати mvn clean install для проєкту обліку курсантів та описати кожну фазу у коментарях.
2	Запустити збірку без тестів через mvn package -DskipTests.
3	Виконати mvn dependency:tree та проаналізувати дерево залежностей.
4	Створити uber-JAR через maven-shade-plugin.
5	Описати у коментарях різницю install vs deploy.

Питання 97. Що таке Continuous Integration. Jenkins та подібні системи

Теоретична частина

Continuous Integration (CI, безперервна інтеграція) — практика розробки, при якій команда часто (декілька разів на день) інтегрує зміни в спільну гілку. Кожна інтеграція автоматично запускає збірку та тести, що швидко виявляє проблеми. Мета CI — уникнути «integration hell»: ситуації, коли розробники тижнями працюють на ізольованих гілках, а при злитті отримують тисячі конфліктів. Часта інтеграція + автоматизована перевірка тримають кодову базу у працездатному стані [71; 72; 73; 74].

Типовий CI-pipeline: розробник push'ить зміни → CI-сервер відстежує push (webhook) → клонує репозиторій → збирає → запускає тести → запускає аналіз якості (SonarQube, лінтери) → формує артефакт → надсилає звіт. При помилці на будь-якому етапі — сповіщення команди.

CD (Continuous Delivery / Deployment) — наступний крок: артефакт, що пройшов CI, автоматично розгортається у тестове, staging, навіть продакшен-середовище. Delivery —

готовність до релізу одним кліком; Deployment — повна автоматизація без ручного схвалення.

Популярні CI-системи. Jenkins — open-source, найгнучкіший, багато плагінів, потребує власного сервера. GitHub Actions — інтегрований у GitHub, опис у .github/workflows/*.yml. GitLab CI — для GitLab, через .gitlab-ci.yml. CircleCI, Travis CI — хмарні. TeamCity (JetBrains), Bamboo (Atlassian) — комерційні. Студентам і курсантам, які лише починають, ми радимо почати з GitHub Actions: його можна спробувати безкоштовно на власному навчальному репозиторії вже сьогодні.

Приклад задачі

Умова. Скласти простий CI-pipeline для Java-проєкту системи чергової частини ЛДУБЖД: збірка, тести, формування артефакту через GitHub Actions.

Конфігурація у файлі .github/workflows/ci.yml.

Лістинг 97.1. Реалізація з покроковим розбором

Файл `.github/workflows/ci.yml` GitHub Actions workflow

```yaml
name: CI
on:
  push:
    branches: [main]
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with:
          java-version: '21'
          distribution: 'temurin'
```

Workflow запускається на push у main та на кожен pull request. Виконується у Ubuntu-контейнері. Кроки: завантажити код (checkout), встановити JDK 21 (Temurin — безкоштовний дистрибутив від Eclipse Adoptium).

```yaml
      - name: Build with Maven
        run: mvn -B package --file pom.xml
      - name: Run tests
        run: mvn test
      - uses: actions/upload-artifact@v4
        with:
          name: dispatch-jar
          path: target/*.jar
```

mvn -B package збирає проєкт у batch-режимі. mvn test запускає тести (хоча package їх уже запустило — тут для наочності). upload-artifact зберігає JAR як артефакт workflow — його можна завантажити у вкладці Actions.

Зверніть увагу. *Додайте у README репозиторію значок статусу збірки (badge) з GitHub Actions. Це дрібниця, але викладачі й роботодавці її помічають.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу скласти або описати CI-конфігурацію згідно теми варіанту.

№	Що зробити
1	Скласти GitHub Actions workflow для збірки курсового Java-проєкту.
2	Описати у коментарях етапи типового CI/CD-pipeline.
3	Скласти простий Jenkinsfile (Declarative Pipeline) для системи обліку спорядження.
4	Описати порівняння CI vs CD vs Continuous Deployment.
5	Перерахувати 5 популярних CI-систем та коротко описати кожну.

ВИСНОВКИ

Запропонований збірник задач об’єднав у собі дев’яносто сім тематичних питань, що покривають базові розділи мови програмування Java та суміжних технологій, які становлять фахове ядро професійних компетентностей розробника-початківця. Структурування матеріалу за чотирма частинами та п’ятнадцятьма тематичними розділами забезпечує послідовне формування знань — від синтаксису мови та об’єктно-орієнтованих конструкцій до веб-розробки, баз даних, систем контролю версій та неперервної інтеграції.

Запропонований формат викладу матеріалу — поєднання розширеної теоретичної довідки, повністю розібраного прикладу задачі та п’яти варіантів для самостійної роботи — дозволяє забезпечити ефективну самостійну підготовку студентів і курсантів. Двостовпцеве оформлення лістингів з покроковим розбором та чітке зазначення приналежності кожного фрагмента коду до конкретного файлу полегшує сприйняття матеріалу здобувачами початкового рівня та сприяє формуванню правильного уявлення про структуру Java-проєктів, а наскрізна предметна область цивільного захисту пов’язує програмування з майбутньою професійною діяльністю. За результатами опрацювання всіх дев’яноста семи питань збірника здобувач формує комплекс компетентностей, необхідних для впевненого проходження технічної співбесіди на позицію Java Trainee або Junior Software Engineer. Зокрема, формуються здатність до аналізу та структурованого пояснення технічних рішень, навички алгоритмічного мислення, розуміння принципів об’єктно-орієнтованого програмування, навички роботи з реляційними базами даних та інструментами веб-розробки, розуміння процесів збірки, тестування та доставки програмного забезпечення.

Подальший розвиток матеріалу збірника може передбачати розширення тематики до рівня Intermediate, включення розділів, присвячених фреймворку Spring та шаблонам проєктування, додавання розгорнутих практичних кейсів інтеграції з зовнішніми сервісами та хмарними платформами, а також формування додатків з контрольними тестовими завданнями для самостійної самоперевірки рівня знань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bloch J. Effective Java. 3rd ed. Boston : Addison-Wesley, 2018. 412 p.
2. Horstmann C. S. Core Java. Volume I — Fundamentals. 12th ed. Boston : Addison-Wesley, 2021. 928 p.
3. Horstmann C. S. Core Java. Volume II — Advanced Features. 12th ed. Boston : Addison-Wesley, 2022. 944 p.
4. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Boston : Addison-Wesley, 2006. 384 p.
5. Eckel B. On Java 8. MindView LLC, 2017. 1130 p.
6. Sierra K., Bates B., Gee T. Head First Java. 3rd ed. Sebastopol : O'Reilly Media, 2022. 752 p.
7. Schildt H. Java: The Complete Reference. 12th ed. New York : McGraw Hill, 2021. 1280 p.
8. Evans B. J., Clark J., Flanagan D. Java in a Nutshell. 8th ed. Sebastopol : O'Reilly Media, 2023. 480 p.
9. Urma R.-G., Fusco M., Mycroft A. Modern Java in Action: Lambdas, Streams, Functional and Reactive Programming. Shelter Island : Manning Publications, 2018. 592 p.
10. Bloch J., Gafter N. Java Puzzlers: Traps, Pitfalls, and Corner Cases. Boston : Addison-Wesley, 2005. 312 p.
11. Naftalin M., Wadler P. Java Generics and Collections. Sebastopol : O'Reilly Media, 2006. 294 p.
12. Warburton R. Java 8 Lambdas: Pragmatic Functional Programming. Sebastopol : O'Reilly Media, 2014. 182 p.
13. Oaks S. Java Performance: In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond. 2nd ed. Sebastopol : O'Reilly Media, 2020. 450 p.
14. Gosling J., Joy B., Steele G., Bracha G., Buckley A., Smith D., Bierman G. The Java Language Specification. Java SE 21 Edition. URL: https://docs.oracle.com/javase/specs/jls/se21/html/index.html (дата звернення: 21.10.2026).
15. Lindholm T., Yellin F., Bracha G., Buckley A., Smith D. The Java Virtual Machine Specification. Java SE 21 Edition. URL: https://docs.oracle.com/javase/specs/jvms/se21/html/index.html (дата звернення: 21.10.2026).
16. Java Platform, Standard Edition 21 API Specification. URL: https://docs.oracle.com/en/java/javase/21/docs/api/ (дата звернення: 21.10.2026).
17. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston : Addison-Wesley, 1994. 395 p.
18. Freeman E., Robson E. Head First Design Patterns. 2nd ed. Sebastopol : O'Reilly Media, 2020. 672 p.
19. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River : Prentice Hall, 2008. 464 p.
20. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.
21. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Boston : Addison-Wesley, 2018. 448 p.
22. Beck K. Test-Driven Development: By Example. Boston : Addison-Wesley, 2002. 240 p.
23. JUnit 5 User Guide. URL: https://junit.org/junit5/docs/current/user-guide/ (дата звернення: 21.10.2026).
24. Date C. J. An Introduction to Database Systems. 8th ed. Boston : Addison-Wesley, 2003. 1024 p.
25. Silberschatz A., Korth H. F., Sudarshan S. Database System Concepts. 7th ed. New York : McGraw-Hill, 2019. 1376 p.
26. Garcia-Molina H., Ullman J. D., Widom J. Database Systems: The Complete Book. 2nd ed. Upper Saddle River : Pearson, 2008. 1248 p.
27. Beaulieu A. Learning SQL. 3rd ed. Sebastopol : O'Reilly Media, 2020. 388 p.
28. Karwin B. SQL Antipatterns: Avoiding the Pitfalls of Database Programming. Raleigh : Pragmatic Bookshelf, 2010. 352 p.
29. Schwartz B., Zaitsev P., Tkachenko V. High Performance MySQL. 3rd ed. Sebastopol : O'Reilly Media, 2012. 826 p.
30. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O'Reilly Media, 2017. 616 p.
31. MySQL 8.0 Reference Manual. URL: https://dev.mysql.com/doc/refman/8.0/en/ (дата звернення: 21.10.2026).

32. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 21.10.2026).
33. JDBC Database Access. The Java Tutorials. URL: https://docs.oracle.com/javase/tutorial/jdbc/ (дата звернення: 21.10.2026).
34. Bauer C., King G., Gregory G. Java Persistence with Hibernate. 2nd ed. Shelter Island : Manning Publications, 2015. 608 p.
35. Mihalcea V. High-Performance Java Persistence. Cluj-Napoca : Vlad Mihalcea, 2016. 470 p.
36. Hibernate ORM User Guide. URL: https://hibernate.org/orm/documentation/ (дата звернення: 21.10.2026).
37. Jakarta Persistence Specification. URL: https://jakarta.ee/specifications/persistence/ (дата звернення: 21.10.2026).
38. Jakarta Servlet Specification. URL: https://jakarta.ee/specifications/servlet/ (дата звернення: 21.10.2026).
39. Jakarta EE Tutorial. URL: https://eclipse-ee4j.github.io/jakartaee-tutorial/ (дата звернення: 21.10.2026).
40. Walls C. Spring in Action. 6th ed. Shelter Island : Manning Publications, 2022. 520 p.
41. Spring Framework Reference Documentation. URL: https://docs.spring.io/spring-framework/reference/ (дата звернення: 21.10.2026).
42. Spring Boot Reference Documentation. URL: https://docs.spring.io/spring-boot/ (дата звернення: 21.10.2026).
43. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Doctoral dissertation. Irvine : University of California, 2000. 162 p.
44. Fielding R., Nottingham M., Reschke J. RFC 9110. HTTP Semantics. 2022. URL: https://www.rfc-editor.org/rfc/rfc9110 (дата звернення: 21.10.2026).
45. Gourley D., Totty B. HTTP: The Definitive Guide. Sebastopol : O'Reilly Media, 2002. 656 p.
46. Richardson L., Amundsen M., Ruby S. RESTful Web APIs. Sebastopol : O'Reilly Media, 2013. 406 p.
47. Bray T. RFC 8259. The JavaScript Object Notation (JSON) Data Interchange Format. 2017. URL: https://www.rfc-editor.org/rfc/rfc8259 (дата звернення: 21.10.2026).
48. Extensible Markup Language (XML) 1.0 (Fifth Edition) : W3C Recommendation. 2008. URL: https://www.w3.org/TR/xml/ (дата звернення: 21.10.2026).
49. SOAP Version 1.2 Part 1: Messaging Framework : W3C Recommendation. 2007. URL: https://www.w3.org/TR/soap12-part1/ (дата звернення: 21.10.2026).
50. Jackson Project Documentation. URL: https://github.com/FasterXML/jackson-docs (дата звернення: 21.10.2026).
51. Tanenbaum A. S., Bos H. Modern Operating Systems. 4th ed. Boston : Pearson, 2014. 1136 p.
52. Shotts W. The Linux Command Line. 2nd ed. San Francisco : No Starch Press, 2019. 504 p.
53. Kerrisk M. The Linux Programming Interface. San Francisco : No Starch Press, 2010. 1552 p.
54. Robbins A., Beebe N. H. F. Classic Shell Scripting. Sebastopol : O'Reilly Media, 2005. 560 p.
55. Tanenbaum A. S., Wetherall D. J. Computer Networks. 5th ed. Boston : Pearson, 2011. 960 p.
56. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. 8th ed. Hoboken : Pearson, 2021. 800 p.
57. Stallings W. Cryptography and Network Security: Principles and Practice. 8th ed. Hoboken : Pearson, 2020. 832 p.
58. Anderson R. Security Engineering: A Guide to Building Dependable Distributed Systems. 3rd ed. Indianapolis : Wiley, 2020. 1232 p.
59. OWASP Top Ten. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 21.10.2026).
60. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. Cambridge : MIT Press, 2022. 1312 p.
61. Sedgewick R., Wayne K. Algorithms. 4th ed. Boston : Addison-Wesley, 2011. 992 p.
62. Knuth D. E. The Art of Computer Programming. Vol. 1 : Fundamental Algorithms. 3rd ed. Reading : Addison-Wesley, 1997. 672 p.
63. Skiena S. S. The Algorithm Design Manual. 3rd ed. Cham : Springer, 2020. 793 p.
64. Bhargava A. Grokking Algorithms. Shelter Island : Manning Publications, 2016. 256 p.
65. McDowell G. L. Cracking the Coding Interview: 189 Programming Questions and Solutions. 6th ed. Palo Alto : CareerCup, 2015. 687 p.
66. SLF4J User Manual. URL: https://www.slf4j.org/manual.html (дата звернення: 21.10.2026).

67. Logback Documentation. URL: https://logback.qos.ch/documentation.html (дата звернення: 21.10.2026).
68. Chacon S., Straub B. Pro Git. 2nd ed. New York : Apress, 2014. 458 p. URL: https://git-scm.com/book/en/v2 (дата звернення: 21.10.2026).
69. Apache Maven Documentation. URL: https://maven.apache.org/guides/ (дата звернення: 21.10.2026).
70. Gradle User Manual. URL: https://docs.gradle.org/current/userguide/userguide.html (дата звернення: 21.10.2026).
71. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston : Addison-Wesley, 2010. 512 p.
72. Kim G., Humble J., Debois P., Willis J. The DevOps Handbook. Portland : IT Revolution Press, 2016. 480 p.
73. GitHub Actions Documentation. URL: https://docs.github.com/en/actions (дата звернення: 21.10.2026).
74. Jenkins User Documentation. URL: https://www.jenkins.io/doc/ (дата звернення: 21.10.2026).
75. Anthropic. Claude : документація моделей та API. URL: https://docs.anthropic.com/ (дата звернення: 21.10.2026).
76. Anthropic. Claude Code : документація інструменту штучного інтелекту для розробки програмного забезпечення. URL: https://docs.anthropic.com/en/docs/claude-code/overview (дата звернення: 21.10.2026).
77. Літнарович Р. М. Об’єктно-орієнтоване програмування мовою Java : навч. посібник. Рівне : МЕГУ, 2018. 220 с.
78. Жуковський С. С., Войченко О. П. Програмування на мові Java : навч. посібник. Житомир : ЖДУ ім. Івана Франка, 2021. 184 с.
79. Глибовець М. М., Олецький О. В. Штучний інтелект : підручник. Київ : Києво-Могилянська академія, 2002. 366 с.
80. Про вищу освіту : Закон України від 01.07.2014 № 1556-VII. URL: https://zakon.rada.gov.ua/laws/show/1556-18 (дата звернення: 21.10.2026).
81. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. Київ : ДП «УкрНДНЦ», 2016. 16 с.