Міністерство науки і освіти України
Донецький національний університет імені Василя Стуса
Фізико-технічний факультет
Кафедра комп'ютерних технологій

Кадомський К.К., Ніколюк П.К.

JAVA.ТЕОРІЯ І ПРАКТИКА
Навчальний посібник

Вінниця – 2019

УДК 004.43(075.8)
К13
Автори
К.К. Кадомський, старший викладач кафедри комп'ютерних технологій
П.К. Ніколюк, професор кафедри комп'ютерних технологій

Затверджено
на засіданні Вченої ради фізико-технічного факультету
Донецького національного університету імені Василя Стуса
(протокол №11 від 15.06.2018 р.)
Кадомський К.К., Ніколюк П.К.
Java. Теорія і практика: навчальний посібник для студентів природничих спеціальностей університетів / Кадомський К.К., Ніколюк П.К. – Вінниця: Донну, 2019. – 197 с.

Навчальний посібник призначений для студентів природничих спеціальностей університетів при вивченні курсу «Java-програмування».

У цьому курсі студенти знайомляться із основами технології Java, можливостями мови програмування Java, а також із інструментальними засобами Integrated Development Environment (IDE) IntelliJ IDEA для розробки програмних продуктів, що відповідають сучасним вимогам. Розглядаються засоби Java для роботи із структурами даних, роботи із файловою системою, створення розвиненого GUI (Graphical User Interface). Також студенти навчаються застосовувати об'єктно-орієнтований підхід при проектуванні складних програмних систем.

Для початку даного курсу бажано мати базові навички програмування: ознайомитись з курсами «Основи програмування» та «Об'єктно-орієнтоване програмування».

Освоєння цього курсу є передумовою для більш спеціалізованого курсу Java, в якому розглядаються технології мультипотокового програмування, засоби синхронізації потоків, технології роботи з базами даних, технології створенням клієнт-серверних додатків, а також додатків для операційної системи (ОС) Android та інші.

Зміст

Вступ

Java є найбільш потужною і активно зростаючою платформою розробки сучасних програмних продуктів, особливо в області web-технологій і додатків корпоративного рівня. Мова займає провідні позиції як в рейтингах популярності мов програмування, так і в рейтингах заробітних плат розробників.

Це великою мірою зумовлено тим, що на ринку програмного забезпечення найбільш затребуваними є додатки рівня підприємства (enterprise-додатки), які автоматизують різні аспекти діяльності підприємств, організацій, спільнот, наприклад документообіг, взаємодія з клієнтами та ін. Вимоги до таких додатків постійно зростають, і, як наслідок, зберігається тенденція до ускладнення концепцій розробки. Це призводить до появи великої кількості нових технологій, в тому числі пов'язаних зі створенням Rich Internet Applications (RIA), розподілених і хмарних додатків і сервісів, додатків для мобільних платформ (наприклад, Android) і багатьох інших. Платформа Java об'єднує цілий пласт технологій програмної інженерії, необхідних для вирішення таких задач. Весь цей пласт неможливо викласти в одному курсі, а повний опис можливостей Java складе цілу бібліотеку. В даному курсі закладаються основи професійних навичок роботи із базовими складовими технології Java. Вивчивши цей курс, ви зможете створювати desktop-додатки на Java, проте для тих, хто в подальшому вибере для себе професію Java-розробника, це тільки перший крок на шляху освоєння всіх можливостей Java.

На протязі п’яти років цей курс лекцій пропонується студентам спеціальності «Комп’ютерні науки» фізико-технічного факультету Донецького національного університету імені Василя Стуса.

Вивчення Java починається з синтаксису мови. Далі розглядається об'єктна модель Java та використання принципів об’єктно-орієнтованого програмування (ООП) при проектуванні складних програмних систем. Паралельно розглядаються прийоми роботи з інструментальними засобами Java-розробки – набором розробника Java Development Kit (JDK) і інтегрованим середовищем розробки IntelliJ IDEA. Розглядаються як загальні принципи розробки сучасних програмних продуктів, так і окремі технології: для роботи зі структурами даних, роботи з файловою системою, створення розвиненого GUI, організації багатопотокових рішень та інші.

Отримані в цьому курсі знання та навички є базою для подальшого більш професійного опанування спеціалізації Java-розробника, зокрема технологій роботи з базами даних (JDBC, JPA, Hibernate), створення клієнт-серверних додатків, додатків для Android, а також зі стеком веб-технологій J2EE і сучасними фреймворками (Spring, Maven, JUnit та ін.).

Тема 1. Огляд технологій JAVA і IDE

Лекція 1. Технологія Java

Java (вимовляється «Джава») – одна із найбільш затребуваних на сьогоднішній день мов програмування, і одночасно – це стек технологій, які покривають весь процес створення сучасних додатків будь-якого спрямування та рівня складності.

Мова Java була розроблена компанією Sun Microsystems (перша офіційна версія Java 1.0 була випущена 26 серпня 1996 року). З 2010 року Java розробляється і підтримується компанією Oracle. Автором мови вважається співробітник компанії Sun Microsystems Джеймс Гослінг (James Gosling), який розробив її основи, створюючи мову для програмування побутових електронних пристроїв. Ця мова спочатку називалась Oak («Дуб»), але згодом стала універсальною мовою програмування та була перейменована в Java. Вважається, що сьогоднішня назва мови і технології Java походить від назви сорту кави.

Межі технології Java досить розмиті і весь час розширюються. Спочатку Java-технологія призначалася для програмування побутової електроніки і телефонів. Потім Java стала виконуватися в інтернет-браузерах – з'явилися *аплети*. Потім виявилося, що на Java можна створювати повноцінні програми. Їх графічні елементи стали оформляти у вигляді компонентів *JavaBeans*, з якими Java увійшла в світ розподілених систем і проміжного програмного забезпечення.

Наступним кроком стало програмування серверів – з'явилися *сервлети* і розподілені компоненти *Enterprise Java Beans (EJB)*. Сервери повинні взаємодіяти з базами даних – тож з'явилась технологія *JDBC* (*Java DataBase Connectivity*). Перераховані технології виявилися вдалими і багато систем управління базами даних і навіть операційні системи включили Java в своє ядро, наприклад Oracle, Linux, MacOS X, AIX.

Таке швидке і широке поширення технології Java не в останню чергу пов'язано з тим, що вона використовує нову, спеціально створену мову програмування, яка так і називається – мова Java. Ця мова створена на базі мов Smalltalk, Pascal, C++ та ін., увібравши їх кращі, на думку творців, риси і відкинувши гірші. На цей рахунок є різні думки, але безперечно, що мова вийшла зручною для вивчення, написані на ній програми легко читати і відлагоджувати.

Чому саме Java?

Java працює на 3 мільярдах пристроїв по всьому світу – це ПК, мобільна електроніка, пристрої що носяться, бортові системи, мережеве обладнання, промислове програмоване обладнання.

На сьогоднішній день Java лідирує серед платформ і мов програмування (на даний момент щільно конкуруючи з PHP, Python і C/C++). Вона займає першу позицію як серед пропозицій на ринку праці IT-індустрії, так і в рейтингу вподобань розробників (рис. 1.1, 1.2).

Рейтинг мови програмування не дає відповіді на запитання, яка мова краща. Цей показник визначає популярність мови програмування. Взагалі існує кілька рейтингів, що оцінюють популярність мов програмування. Серед них – TIOBE Software, Red Monk, PYRL, IEEE Spectrum. Які висновки можна зробити, аналізуючи ці рейтинги? Незважаючи на деякі відмінності в розподілі місць мов, лідерами є Java, JavaScript, Python, C. Саме ці чотири мови займають близько 60% всього обсягу використання мов програмування в світі. Тому, зупинивши вибір на одному з них, ви будетев програмістом високого світового рівня. Взагалі класний програміст повинен досконало володіти двома мовами програмування. Щоб вибрати дві мови із приведеного квартету, необхідно детально познайомитись із особливостями кожної із чотирьох мов програмування з урахуванням специфіки ваших уподобань.

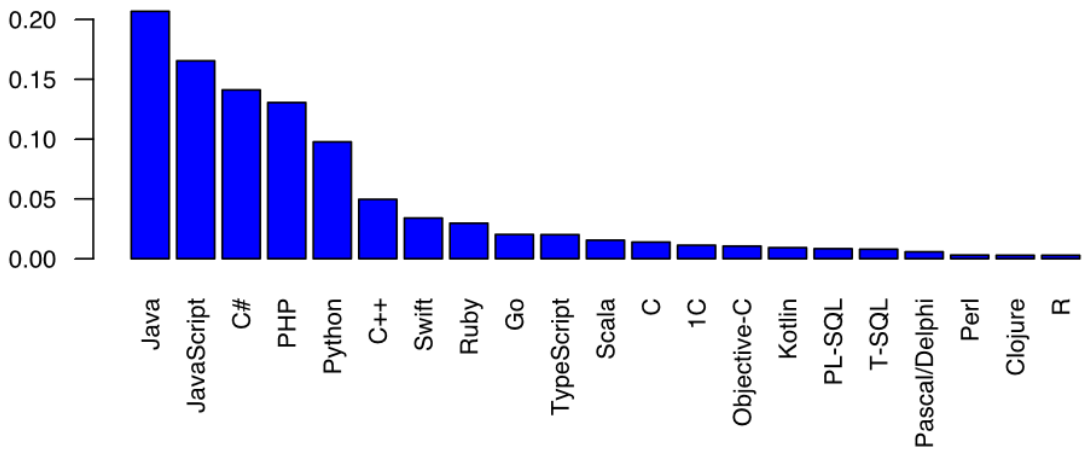

Рис. 1.1. Рейтинг мов програмування компанії dou.ua за частотою комерційного використання на початок 2018 р. В опитуванні взяло участь 7361 розробників, 90% з яких проживають в Україні. Джерело: https://dou.ua/lenta/articles/language-rating-jan-2018/

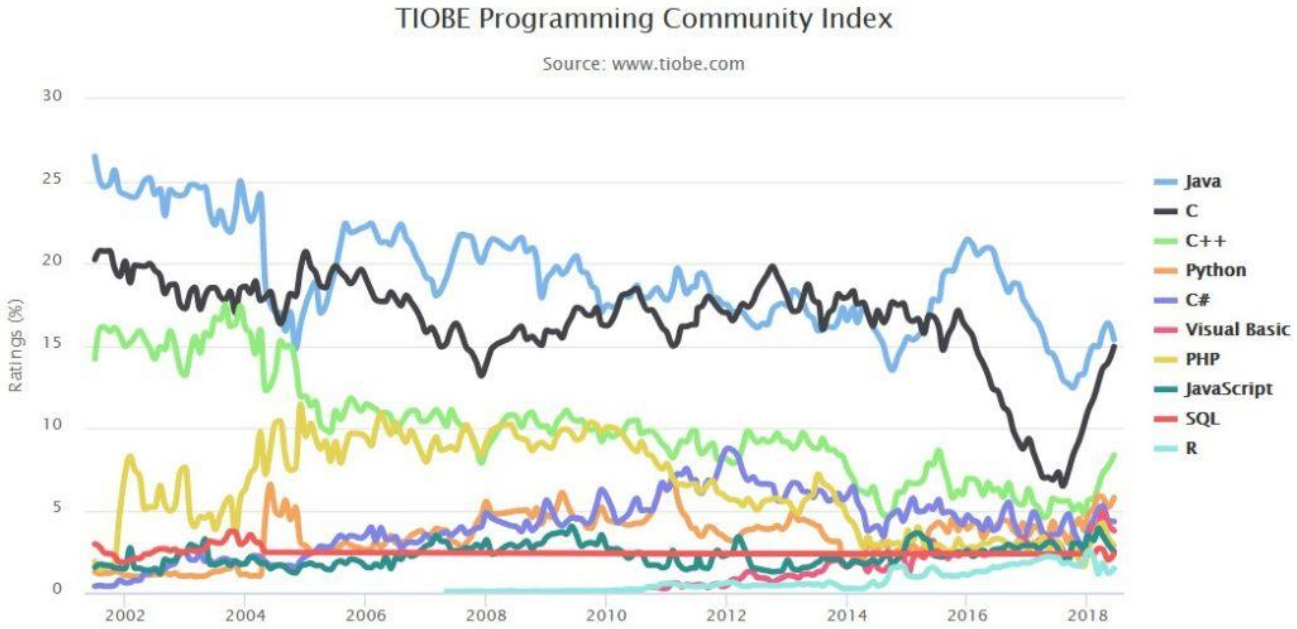

Рис. 1.2. Рейтинг популярності мов програмування в 2002-2018 рр., побудований компанією TIOBE Software шляхом аналізу трендів 25 основних пошукових систем, включаючи Google,Ebay,YouTube, Yahoo !, Wikipedia, Amazon, Bing та ін. Джерело: www.tiobe.com.

На основі Java технологій (точніше, в основному на основі J2EE-технологій) реалізовані такі проекти світового рівня, як Amazon, eBay, Twitter, Yandex, LinkedIn, Yahoo!, OpenOffice, RuneScape. Продукти компанії Google також багато в чому спираються на технології Java.

На Java-технологіях в основному спеціалізуються такі провідні компанії як Oracle, SAP, IBM. Зокрема, СУБД Oracle включає Java-машину як свою складову частину, що забезпечує можливість безпосереднього програмування СУБД на мові Java, в тому числі написання збережених процедур на Java.

Java підтримує всі новітні технології (в тому числі web- та android-розробки) і містить засоби, що значно спрощують і прискорюють процес розробки. Крім того, мова активно розвивається, а проблеми продуктивності зведені до мінімуму.

Особливості технології Java

Відомо, що програма, написана на будь-якій мові високого рівня, не може бути відразу ж виконана. Її спочатку треба *компілювати*, тобто перевести в послідовність машинних команд

процесора – *об'єктний модуль*. Але і він, як правило, не може бути відразу ж виконаний: об'єктний модуль треба ще скомпонувати з бібліотеками використаних в модулі функцій і розв'язати перехресні посилання між секціями об'єктного модуля, отримавши в результаті *завантажувальний модуль* – повністю готову до виконання програму (рис. 1.3).

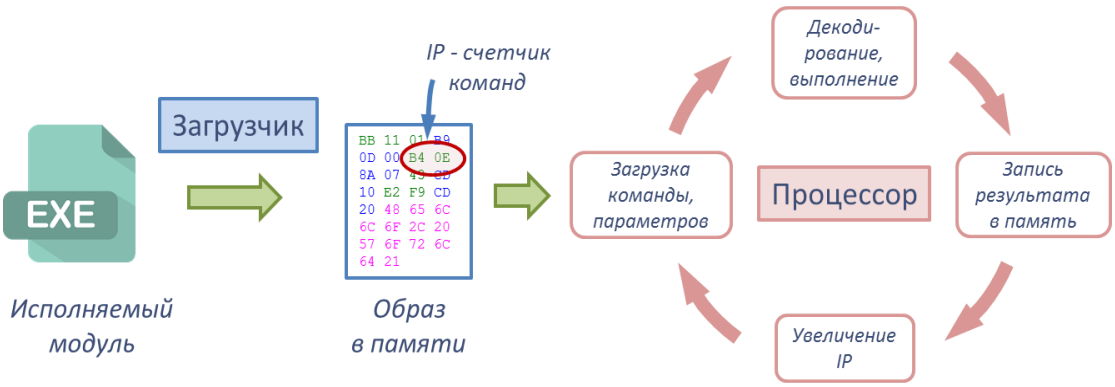

Рис. 1.3. Процес виконання звичайної програми.

Java-програма не може уникнути цих процедур, але тут проявляється головна особливість технології Java – програма компілюється не в команди якогось конкретного процесора, а в команди так званої ***віртуальної машини Java*** (JVM, Java Virtual Machine). Віртуальна машина Java – це сукупність команд разом з системою їх виконання. Віртуальна машина Java повністю стекова, використовує плоску адресацію пам'яті і невелику кількість регістрів. Тому команди JVM короткі, більшість з них має довжину 1 байт, а середня довжина команди складає 1,8 байти: ось чому команди JVM називають ***байт-кодами*** (bytecodes).

Повний опис команд і всієї архітектури JVM міститься в специфікації віртуальної машини Java (VMS, Virtual Machine Specification). Ця специфікація гарантує однакове виконання java-коду на будь-якій апаратній платформі. Так реалізується принцип Java "*Write once, run anywhere*"– "Написано один раз, виконується де завгодно".

Таким чином, програми на Java транслюються в ***байт-код***, що виконується ***віртуальною машиною Java*** (JVM) – програмою-інтерпретатором, яка обробляє команди байт-коду і перетворює їх в інструкції конкретного процесора або виклики функцій конкретної операційної системи (рис. 1.4). Такі віртуальні Java-машини розроблені для всіх популярних операційних систем, а також для багатьох маловідомих.

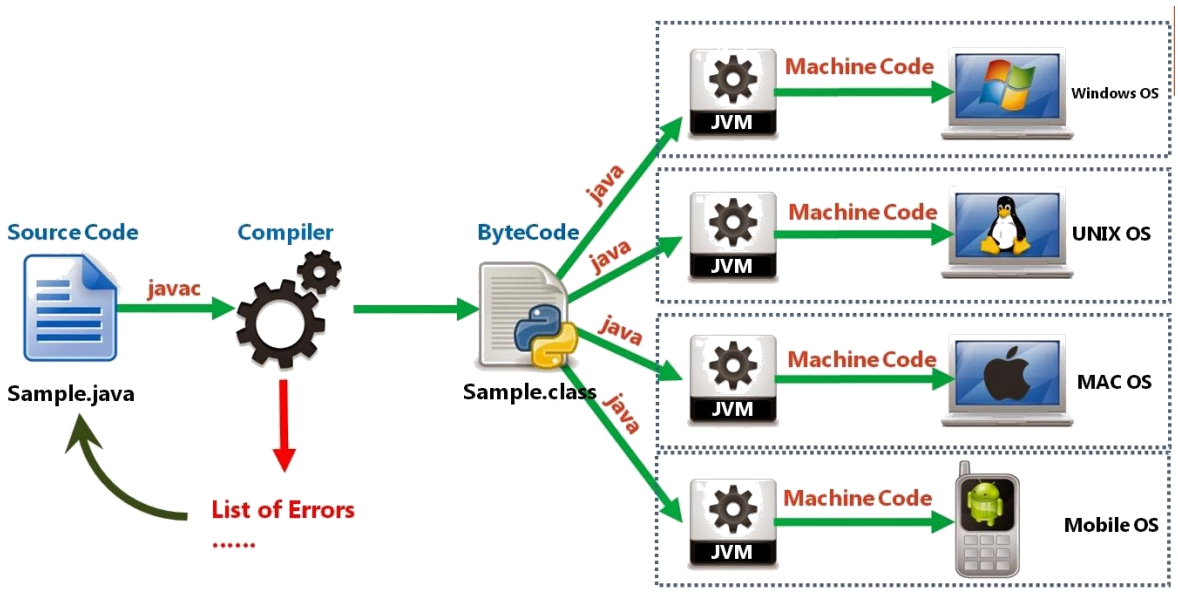

Рис. 1.4. Виконання Java-програми

Мова Java володіє всіма ***перевагами*** сучасної мови програмування:
Java-код володіє всіма перевагами сучасної мови програмування:

- *Простий* – завдяки продуманому, простому і лаконічному синтаксису: почати розробку на Java легко, навіть для програмістів-початківців (але комерційне використання Java вимагає багато знань і вміння працювати з великою кількістю технологій). Написання програм також спрощується за рахунок автоматичного управління пам'яттю і автоматичного збирання сміття.
- *Інтерпретований* – код виконується на віртуальній машині (JVM), що великою мірою визначає і наступні переваги.
- *Переносний* – повна незалежність байт-коду від операційної системи і устаткування, що дозволяє виконувати Java-додатки на будь-якому пристрої, для якого існує JVM
- *Надійний* – виключені помилки, пов'язані з управлінням пам'яттю; проста і гнучка обробка виняткових ситуацій.
- *Безпечний* – виконання програми повністю контролюється JVM. Будь-які операції, які перевищують встановлені повноваження програми (наприклад, спроба несанкціонованого доступу до даних або з'єднання з іншим комп'ютером) викликають негайне переривання.
- *Об'єктно-орієнтований* – тобто застосовує всі переваги ООП та підтримує об’єктно-орієнтований цикл розробки.
- *Високопродуктивний* (питання швидкості роботи Java-програм детально розглядається далі).
- *Багатопотоковий, розподілений* – тобто ядро мови має вбудовані механізми паралельного виконання та синхронізації потоків.
- *Активно розвивається* – постійно додаються нові можливості, в тому числі на рівні стандарту мови. Наприклад, *анотації* були додані в стандарт синтаксису java 1.5, і з тих пір дуже активно використовуються і значно спрощують розробку. Також можливості мови Java в свій час розширювалися за рахунок додавання параметричної типізації, лямбда-виразів і багато чого іншого.

Недоліки концепції віртуальної машини:

За рахунок виконання байт-коду віртуальною машиною знижується продуктивність програм і алгоритмів. За даними сайту shootout.alioth.debian.org, для семи різних задач час виконання алгоритму на Java в середньому в півтора-два рази більший, ніж час виконання того ж алгоритму на C/C ++. В деяких випадках Java-код виконується швидше, а в окремих випадках – до 7 разів повільніше. З іншого боку, в більшості з тестів споживання пам'яті Java-машиною було в 10-30 разів більше, ніж програмою на C/C ++. Також примітним є дослідження, проведене компанією Google, згідно з яким відзначається істотно нижча продуктивність і більше споживання пам'яті в тестових прикладах на Java в порівнянні з аналогічними програмами на C++.

За останні роки проведена велика робота, щоб звести до мінімуму проблеми продуктивності Java. Зокрема, були зроблені такі удосконалення:

- Застосування *JIT-технології* – тобто трансляції байт-коду в машинний код безпосередньо під час роботи програми з можливістю збереження версій класу в машинному коді. Наприклад, в циклах JVM інтерпретує повторювані ділянки коду тільки один раз, а при повторному зверненні до них просто виконує готові машинні команди.
- У стандартних бібліотеках Java широко використовується платформо-залежний код (*native-код*). Це означає, що стандартні функції Java реалізовані найбільш оптимальним чином і максимально використовують можливості API кожної конкретної ОС.
- *Оптимізація* віртуальної машини Java, яка полягає у вдосконаленні роботи самого інтерпретатора JVM з метою зменшити використання пам’яті та підвищити швидкість перетворення байт-коду на машинні команди.

- У стандартних бібліотеках Java широко використовується переносний залежний код (*native-код*). Це означає, що стандартні функції Java реалізовані найбільш оптимальним чином і максимально використовують можливості API кожної конкретної ОС.
- Розробляються *апаратні засоби*, що забезпечують прискорену обробку байт-кодів. Наприклад, фірма SUN Microsystems випустила мікропроцесори PicoJava, що працюють на системі команд JVM і безпосередньо виконують байт-коди. Компанія Oracle пропонує потужні сервери, засновані на java-процесорах. Є і Java-процесори інших фірм. Сюди ж відноситься технологія Jazelle, яка прискорює виконання байт-кодів і підтримується деякими процесорами фірми ARM.

В результаті цих новацій вже станом на 2012 рік, код Java 1.7 в тестах показав продуктивність, порівнянну з продуктивністю коду, написаного на мові С. У середньому швидкість виконання відрізнялася лише в 1.8 рази. Бенчмаркінг поточної версії openjdk 11 2018-09-25 (2018 рік), проведений ресурсом debian.net, як і раніше показує різницю в швидкості в 1.5-3 рази в порівнянні з С.

(Джерело: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/java-gpp.html).

Ідеї, закладені в концепцію віртуальної машини Java, надихнули безліч ентузіастів на розширення переліку мов, що виконуються на віртуальній машині. Ці ідеї знайшли також вираз в специфікації загальномовної інфраструктури CLI (*Common Language Infrastructure*), закладеної в основу платформи *.NET* компанією Microsoft.

Складові технології JAVA

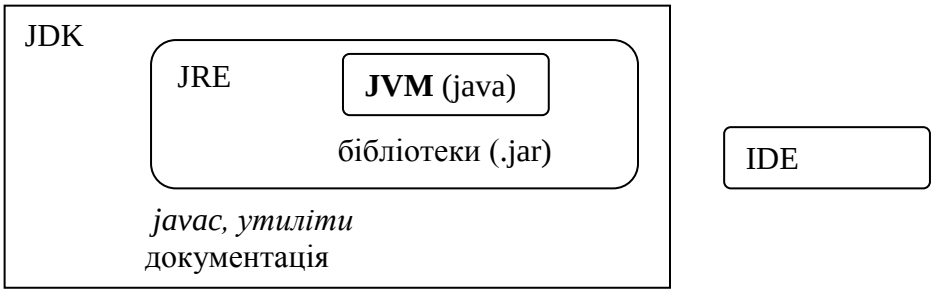

Рис. 1.5. Складові технології JAVA

Java Virtual Machine (JVM) – віртуальна машина Java є основою всієї технології (рис. 1.5). Вона безпосередньо відповідає за виконання java-програм, організовує для них роботу з системними ресурсами (пам'яттю, файловою системою) і системні виклики (тобто організує необхідні виклики функцій конкретної ОС).

Java Runtime Environment (JRE) – це мінімальна реалізація віртуальної машини, необхідна для виконання готового байт-коду. Складається з віртуальної машини – JVM і бібліотеки стандартних Java-класів, але не включає компілятор та інші засоби розробки.

Java Development Kit (JDK) – комплект розробника додатків на мові Java, що включає в себе все, що необхідне для розробки і зборки додатків, в тому числі:

- JRE, яка включає JVM (програма java.exe), стандартні бібліотеки у вигляді файлів .jar та ін.;
- Компілятор Java (програма javac.exe);
- Приклади, документацію;
- Програми-утиліти, що виконують різні допоміжні функції:
  - jar – програма архівації (стиснення) скомпільованих модулів. В результаті стиснення виходить Java-архів (з розширенням .jar), який можна розміщувати на сервері, або запускати на виконання на десктоп-комп'ютері;
  - javadoc – генерує документацію в HTML-форматі з вихідного коду класів, використовуючи для цього спеціальні коментарі в коді;
  - javah – генерує заголовкові та вихідні файли на мові С, за допомогою яких можна використовувати Java-класи в програмах на С;

- javakey – програма для додавання електронного підпису до скомпільованих модулів;
- javap – дизасемблер, що дозволяє відновити вихідний код класу за наявним скомпільованим файлом байт-коду (.class).

До складу JDK *не* входить середовище розробки (*IDE, Integrated Development Environment*), тому розробник, що використовує тільки JDK, змушений використовувати зовнішній текстовий редактор і компілювати свої програми з командного рядка за допомогою компілятора javac та інших утиліт.

Oracle і сторонні розробники також надають додаткові *SDK* (Software Development Kit – набір інструментів розробника) – набори додаткових бібліотек і утиліт, необхідних для створення певного виду додатків або компонентів. Наприклад, Java EE SDK дозволяє розробляти додатки Java Enterprise, про які йтиметься далі, Android SDK – відповідно містить все необхідне для розробки android-додатків.

Для компіляції і запуску звичайних desktop-додатків необхідним є тільки *JDK*, який можна завантажити на сайті компанії Oracle:
http://www.oracle.com/technetwork/java/javase/overview/index.html.

Існує велика кількість підручників, посібників і довідників з мови і технологій Java, наприклад: Герберт Шилдт «Java. Повне керівництво»; Кей С. Хорстманн, Гарі Корнелл «Java. Бібліотека професіонала ».

Повний та постійно оновлюваний довідник Java™ Platform, Standard Edition 10 API Specification опублікований на сайті компанії Oracle:
http://docs.oracle.com/javase/10/docs/api/index.html.

Крім того, компанія Oracle створює і підтримує керівництво для вивчення платформи java (https://docs.oracle.com/javase/tutorial/) – він містить найбільш якісний опис синтаксису мови, а також багатьох технологій, які супроводжують Java.

Приклад роботи з JDK (для Windows) і найпростіша програма на Java

Всі функції JDK орієнтовані на виклик з командного рядка. Незважаючи на це JDK повністю підтримує створення програм, що працюють в графічних середовищах, таких як MS Windows або X Window System.

Написати java-програму можна в будь-якому текстовому редакторі, наприклад, Notepad, WordPad в MS Windows, або в редакторах vi, emacs в Linux. Щоб створити java-додаток за допомогою JDK, потрібно виконати наступні кроки:

1. Підготовка Java SDK
   1.1. Завантажити останню версію SDK с сайту Oracle, або завантажити потрібну версію JDK. Після виконання установки в директорії SDK знаходяться наступні піддиректорії:
      - bin – програми, що виконують різні функції JDK (в т.ч. компілятор);
      - demo – приклади написаних на Java програм;
      - docs – документація стосовно мови та платформи Java;
      - include – заголовки native-методів;
      - jre – файли JRE;
      - lib – стандартні бібліотеки класів;
      - src.zip – вихідні коди бібліотек.

   1.2. Додати шлях до папки bin пакета SDK або JDK в системну змінну PATH. Це можна зробити в командному вікні (Win + R), ввівши системну команду із зазначенням шляху до папки bin, наприклад:
   ```
   path C:\Program Files\Java\jdk1.7.0\bin.
   ```
   У Windows 7/10 щоб змінити значення системної змінної, потрібно зайти в Панель управління – система – додаткові параметри системи – додатково – змінні середовища.
   ВАЖЛИВО: шлях до JDK повинен додаватися до наявного значенням PATH через «;», але не замінювати його.

2. Створення класу додатка.

Будь-яка java-програма складається з *класів*. Найпростіша програма має всього один клас – клас додатка. Для того щоб цей клас можна було запустити на виконання, він повинен мати спеціальний метод з ім'ям *main*(). Фактично в додатку може бути кілька класів з таким методом, тоді налаштуваннями проекту визначається, який саме з них буде запускатися при старті програми. При завершенні методу main() основний потік додатку завершується – якщо не було створено інших потоків, то це означає закриття програми.

Створимо файл вихідного коду нашого класу з розширенням .java (наприклад, HelloWorld.java) в будь-якій директорії за допомогою будь-якого текстового редактора. Назва файлу *обов'язково* має збігатися з назвою класу нашого додатку:

```
public class HelloWorld {                          // Основний клас програми
      public static void main (String [] args) {   // Головний метод
                                                    // (точка входу в програму)
                                                    // args - Параметри виклику
          System.out.println ("Hello World!");      // Виведення в консоль
      }
}
```

Метод main() має бути оголошений відкритим (*public*), щоб JVM могла його викликати. Також він повинен бути *статичним* (тобто таким, що відноситься до класу, а не до об'єкта), щоб для його виклику не потрібно було створювати об'єкт класу програми. Параметри цього методу завжди однакові і дозволяють отримати доступ до параметрів командного рядка, який використовувався для запуску нашої програми.

Всередині методу main() наразі знаходиться тільки один оператор, що використовує об'єкт System.out – стандартний потік виведення, пов'язаний з консоллю, – і викликає його метод println(), який відповідно виконує виведення на консоль.

3. Компіляція програми з командного рядка (рис. 1.6).

Викликаємо компілятор із зазначенням нашого .java-файлу.

```
C:\path\>javacHelloWorld.java
```

Результатом компіляції є файл HelloWorld.class – байт-код нашого класу (рис. 6).

4. Запуск програми. Викликаємо JVM із зазначенням імені класу-додатка (але не імені .class-файлу).

```
C:\path\>java HelloWorld
```

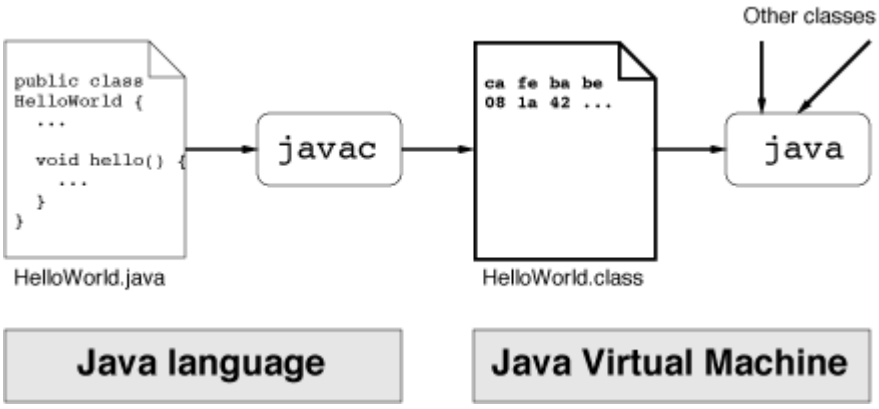

Рис. 1.6. Компіляція і виконання програми

Розглянемо тепер приклад програми, яка не просто виводить текст, але також вводить вихідні дані і проводить над ними деякі дії. Нехай необхідно порахувати суму всіх натуральних чисел від *a* до *b*. На Java це буде виглядати так:

```
import java.util.Scanner;                 // Підключаємо  клас із бібліотеки

public class Sum {                      // Основний клас програми

     public static void main(String[] args) {        // Головний метод

        Scanner scanner = new Scanner(System.in);  // Створюємо об’єкт для читання
        System.out.println("Enter a and b");       // Виводимо текст
        int a = scanner.nextInt();      // Читаємо два цілих числа
        int b = scanner.nextInt();
        int res = 0;                    // Змінна для збереження результату
        for (int i = a; i<b; i++) {     // Зчитуємо суму
            res += i;
        }
        System.out.println("The sum is "+res);     }
}
```

Тут першим рядком підключається стандартний клас *Scanner* із пакета *java.util*, який дозволяє дуже легко читати дані, введені користувачем.

Всередині методу main() створюється об'єкт цього класу – scanner (в іменах великі і маленькі букви розрізняються). Для читання цілого числа використовується метод scanner.nextInt(). Нарешті, підрахунок суми виконується циклом for, який в даному випадку абсолютно аналогічний циклу for мов C++ та C#.

Результат роботи компільованої програми буде таким:

```
Enter a and b
3   14
The sum is 88
```

Якщо в коді програми допущені помилки, то компілятор повідомить про це, із зазначенням опису помилки і рядка програми. Файл .class при цьому не створюється. Наприклад, якщо в попередній програмі написати:

```
int a = Scanner.nextInt();
```

то компілятор сповістить про помилку і опише її наступним чином:

```
Sum.java: 11: non-static method nextInt () can not be referenced from a
static context
```

Тобто в 11 рядку є помилка, яка полягає в тому, що замість імені об'єкта використано ім'я класу, а оскільки метод nextInt() не є статичним (в чому можна переконатися переглянувши документацію з цього методу), то він повинен обов'язково бути викликаний від об'єкта.

Лекція 2. Платформи JAVA і Java IDE

Платформи Java

У Java існує кілька основних сімейств технологій, які зібрані в різні «видання» (editions) і встановлюються окремо:

Java SE (Standard Edition) – основне видання Java, містить JDK для створення звичайних desktop-додатків. Саме з нього починається вивчення Java.

Java EE (Enterprise Edition) – набір технологій і специфікацій для створення програмного забезпечення рівня підприємства. Це видання призначене для створення серверних і розподілених

компонентів. Воно включає ряд серверних і web-технологій, таких як Servlet, JSP/JSTL, EJB, JAX-WS/JAX-RS, JTA (Transaction API), JPA (Persistance API). Додатки, що створюються за технологією Java EE, зазвичай також використовують сторонні фреймворки (які не є частиною Java EE), наприклад:

- Spring – набір різноманітних засобів, що спрощують розробку додатків корпоративного масштабу;
- Hibernate – фреймворк для доступу до даних із зовнішніх джерел, в першу чергу, із БД. Реалізує об'єктно-реляційне відображення (object-relational mapping, ORM), тобто відображення об'єктно-орієнтованої моделі даних в традиційні реляційні бази даних.

Java ME (Micro Edition) – створена для використання в пристроях, обмежених по обчислювальній потужності, наприклад в мобільних телефонах, КПК, вбудованих системах;

JavaFX – технологія створення додатків з розвиненим графічним інтерфейсом користувача, в тому числі RIA (Rich Internet Applications) – корпоративних і бізнес-web-додатків.

Java Card – технологія розробки додатків, що працюють на смарт-картах та інших пристроях з дуже обмеженим обсягом пам'яті і слабкими можливостями обробки.

Java и Android

Мова Java, починаючи з JDK 1.5, активно використовується для створення мобільних додатків для операційної систему Android. ОС Android поставляється із вбудованою віртуальною машиною java (JVM): ранні версії Android включали в себе JVM Dalvic, яка наразі (починаючи з Android 5.0 "Lollipop") замінена на більш сучасну ART (Android RunTime). При цьому java-програми компілюються в байт-код спеціального виду, який безпосередньо виконується такою вбудованою JVM.

Для компіляції таких додатків використовується додатковий інструмент – Android SDK, який розробляється і підтримується компанією Google. Розробку Android-додатків можна вести в різних IDE, в тому числі в IntelliJ IDEA і Eclipse (в Eclipse необхідно використовувати додатковий плагін – Android Development Tools, ADT). Також існує спеціальна збірка IDE на основі IntelliJ IDEA, яка спеціально призначена для Android-розробки – Android Studio.

IntelliJ IDEA

Із сучасних Java IDE найбільш популярні IntelliJ IDEA, Eclipse та NetBeans. Кожна з них має свої особливості і своїх шанувальників. Всі ці IDE опираються на сервіси (утиліти), що надаються JDK. У більшості випадків для компіляції програм середовище розробки використовує компілятор зі складу JDK (програму javac). Тому середовища розробки або включають в свій комплект поставки одну з версій JDK або вимагають для своєї роботи попередньої інсталяції JDK на машині розробника (в останньому випадку в налаштуваннях IDE потрібно вказати шлях до встановленого JDK).

В даному курсі в якості основної IDE розглядається IntelliJ IDEA. Це комерційна IDE для розробки на Java, яка за правом вважається найбільш зручним середовищем для Java-розробки. Весь її дизайн орієнтований на продуктивність роботи програмістів, дозволяючи їм сконцентруватися на розробці функціональності, в той час як середовище бере на себе виконання рутинних операцій. Так, IDEA має великі можливості генерації заготовок коду, конфігураційних файлів. Середовище надає потужні можливості автоматичного доповнення коду. Система підказок не тільки пропонує найбільш прийнятні об'єкти або методи, але також дозволяє автоматично виправляти багато помилок, наприклад, пропущене підключення класу або не вказану залежність модуля.

IntelliJ IDEA надає вбудований інструментарій для розробки графічного інтерфейсу – графічний редактор форм.

Серед інших можливостей, IntelliJ IDEA добре сумісна з багатьма популярними вільними інструментами розробників, такими як системи контролю версій (CVS, GIT, Subversion), системами зборки проектів (Ant, Maven), і фреймворками тестування (JUnit).

IntelliJ IDEA доступна в двох версіях: Community Edition і Ultimate Edition, порівняння яких наведене в табл. 1.1:

Таблиця 1.1. Порівняння версій IntelliJ IDEA.

Версія	Community Edition	Ultimate Edition
Ліцензія	Разповсюджується вільно під відкритою ліцензією Apache 2.0	Комерційна (платна)
Підтримує технології	Java SE, системи контролю версій, android- розробка, системи зборки Ant, Maven	Те ж + Java EE, + UML-діаграми, CASE-засоби, + інтегровані сервери додатків (Tomcat, GlassFish та ін.), + підтримка відлагодження веб-додатків, + підтримка фреймворків (Spring, Hibernate, …).

Таким чином, вільна версія Community Edition підходить для початку навчання Java і створення desktop-додатків, в той час як для комерційної розробки і створення веб- та корпоративних додатків необхідна версія Ultimate Edition. Всі практичні завдання даного курсу можна виконати в IntelliJ IDEA Community Edition. Далі ми розглянемо основні прийоми роботи в IDEA.

Створення проекту

У стартовому вікні «Welcome to IntelliJ IDEA» тиснемо «Create New Project». Або в меню вибираємо File | New project.

У вікні «New Project» (рис. 1.7):

1) Вказуємо ім'я проекту;
2) Додаємо Project SDK, тобто шлях до встановленої версії JDK:
   – тиснемо кнопку New;
   – із списку, що розкрився, вибираємо «JDK»;
   – вказуємо шлях до JDK, тобто стандартний шлях в Windows ОС, наприклад:
   ```
   C:/Program Files/Java/jdk1.9.0_51.jdk
   ```
3) Тиснемо Next внизу вікна, а потім на наступному екрані натискаємо кнопку Finish.

Рис. 1.7. Створення проекту

Створення класу

1) Створюємо новий клас в папці src на вкладці Project за допомогою правої кнопки миші (рис. 1.8). Керувати відображенням/приховуванням вкладок можна за допомогою меню View | Tool Windows або за допомогою гарячих клавіш.

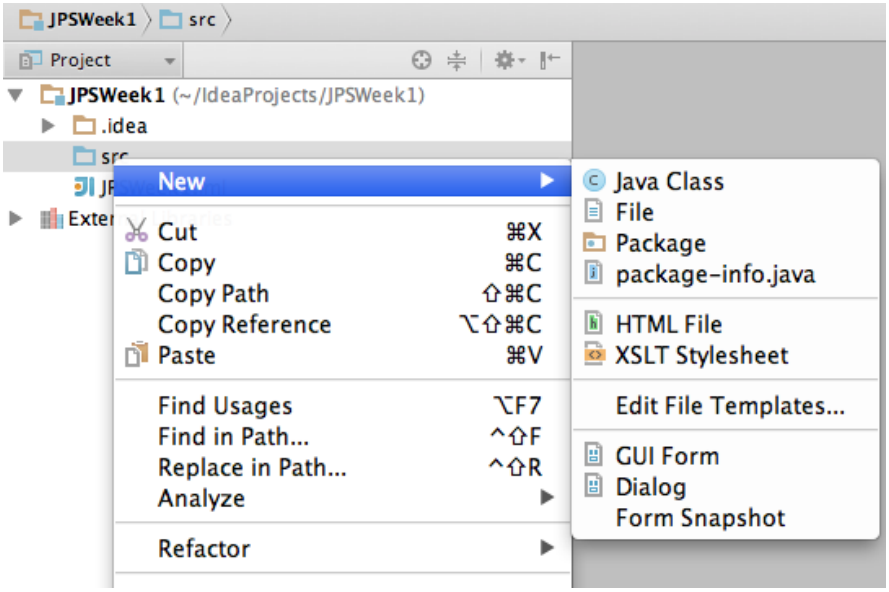

Рис. 1.8. Створення класу. Крок 1

2) Вказуємо ім’я класу HelloWorld, а в полі Kind залишаємо “Class” (рис. 1.9). Тиснемо ОК.

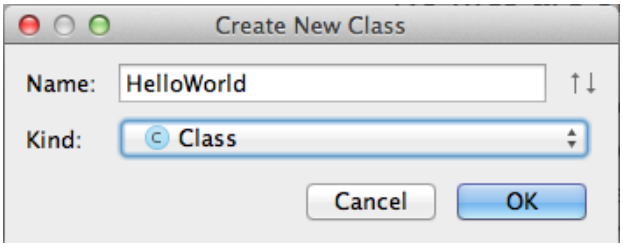

Рис. 1.9. Створення класу. Крок 2

Налаштування проекту

Налаштування проекту встановлюються у вікні Project Structure. Відобразити його можна кількома способами:

– вибравши пункт меню File | Project Structure;
– виділивши модуль в панелі Project і натиснувши F4;
– за допомогою гарячої клавіші Ctrl-Alt-Shift-S.

На вкладці Project (рис. 1.10) вказується ім'я проекту і задається версія JDK, наприклад "1.9", в полі Project SDK. Якщо JDK не задано, середовище не може впізнати стандартні класи і виділяє їх червоним кольором. Проект в цьому випадку не компілюється.

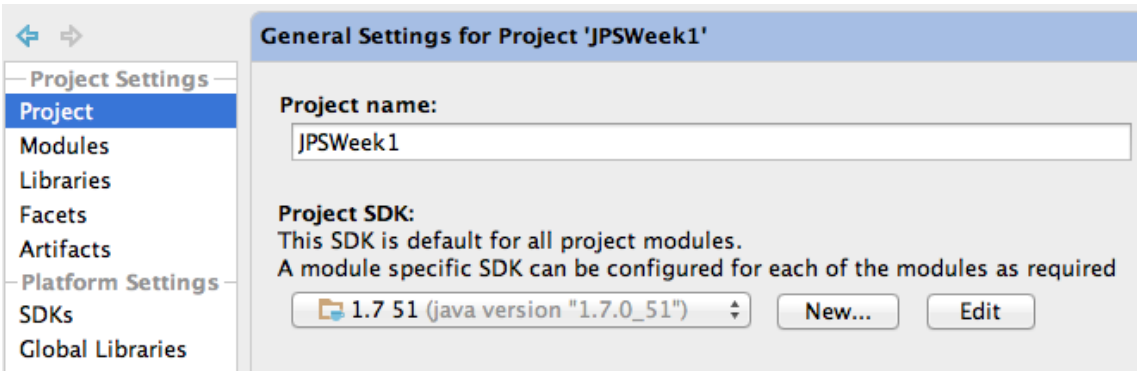

Рис. 1.10. Налаштування проекту: вкладка Project.

Якщо в списку Project SDK немає потрібної версії JDK, але вона встановлена на комп'ютері, її можна додати, натиснувши кнопку new і вказавши директорію установки JDK.

На вкладці Modules можна налаштувати JDK окремо для кожного модуля, що входить до складу проекту, а також задати інші налаштування модулів, наприклад, їх залежності.

Конфігурація запуску

Конфігурація запуску – це набір налаштувань модуля, який визначає, як він буде збиратися та запускатися в IDE. Для одного модуля можна створити кілька різних конфігурацій, наприклад, для запуску програми з різними параметрами.

Щоб налаштувати конфігурації запуску, потрібно в меню Run вибрати Edit Configurations. Конфігурації запуску звичайного desktop-додатка знаходяться в секції Application. Для кожної конфігурації задається ім'я (1), клас, що містить головний метод main() (2), домашня директорія для запущеного додатку (3), а також, якщо необхідно, опції запуску JVM, параметри командного рядка, змінні оточення (рис. 1.11).

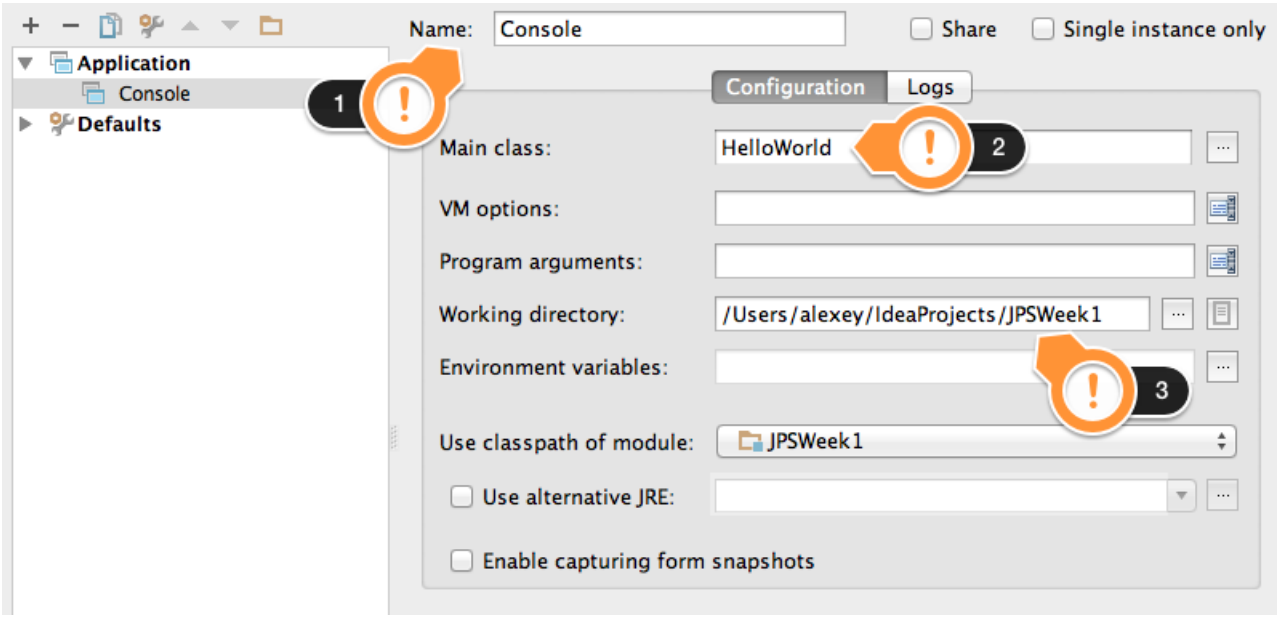

Рис. 1.11. Налаштування конфігурації запуску

Запуск та налагодження

Запустити модуль на виконання можна, вибравши команду Run (Shift + F10) або Debug (Shift + F9) з меню Run, або з контекстного меню класу додатка (який містить метод main()). Для консольних додатків IDEA не відкриває системну консоль, а направляє вихідний потік додатку в спеціальну панель Run.

У вікні налагоджувача (рис. 1.12) ви можете бачити стек викликів методів і список потоків, які наразі виконуються, разом з їх поточним станом і змінними. Вибравши контекст виклику методу, можна переглянути значення всіх змінних, присутніх в цьому контексті.

Іноді при покроковому налагодженні треба перейти всередину певного методу, але не першого який буде викликаний. В такому випадку ви можете натиснути Shift + F7 щоб вибрати із запропонованого списку метод, який потрібен. Це може заощадити масу часу.

Якщо потрібно «повернутися назад у часі» під час налагодження, ви можете зробити це, видаливши контекст виклику функції. Це сильно допоможе, якщо помилково зайдено занадто глибоко. Таким чином ви як мінімум повернетеся назад по стеку викликів функцій.

Іноді є необхідність відновити виконання програми і зупинитися на якомусь іншому рядку коду, не створюючи точку зупину. Це легко зробити – треба просто натиснути Alt + F9.

Якщо потрібно бачити якийсь елемент під час налагодження, то можна додати до нього кольорову мітку, натиснувши F11 або вибравши відповідний пункт в меню вкладки Variables і Watches. Коли цей елемент з'явиться в списку, тоді легко побачити його мітку. У режимі налагодження можна вирахувати будь-який вираз за допомогою дуже потужного інструменту натисканням Alt + F8.

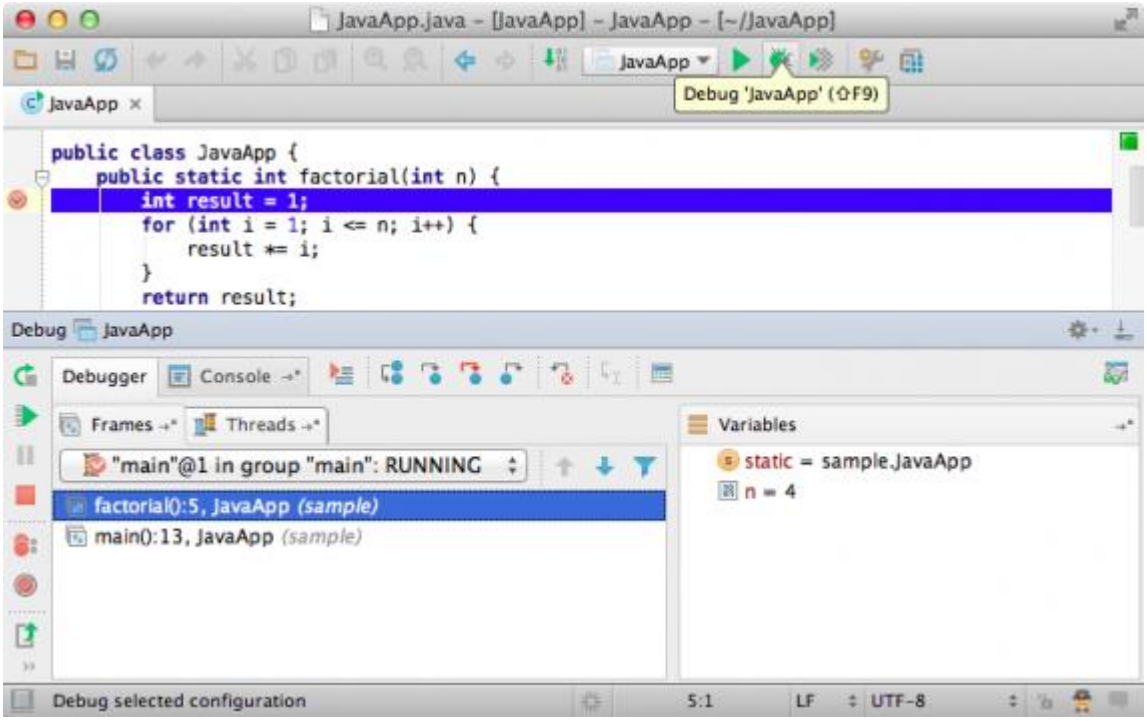

Рис. 1.12. Робота налагоджувача

Корисні клавіатурні скорочення налагоджувача

- Встановити/зняти точку зупинки – *Ctrl + F8;*
- Змінити налаштування точки зупинки (задати умову або зробити зупинку тільки для одного потоку) – *Shift + Ctrl + F8;*
- Відновити виконання програми – *F9;*
- Перейти до наступної інструкції – *F8*;
- Перейти всередину функції – *F7;*
- Виконання до позиції курсору – *Alt + F9;*
- Призупинити виконання – *Ctrl + F2;*
- У режимі налагодження ви можете обчислити будь-який вираз за допомогою дуже потужного інструменту, що викликається натисканням *Alt + F8.*

Точка зупинки змінної – це можливість налагоджувача зупинити виконання програми, коли проводиться читання або запис в задану змінну. Для того щоб створити таку точку зупинки, натисніть на панелі зліва від оголошеної змінної, затиснувши *Alt* (рис. 1.13).

Якщо ви хочете поміняти якісь настройки точки зупину, ви можете натиснути Shift + Ctrl + F. У спливаючому вікні ви можете ввести потрібні вам параметри.

Для того щоб отримати список всіх точок зупину у вашому проекті з розширеними налаштуваннями, треба знову натиснути Shift + Ctrl + F8.

На додаток до умовних точок зупину, ви можете також використовувати точки зупину змінної. Такі точки спрацьовують, коли проводиться читання або запис в якусь змінну. Для того щоб створити таку точку зупину, клацніть на панелі зліва від тексту, що редагується, навпроти потрібної вам змінної, затиснувши Alt.

Існує ще одна корисна можливість – обчислити певний вираз в потрібному рядку коду, не перериваючи виконання. Для цього потрібно клацнути на панелі зліва від редагованого коду навпроти потрібного рядка, затиснувши Shift.

З метою створення точки зупинки, яка спрацює тільки один раз, клацніть на панелі зліва від коду затиснувши Shift + Alt.

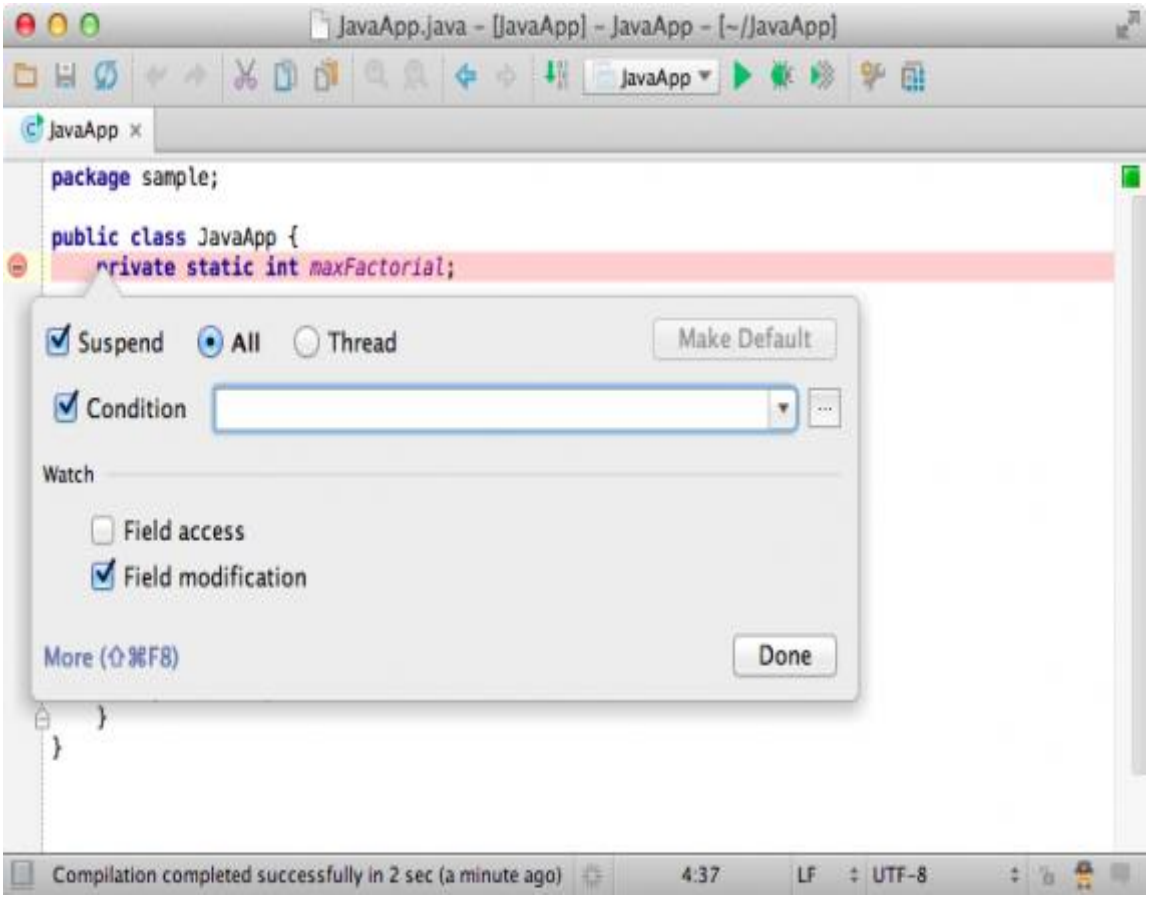

Рис. 1.13. Налаштування точки зупинки змінної

Доповнення коду

Перед тим, як розглянути варіанти доповнення коду, приведемо найбільш часто вживані так звані «гарячі клавіші» :

1. Ctrl + Space – список компонентів (класу, метада, змінної);
2. Ctrl + Q – швидкий пошук документації;
3. Shift + F1 – зовнішня документація;
4. Ctrl + наведення миші на фрагмент коду – коротка інформація;
5. Ctrl + F1 – показати опис помилки;
6. Ctrl+ / – однорядкове коментування/роз коментування;
7. Alt + Q – контекстна інформація;
8. Alt + Enter – показати пропоноване виправлення;
9. Ctrl + Alt + L – форматування коду;
10. Ctrl + X або Shift + Delete – вирізати фрагмент коду;
11. Ctrl + C або Ctrl + Insert – копіювати фрагмент коду;
12. Ctrl + D – дублювання рядка;
13. Ctrl + Y – видалення рядка;
14. Ctrl + P – відомості про параметри.

Основною метою IntelliJ IDEA є продуктивність розробника, і для цього вона надає потужні можливості автоматичного доповнення коду. Є кілька функцій доповнення коду, то ж важливо розібратися, як вони працюють і яку з них краще використовувати в кожній ситуації.

1) *Просте доповнення* пропонує найпримітивніші припущення імен змінних, типів, методів, виразів і т. д. (рис. 1.14). Воно автоматично працює по мірі введення тексту. Якщо список варіантів закінчення слова не відображається, його можна викликати примусово – *Ctrl + пробіл*. Зверніть увагу, якщо ви викликаєте простий додаток двічі, він покаже вам більше варіантів, включаючи приватні члени.

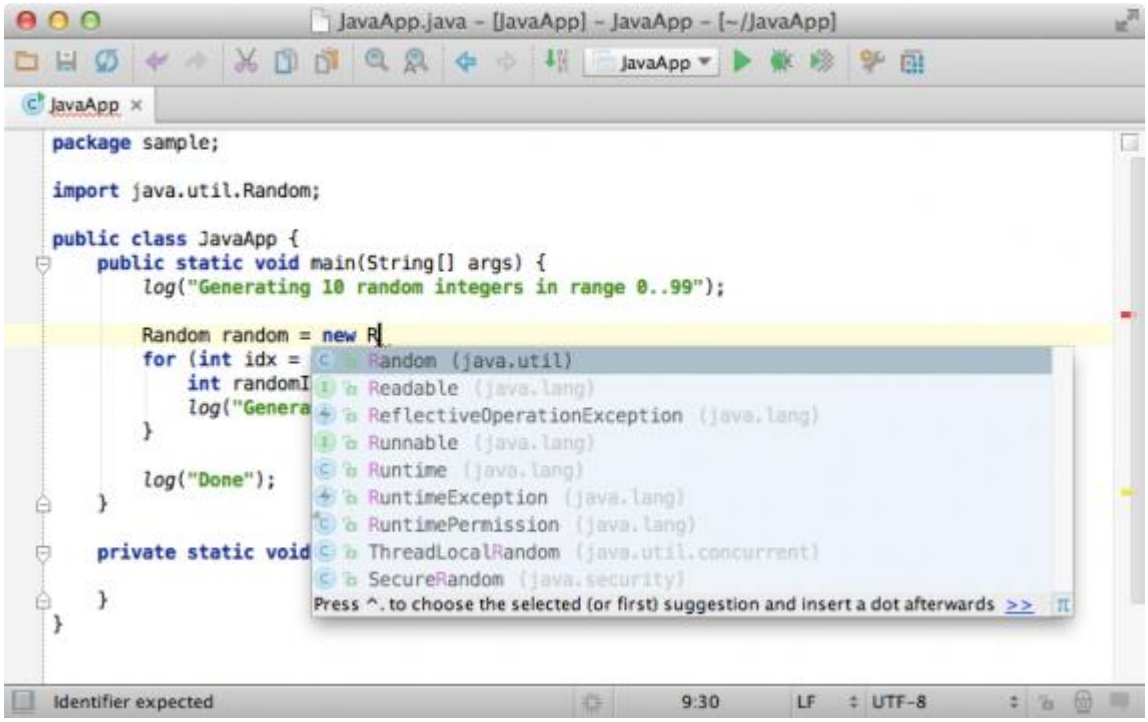

Рис. 1.14. Просте доповнення

2) *Розумне доповнення* для підбору варіантів аналізує контекст, в якому знаходиться слово що вводиться, тобто розбирається в тому, який тип виразу очікується в даномі місці коду, і як відбувається перетворення типів даних, та пропонує тільки ті варіанти, які підходять за контекстом (рис. 1.15). Для того щоб запустити *Розумне доповнення*, натисніть *Shift + Ctrl + пробіл*.

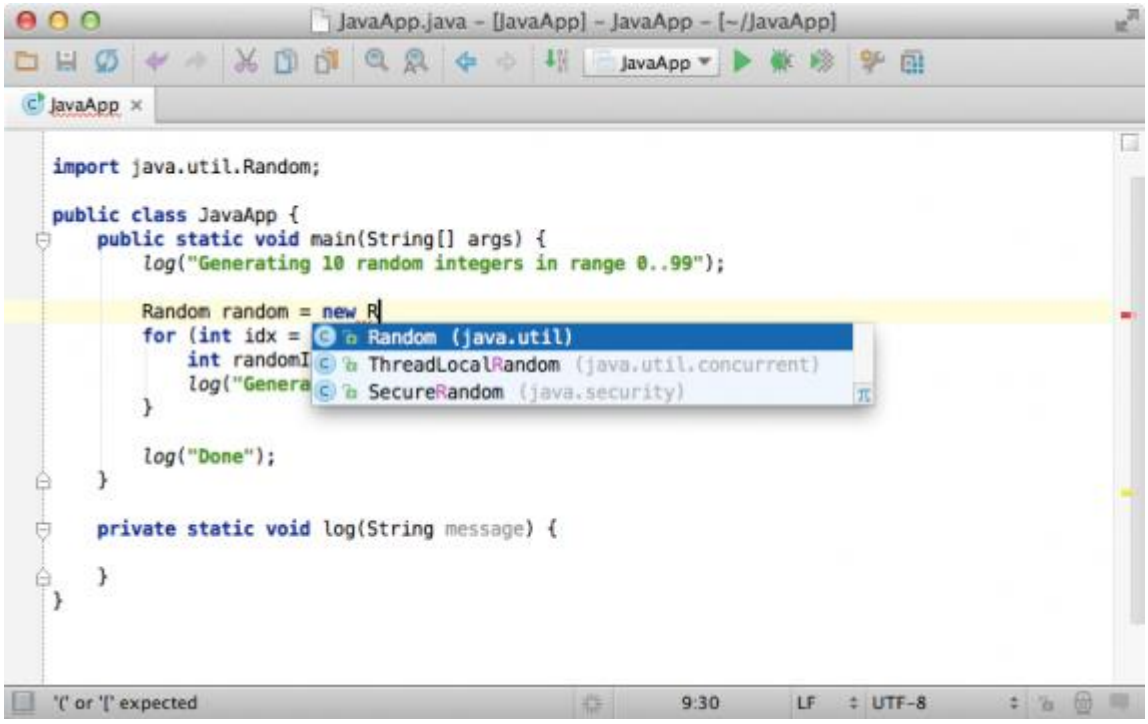

Рис. 1.15. Розумне доповнення підказує найбільш ймовірне продовження слова

Зверніть увагу, що якщо ви запустите *Розумне доповнення* двічі, воно покаже вам більше результатів, включаючи ланцюжки і не імпортовані статичні члени (рис. 1.16).

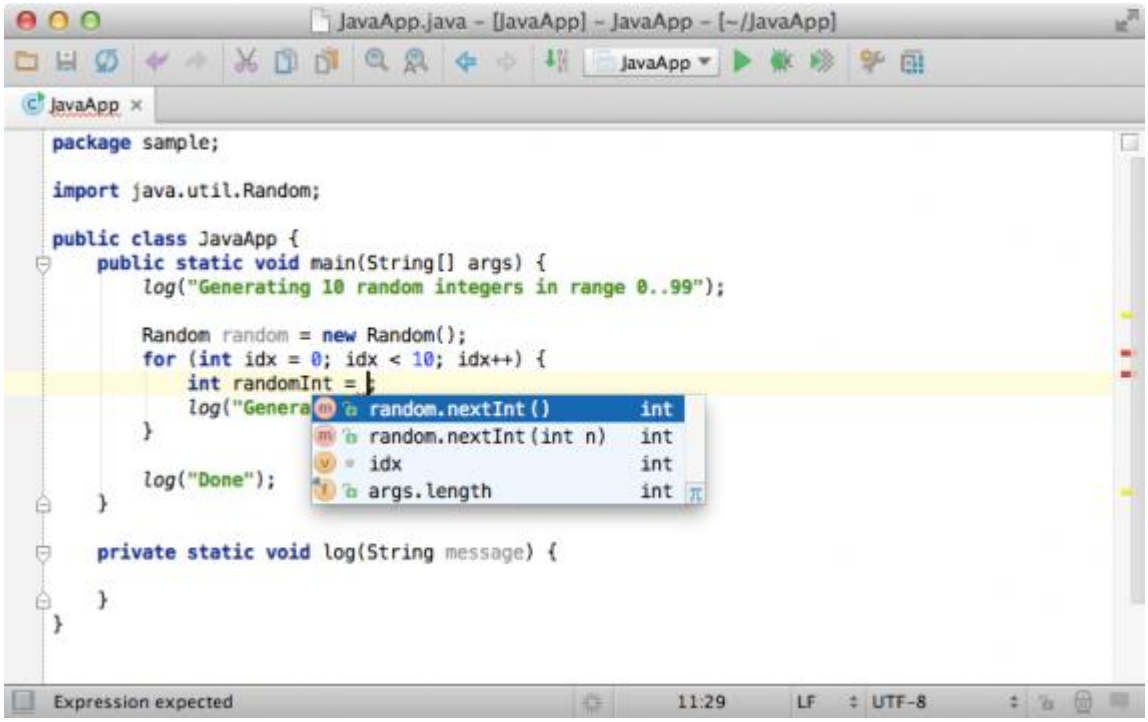

Рис. 1.16. Розумне доповнення підказує найбільш ймовірні ланцюжки за змістом

3) *Доповнення за клавішею Tab*. Якщо під час вибору елементу зі списку запропонованих варіантів натиснути *Tab*, то обраний варіант замінить собою ідентифікатор, на якому знаходиться курсор. Це буває корисно, якщо ви редагуєте частину ідентифікатора, наприклад, частину назви класу (рис. 1.17).

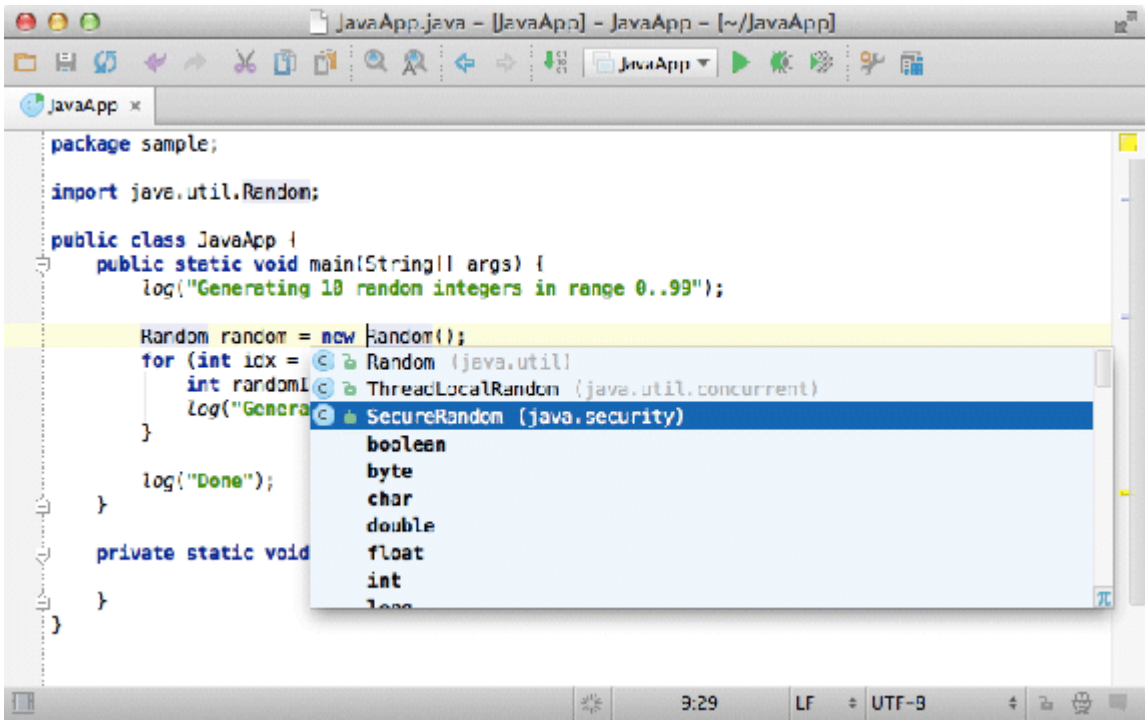

Рис. 1.17. Доповнення за клавішею *Tab*.

4) *Доповнення операторів* автоматично додає відсутні дужки, обробники до секцій *try* і необхідне форматування. Щоб доповнити оператор, просто натисніть *Shift + Ctrl + Enter*.

5) *Серединна відповідність* дозволяє не писати ідентифікатор з самого початку. Наприклад, якщо ви пам'ятаєте тільки середню частину імені класу або методу, напишіть її, і IntelliJ IDEA знайде правильний варіант (рис. 1.18).

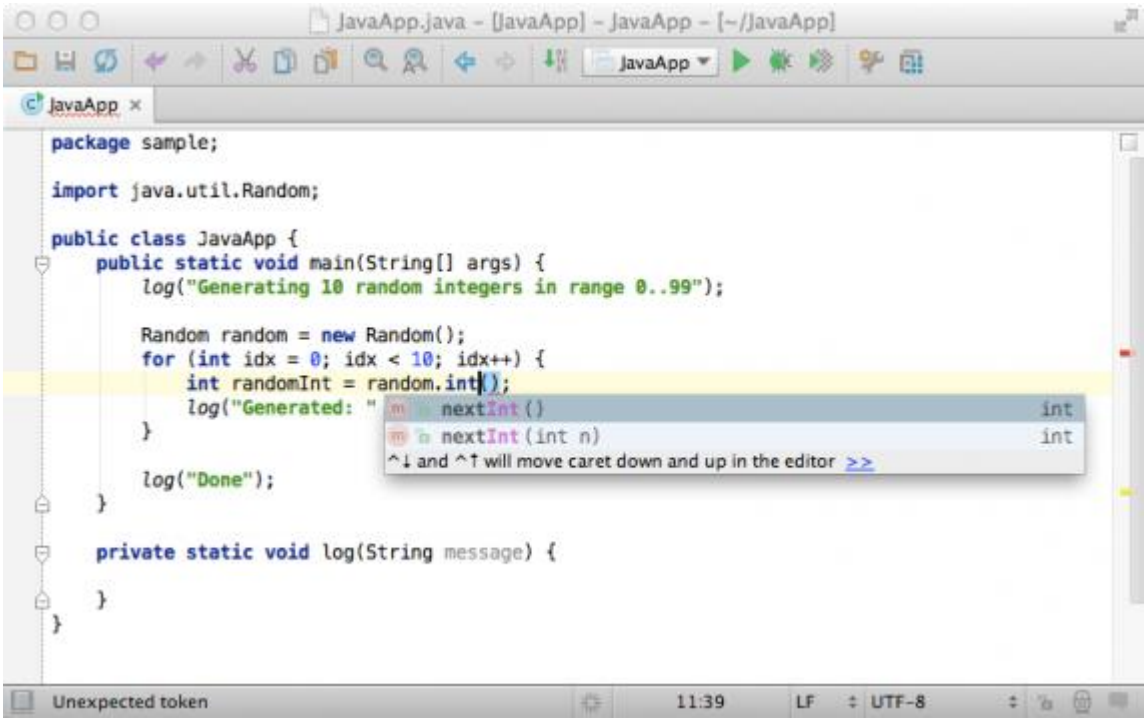

Рис. 1.18. Серединна відповідність

6) *Скорочений запис* часто використовуваних фрагментів коду. Наприклад, заголовок методу main() завжди виглядає однаково і програмісту не потрібно кожен раз писати його заново. Якщо в порожньому рядку надрукувати `psvm` і натиснути *Ctrl+Пробіл*, то IDEA згенерує стандартний заголовок main():

```
public static void main (String [] args) {}
```

Якщо всередині методу надрукувати `fori` і натиснути *Ctrl+Пробіл*, буде згенеровано заготовку коду для найбільш поширеного циклу:

```
for (int i = 0; i <; i ++) {}
```

Якщо надрукувати `sout` і натиснути *Ctrl+Пробіл*, буде згенеровано заготовку коду для виведення на консоль:

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

Спливаючі вікна та підказки

Спливаючі підказки IDEA корисні для перевірки додаткової інформації, що відноситься до елементу, на якому встановлений курсор. Наведемо список підказок, які допоможуть вам підвищити продуктивність праці.

1) Якщо ви вже відкрили документацію і хочете залишити документацію відкритою, клацніть на кнопку Pin в верхньому правому куті. Буде показана (і не зникне) документація по поточному елементу.

2) Спливаючі вікна з підказками можуть бути викликані для будь-якого елемента коду в вашому редакторі. Крім того, ці підказки можуть бути отримані і для елементів в будь-якому іншому списку, якщо при цьому використовуються такі самі сполучення клавіш.

3) *Інформація про параметри*, необхідні для виклику методу доступна після натискання *Ctrl+P*. Спливаюче вікно показує відповідний перевантажений метод, виділяючи поточний параметр (рис. 1.19).

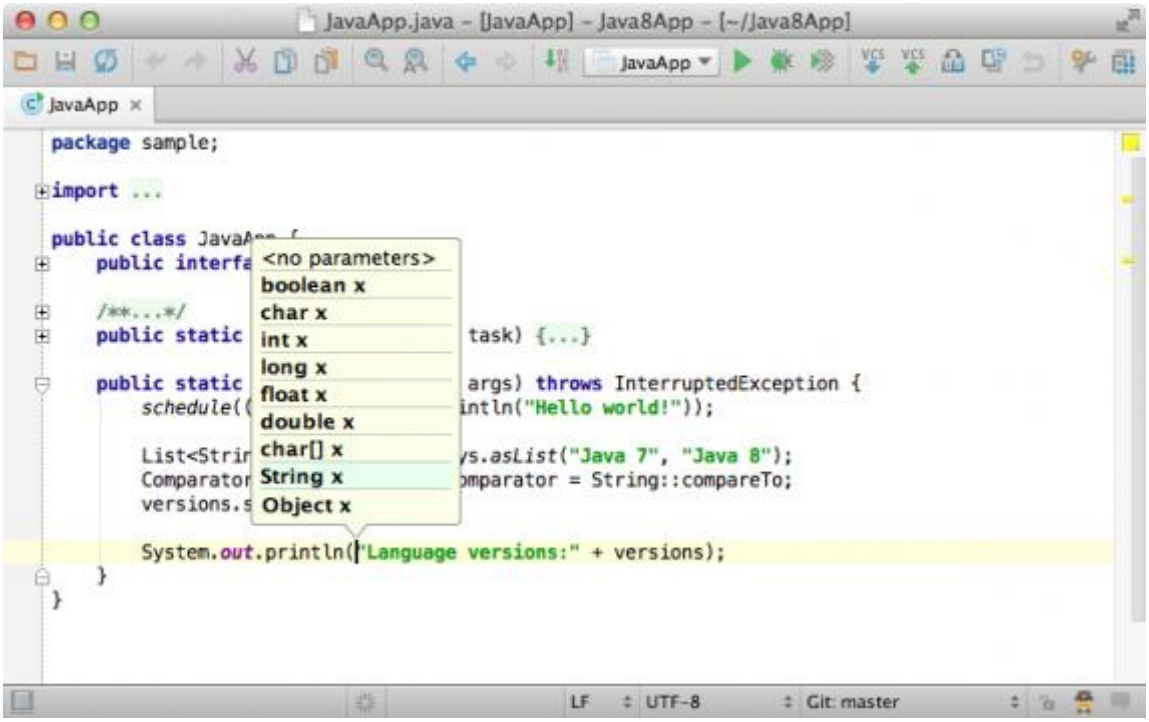

Рис. 1.19. Підказка: параметри методу

4) *Швидкий опис*. Для швидкого перегляду реалізації методу, на якому встановлений курсор, викличіть швидкий опис натиснувши *Shift + Ctrl + I* (рис. 1.20).

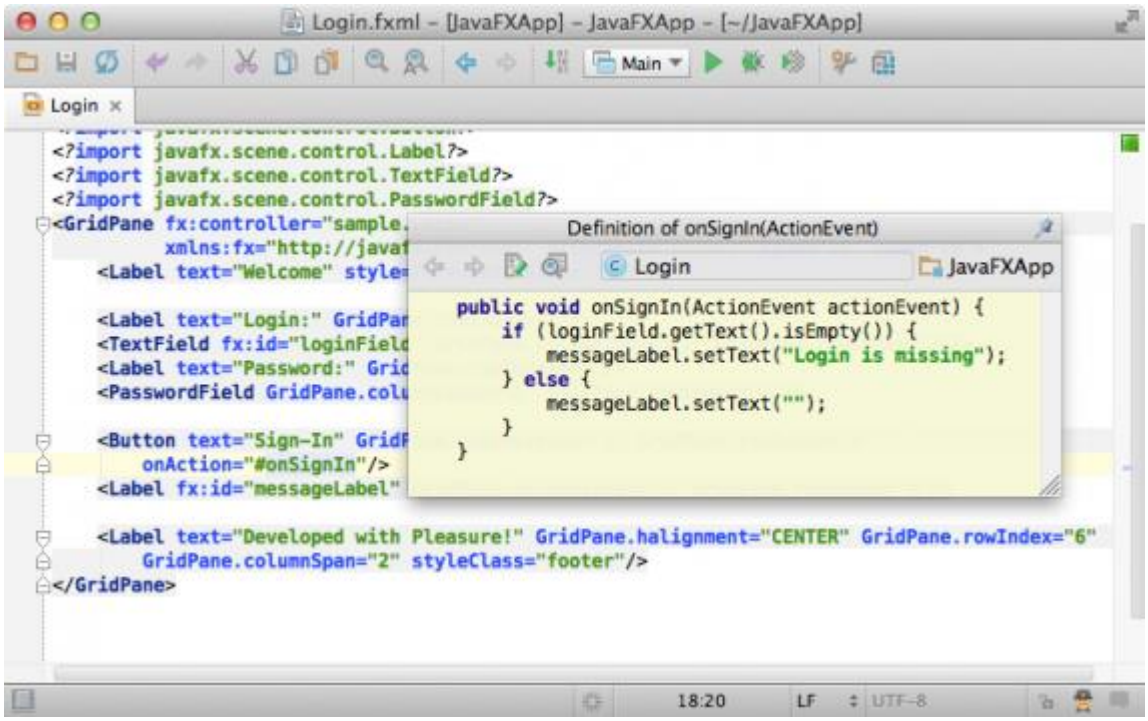

Рис. 1.20. Вікно «definition» показує, як оголошений в коді елемент,
на якому знаходиться курсор

5) *Документація*. Спливаюче вікно з документацією є одним з найбільш часто використовуваних. Воно відображає коротку довідку за класом або методом (рис. 1.21, 1.22). Щоб його відкрити, помістіть курсор на імені метода або класу та натисніть *Ctrl + Q*.

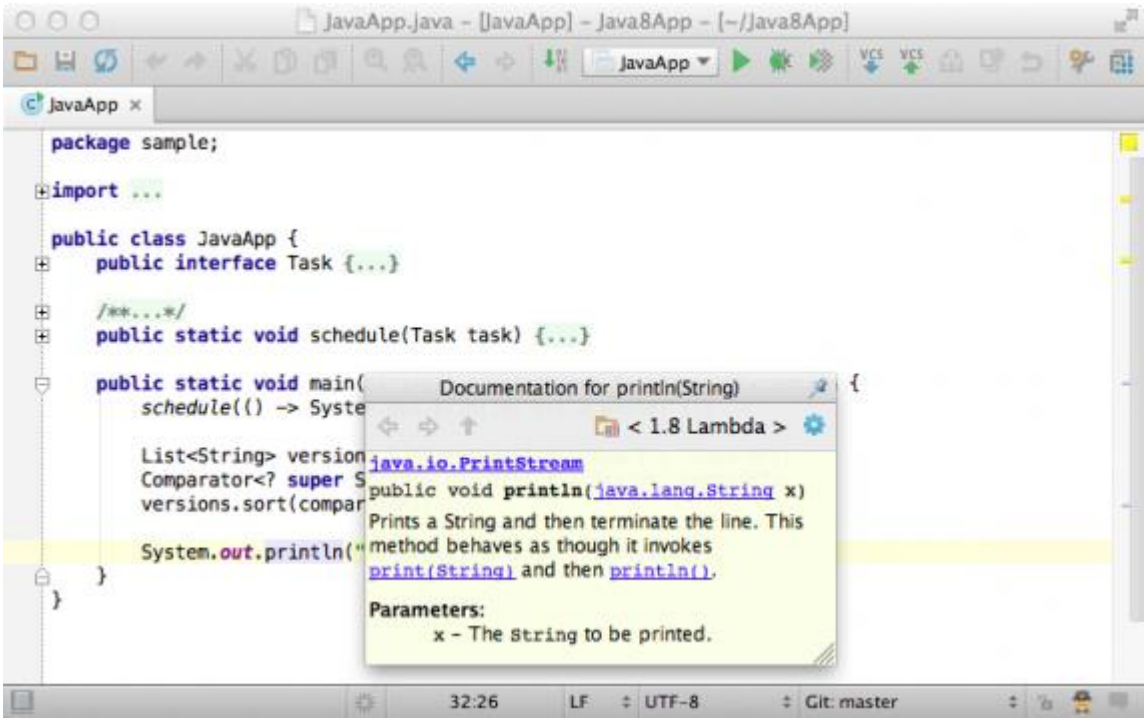

Рис. 1.21. Документація за стандартним методом println()

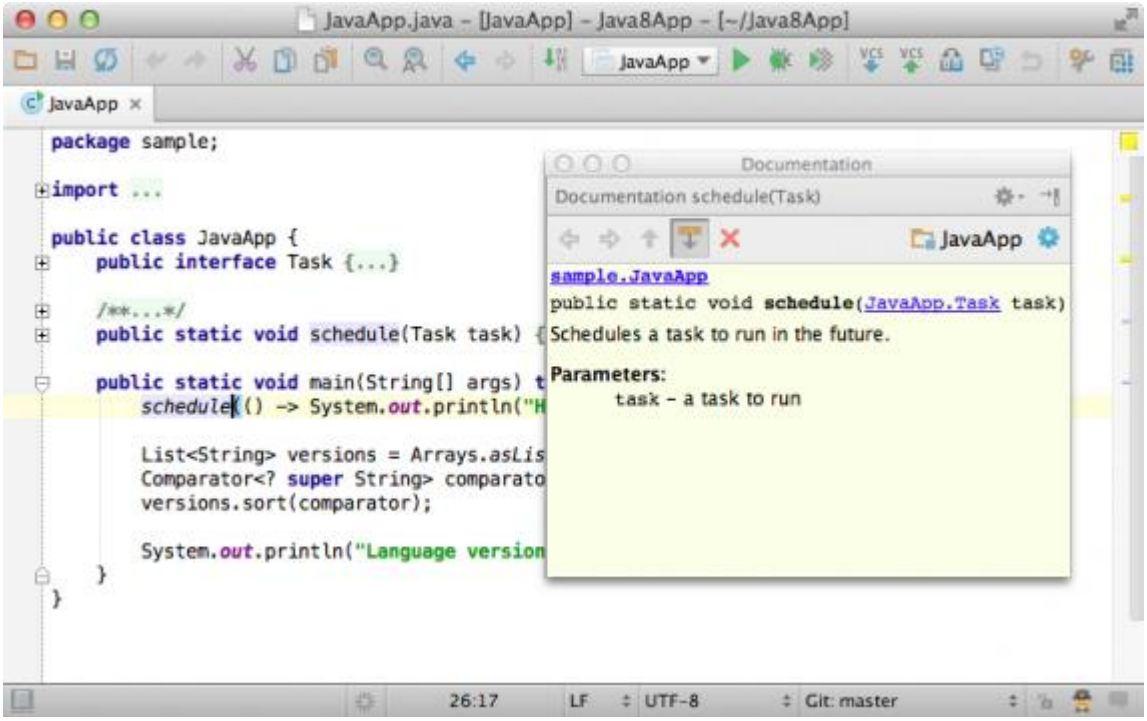

Рис. 1.22. Документація за методом shedule() з поточного проекту

Підказки без спливаючих вікон

Якщо вас не влаштовують "спливаючі" вікна, ви можете приєднати їх до краю головного вікна середовища, натиснувши на кнопку в правому верхньому кутку спливаючого вікна (рис. 1.23).

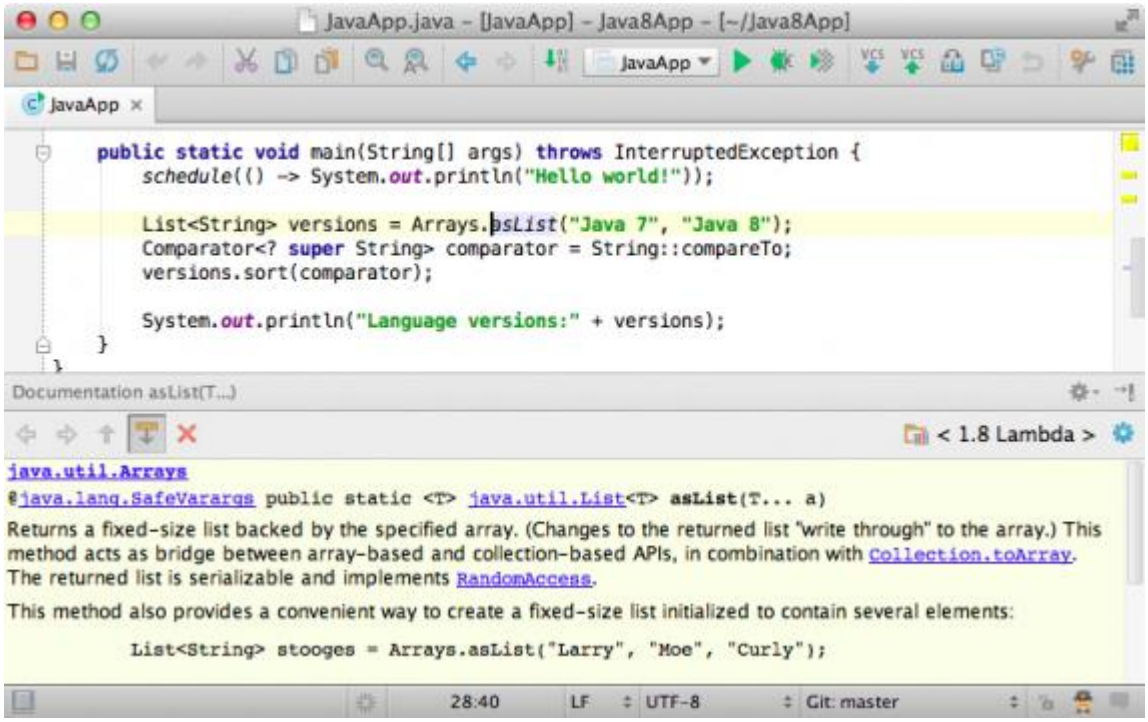

Рис. 1.23. Відображення підказок в панелі головного вікна

Спливаючі вікна в інших компонентах середовища розробки

Добре відомо, що спливаючі вікна з підказками можуть бути викликані для будь-якого елемента коду в вашому редакторі. Однак, також вони доступні і для елементів в будь-якому іншому списку або панелі при використанні таких же сполучень клавіш (рис. 1.24).

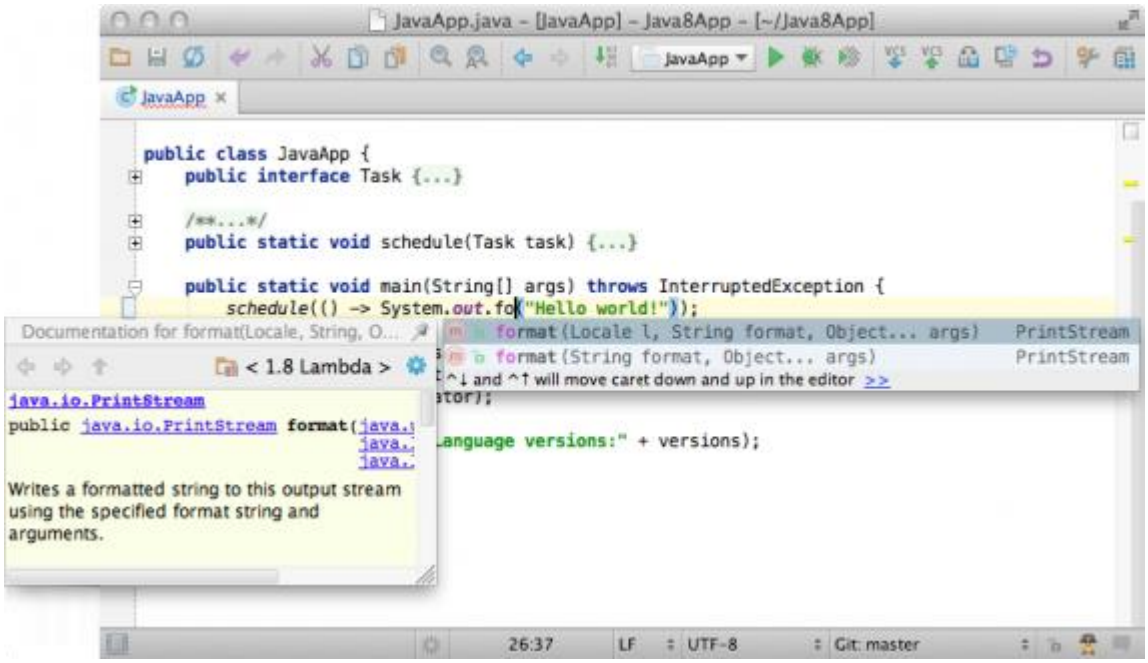

Рис. 1.24. Документація стосовно одного з елементів випадного списку в коді

Ви можете викликати спливаючі підказки навіть коли перебираєте елементи на панелі навігації після натискання *Alt + Home* (рис. 1.25).

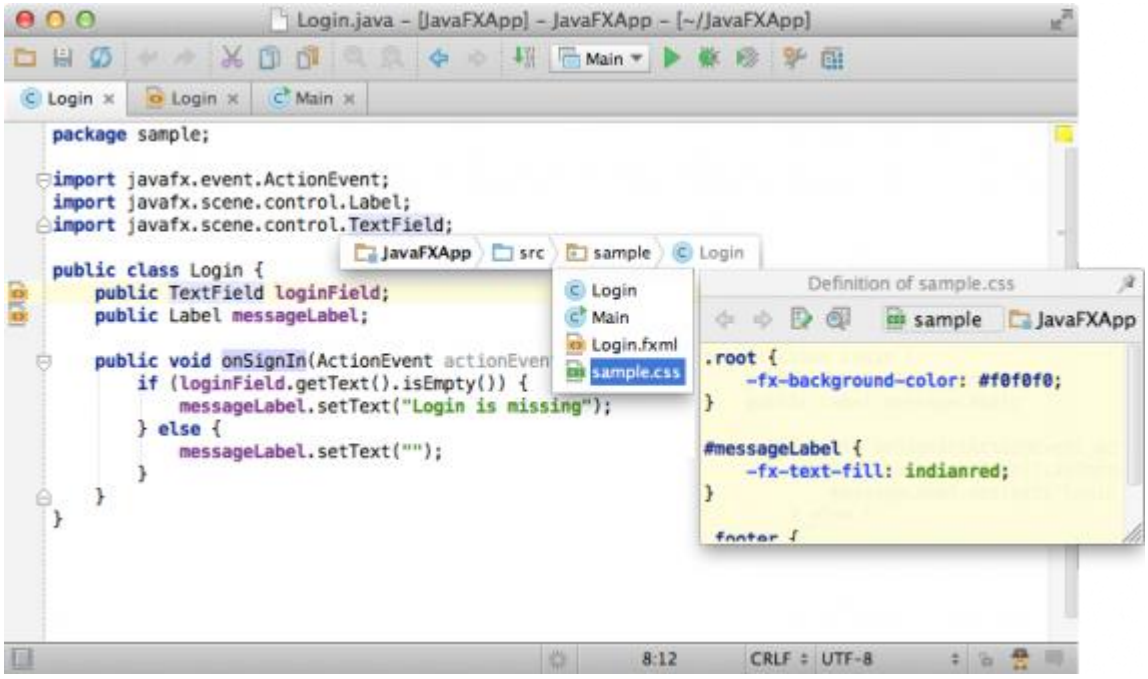

Рис. 1.25. Підказка «визначення» стосовно одного з елементів проекту

"Показати місця використання"

"Показати місця використання" – це дуже проста, але потужна функція. Помістіть курсор на ім'я змінної, методу або класу, натисніть *Alt + Ctrl + F7*, і ви побачите список тих місць в вашому коді, де використовується елемент (змінна, метод або класс), що знаходиться під курсором (рис. 1.26).

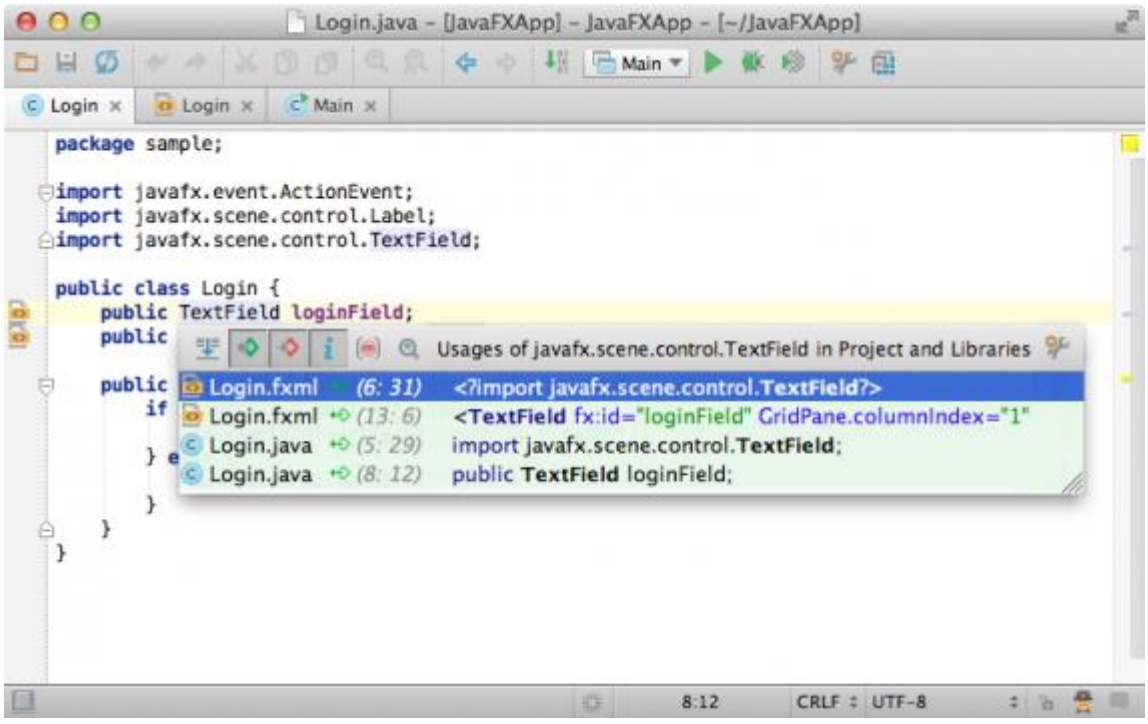

Рис. 1.26. Підказка «використання»

Основні гарячі клавіші IDEA в зручному вигляді перераховані тут: http://eax.me/intellij-idea-hotkeys/.

Крім основної IDE, коротко розглянемо також два її конкуренти – Eclipse і NetBeans – щоб розібратися, в яких випадках вони можуть бути корисні.

Eclipse

Eclipse – це вільне інтегроване середовище розробки модульних крос-платформових додатків, яке розвивається і підтримується компанією Eclipse Foundation.

Eclipse відрізняється модульною розширюваною архітектурою, маючи безліч розширень (плагінів) для всіляких завдань розробки. Eclipse також слугує платформою для розробки таких плагінів, чим він і завоював свою популярність: будь-який розробник може розширити Eclipse власними модулями-плагінами. Існують комерційні і некомерційні розширення, найбільш популярними з них є:

- розширення середовища Eclipse для роботи з базами даних, серверами додатків і ін.;
- модулі, що спрощують створення графічних інтерфейсів (наприклад, Window Builder);
- Eclipse JDT (Java Development Tools), який призначений для групової розробки та підтримує системи управління версіями – CVS, GIT;
- розширення для роботи з мовами програмування C/C++, Perl, PHP, JavaScript, Python, Ruby.

Eclipse написана на Java, тому є платформо-незалежним продуктом, тобто може працювати під усіма популярними ОС.

Основними перевагами Eclipse в порівнянні з IntelliJ IDEA є безкоштовність і велика гнучкість за рахунок плагінів. Однак в IDEA набагато краще розвинені засоби інтелектуального аналізу коду. Ці засоби дозволяють, зокрема, при підказці варіантів продовження враховувати весь контекст коду і пропонувати тільки ті варіанти, які підходять і за типом, і за семантикою.

Також IDEA має набагато більше можливостей авторефакторинга і генерації коду. Тому, якщо є можливість платити за ліцензію, зазвичай вибирають IntelliJ IDEA, а Eclipse залишається лідером серед вільних IDE.

NetBeans IDE

Це вільна IDE для мов Java, Python, PHP, JavaScript, C, C++. Проект NetBeans IDE підтримується і спонсорується компанією Oracle, однак розробка NetBeans ведеться незалежною спільнотою розробників-ентузіастів (NetBeans Community) і компанією NetBeans Org.

За якістю і можливостями останні версії NetBeans IDE наближаються до кращих комерційних (платних) IDE, таких як IntelliJ IDEA, підтримуючи рефакторинг, профілювання, виділення синтаксичних конструкцій кольором, автодоповнення набраних конструкцій на льоту, безліч вбудованих шаблонів коду. Перевагою цієї IDE також є те, що вже базова версія має повну підтримку технологій Java EE (тобто для розробки веб-додатків не потрібно нічого встановлювати додатково).

Однак в плані зручності розробки NetBeans IDE поступається IntelliJ Idea, оскільки не має настільки розвинених інтелектуальних засобів аналізу коду, а у багатьох задачах (особливо пов'язаних з J2EE-розробкою) вимагає набагато більше ручної роботи зі створення інтерфейсів, дескрипторів та ін.

Тема 2. Основи синтаксису Java

Лекція 3

Особливості синтаксису Java

- Синтаксис Java багато в чому схожий з синтаксисом C# та C++. Як ми побачимо далі, набір ключових слів і операторів цих двох мов мають багато спільного.
- Малі та великі літери – розрізняються (наприклад someName і SomeName – різні імена). Існує угода називати класи з великої літери (SomeClass), змінні та методи – з маленької (someVar, someMethod()), а константи – усіма великими літерами (SOME_CONST).
- Кожен оператор завершується крапкою з комою.
- Оператори можуть перебувати тільки всередині блоків. Блоками є тіло методу, тіло складного оператора (умовного оператора, циклу, секції try, синхронізованої секції), блок ініціалізації класу (про який піде мова в наступних лекціях). Також будь-яка послідовність операторів може бути згрупована в окремий блок. Усюди, де за правилами мови може перебувати одиничний оператор, замість нього може використовуватися блок. У всіх випадках блок виділяється фігурними дужками, наприклад:

```
if (condition) {        // Блок – тіло складного оператора
     operators;
     {                  // Блок – угрупування операторов
          Operators;
     }
}
```

- Ознакою оголошення або виклику методу є круглі дужки після імені:

```
void someMethod() {          // Оголошення методу
     operators;
}
// ...
void someMethod() {          // Оголошення методу
         someMethod();       // Виклик методу
}
```

- Java – повністю *об'єктно-орієнтована* мова. Тобто будь-яка програма будується з класів і об'єктів цих класів, процедурне програмування не підтримується.
- Мова Java *строго типізована*. Це значить, що тип змінної завжди вказується при її оголошенні, а перетворення типів обмежене жорсткими правилами. Наприклад, в Java неможливо перетворити логічне значення в ціле і назад.

  Для порівняння, в мові C++ тип змінної також потрібно вказувати при її оголошенні, але за допомогою покажчиків можна безпосередньо звернутися до області пам'яті змінної і працювати зі збереженим там значенням як з масивом байтів незалежно від типу змінної. В Java, навпаки, JVM повністю приховує деталі внутрішньої організації типу даних або класу і дозволяє виконувати тільки ті операції, які передбачені цим типом даних або класом. Механізм покажчиків в Java відсутній, тобто неможливо безпосередньо звернутися до довільної комірки пам'яті. Замість цього використовуються ***посилання***, які, подібно до покажчиків, вказують на деяку комірку пам'яті, але, в той же час, строго типізовані, як і змінні. Як ми побачимо далі, поняття «посилання» і «змінна» в Java дуже близькі і часто розглядаються як синоніми.
- Java підтримує автоматичне *керування пам'яттю*. Це означає, що програмісту не потрібно піклуватися про звільнення виділеної пам'яті. Всі об'єкти автоматично створюються в динамічній області пам'яті. JVM містить спеціальний компонент – ***збирач сміття,*** – який знаходить невикористовувані об'єкти і видаляє їх з пам'яті.

- Для створення локалізованих версій додатків дуже важлива підтримка універсальних багатобайтових кодів символів (*Unicode*). У таблиці Unicode кожен символ кожної мови представлений унікальним кодом, тому, на відміну від інших кодувань, не виникає проблем з коректним відображенням символів. У той час, як у багатьох мовах для перекодування тексту в Unicode необхідно використовувати спеціальні функції, в Java підтримка Unicode вбудована. Це означає, що прямо в коді програми можуть використовуватися будь-які Unicode-символи на будь-якій мові, і ці символи завжди коректно відображаються.
- Java має багатий набір стандартних класів. Так, є ціла вбудована бібліотека класів-колекцій, які реалізують різноманітні структури даних: вектор, список, стек, хеш-таблицю. При цьому багато структур даних мають кілька різних реалізацій з різними властивостями. Наприклад, структура даних «список» реалізована як на основі динамічного масиву (клас ArrayList), так і на основі зв'язного списку (клас LinkedList).
- Java має потужні засоби (бібліотеки, фреймворки, допоміжні технології), що полегшують створення web-додатків (в тому числі з використанням протоколу RMI), web-сервісів і розподілених компонентів. Найбільш простим прикладом є *сервлети* – класи, що дозволяють обробляти і виконувати HTTP-запити.
- До мови вбудовані засоби створення багатопотокових додатків – синхронізація потоків і управління ними. Сюди відносяться секції синхронізації, підтримка моніторів об'єктів, з якими познайомимося далі в цьому курсі.

Синтаксис мови Java активно розвивається. З кожною новою версією з'являються нові можливості для прискорення процесу створення програмних продуктів. Наприклад, починаючи з версії JDK 1.5, до складу мови включені ***анотації*** – спеціальні короткі вставки, зазвичай декларативного характеру, які часто здатні замінити великі фрагменти звичайного коду. У наступних версіях в ядро мови додані можливості параметричної типізації (***generics***), *лямбда вирази* і безліч інших засобів, що дозволяють легко писати більш ефективний і читабельний код.

Оголошення

Всі змінні, методи і класи, використовувані в програмі, повинні бути заздалегідь ***оголошені***. Розглянемо синтаксис оголошень. У дужках [] показані необов'язкові елементи оголошення, а жирним шрифтом виділені обов'язкові ключові слова.

1. Змінні:
```
[Спосіб_доступу] [модифікатори] Тип ім'яЗмінної
                                    [= початковеЗначення];
```
2. Константи:
```
[Спосіб_доступу] static  final  Тип ім'яЗмінної
                                    [= ПочатковеЗначення];
```
3. Методи (функції)
```
[Спосіб_доступу] [модифікатори]
ТипЗначенняЩоПовертається   ім'яМетода (
                         ТипПараметра ім'яПараметра, ...) {
   Оператори;
   return ЗначенняЩоПовертається;
}
```
4. Клас:
```
[Спосіб_доступу] [модифікатори] class им'яКласу {
      Тіло класу (оголошення змінних і методів)
}
```

Тут *Спосіб_доступу* – це один з модифікаторів `private`, `protected`, `public`. Тонкощі використання способів доступу ми розглянемо в наступних лекціях. На початковому етапі спосіб доступу можна просто не вказувати.

До *модифікаторів* змінних відносяться ключові слова `final`, `static`, `volatile`, `transient`. Зміст цих модифікаторів будемо розглядати далі.

Як приклад наведемо оголошення невеликого класу, що містить оголошення двох методів sum() і main(). У методі main() також оголошуються локальні змінні, необхідні для його роботи: i, j, res. В цілому, синтаксис оголошень в java схожий з іншими близькими мовами – C# і C++.

```
public class Summ {              // Оголошення класу додатку

   int sum(int a, int b) {       // Метод, повертаючий ціле число
      return a+b;
   }

   public static void main(String[] args) {        // Головний метод
      int i = 1, j = 2;                            // Дві цілі змінні
      double res = sum(i,j);                       // Дійсна змінна
      System.out.println(res);
   }
}
```

Ключові слова (key words)

Нижче наведено список ключових слів Java. Всі ці слова можна використовувати як імена. Жирним виділені ті з них, які є новими в порівнянні з мовою C++. Курсивом виділено ключові слова, зміст яких істотно змінений у порівнянні з C++. Нарешті, сірим виділені зарезервовані слова, які на даний момент не використовуються.

abstract	continue	for	*new*	switch
assert	default	goto	**package**	synchronized
boolean	do	if	private	this
break	double	**implements**	protected	throw
byte	else	**import**	public	throws
case	enum	**instanceof**	return	**transient**
catch	**extends**	int	short	try
char	**final**	interface	*static*	void
class	**finally**	long	**strictfp**	*volatile*
const	float	**native**	**super**	while

Літерали (константи)

Деякі імена зарезервовані за ***літералами***. Це не ключові слава, але вони також не можуть бути використані в якості ідентифікаторів (імен) у вашій програмі:

```
true;
false;
null.
```

Правила запису **арифметичних констант** в цілому аналогічні мові C++. Числа можна записувати в десятковій системі:

- 10 – звичайна ціла константа (типу int);
- 1.23 – звичайна дійсна константа (типу double);
- 1e5 – 100000 в експоненційної форе;
- 1.23e-3 – 0.00123 в експоненційної формі;

в вісімковій:

- 010 – це число 8;
- 0123 – це 83 (1 * 64 + 2 * 8 + 3);

або в шістнадцятковій:

- 0x10 – це 16
- 0x123 – це 291 (1 * 256 + 2 * 16 +3).

Починаючи з Java SE 7 константи цілочислових типів byte, short, int, long можна записувати в двійковій системі, наприклад:

```
byte aByte = (byte)0b00100001;
short aShort = (short)0b1010000101000101;
int anInt2 = 0b101;
```

Такий запис дуже зручний при перевірці бітових «прапорців». Наприклад, щоб перевірити, 3-й і 8-й біти змінної flag на рівність нулю, достатньо написати:

```
if ((flag & 0b100001000) == 0b0) { ... }
```

Для вказівки типу константи застосовуються суфікси:

l (або L) – long,
f (або F) – float,
d (або D) – double.

Наприклад, 1L – одиниця, але типу long, а не int.

Строкові літерали записуються в подвійних лапках, наприклад "`це рядок`".

Символьні літерали записуються в апострофах, наприклад `'F'`, `'ш'`.

В строкових і символьних літералах є правила для запису спеціальних символів. По-перше, є набір визначених спеціальних символів:

– '\ n' – кінець рядка (переведення рядка)
– '\ r' – повернення каретки
– '\ t' – табуляція

і ряд інших.

По-друге, можна явно записати восьмеричний код символу, наприклад: '\ 001' – це символ з кодом 1.

Керуючі конструкції

1. Умовний оператор (if)

Синтаксис:

```
if ( <умова> )
       <оператор1>
[else
       <оператор2>]
```

Тут `<умова>` — це логічний вираз типу boolean, тобто вираз, що повертає true або false. Вирази інших типів, навіть цілочислових, не можуть бути перетворені в boolean, тому перевірка на нуль завжди вимагає явного порівняння:

```
if (a == 0) {...}
```

Як видно з синтаксису, частина else є необов'язковою. Після if і після else стоїть по одному оператору. Якщо потрібно помістити туди кілька операторів, то потрібно поставити **блок** { }. В Java прийнято блок ставити завжди, навіть якщо після if або else стоїть один оператор.

Приклади:

```
if ( a > b ) {
    x = a;
```

```
} else {
    x = b;
}

if ( flag ) {
    flag = false;
    init();
}
```

В останньому прикладі flag – логічна змінна або поле, init() – метод, що викликається, якщо флаг дорівнює true (кажуть, "якщо flag встановлений").

2. Багатоваріантний вибір (switch)

Служить для організації вибору по деякому значенню однієї з кількох гілок виконання.
Синтаксис:

```
switch ( <вираз> ) {
       case <константа1>:
             <оператори1>
       case <константа2>:
             <оператори2>
       ...
       [default:
             <оператори_D>]
}
```

Зауважимо, що <вираз> має видавати ціле число або символьне значення; константи повинні бути того ж типу, що і значення цього виразу.

Елементи case <константа>: є мітками переходу. Якщо значення виразу збігається з константою, то буде здійснено перехід на цю мітку. Якщо значення виразу не збігається ні з однією з констант, то все залежить від наявності фрагмента **default**. Якщо він є, то перехід відбувається на мітку default, якщо його немає, то весь оператор switch пропускається.

В операторі switch фрагменти case не є блоками. Якщо після останнього оператора даного case-фрагмента стоїть наступний case, то виконання буде продовжено, починаючи з першого оператора цього case-фрагмента. Тому в операторі switch зазвичай застосовується оператор **break**, який ставиться в кінці кожного case-фрагмента.

3. Оператори циклу

3.1. Цикл з передумовою (while)

Синтаксис:

```
while ( <умова> )
        <оператор>
```

Як і у випадку оператора if, в Java прийнято <оператор> заключати в фігурні дужки.
Приклад:

```
int x = 123, i=0;
while ( x>0 ) {
    x /= 2;
    i++;
}
```

3.2. Цикл с постумовою (do while)

Синтаксис:

```
do
        <оператор>
while ( <умова> );
```

Відрізняється від попереднього тільки тим, що умова кожен раз перевіряється не перед, а після виконання тіла циклу. Тому в **do while()** тіло циклу завжди виконується як мінімум один раз, в той час як **while()** може не виконатися жодного разу, якщо умова є помилковою спочатку.

Приклад:

```
int x = 123, i=0;
do {
    x /= 2;
    i++;
} while ( x>0 );
```

Даний приклад еквівалентний попередньому.

3.3. Цикл for()

Синтаксис:

```
for ( <ініціалізація>; <умова>; <зміна> )
     <оператор>
```

Вираз <ініціалізація> виконується один раз перед першим витком циклу. Перед кожним витком перевіряється <умова>, і після кожного витка циклу виконується вираз <зміна>. Як і в if() вираз «умова» може бути тільки логічним (типу boolean).

Для зручності складання циклів з лічильником в даній конструкції можливо кілька варіантів:

- <ініціалізація> може бути не виразом, а описом змінної з ініціалізацією, наприклад `int i = 0`. Така змінна є *локальною* для циклу і не доступна після його завершення.
- <ініціалізація> може бути списком виразів через кому, наприклад `i = 0, r = 1`.
- <зміна> також може бути списком виразів, наприклад `i++, r *= 2`. Ці два випадки використання оператора «*кома*» є винятком. Ніде, крім заголовка циклу for цей оператор не використовується.
- Всі складові (<ініціалізація>, <умова> і <зміна>) є необов'язковими. Якщо відсутня умова, то вона вважається завжди істинною, і вихід з циклу повинен бути організований якимись засобами всередині самого циклу.

Приклад:

```
int x = 123;
for (int i = 0; x>0; i++) {
x /=2;
}
```

3.4. Цикл for each

Синтаксис:

```
for (<оголошення змінної> : <масив або колекція> )
     <оператор>
```

Це дуже зручна модифікація циклу for, яка дозволяє пройти по всіх елементах масиву або колекції без використання лічильника. Рекомендується всюди, де це можливо, використовувати саме таку форму циклу. Усередині цієї форми for обов'язково оголошується нова локальна змінна, тип якої повинен бути сумісний з типом елементів масиву (колекції). Наприклад:

```
int sum =0, a[] = {5,6,7,8};
```

```
for(int i : a)  sum+=i;
```

Тут перед циклом оголошується масив `a[]` з чотирьох цілих чисел. Всередині циклу оголошується змінна `i` – теж ціла. На кожному витку виконання циклу в змінну `i` заноситься (копіюється) значення чергового елемента з масиву, і для неї виконується тіло циклу. Цикл гарантовано обробить всі елементи масиву і після цього завершиться. Зверніть увагу, що в даному випадку через змінну `i` неможливо змінити вміст масиву. Зовсім інша ситуація буде, якщо в масиві зберігаються не числа, а об'єкти – до цього питання повернемося в наступній лекції.

4. Оператори переходу: break, continue, return.

Використання цих операторів повністю аналогічно мові C++.

break – завершує виконання циклу або оператора switch();

continue – перехід на наступний виток циклу;

return – завершення поточного методу з можливістю повернути значення в метод, що його викликав.

Як приклад спільного використання операторів розгалуження і циклів розглянемо програму, яка генерує випадковим чином 100 символів латинського алфавіту і класифікує їх як "голосні", "приголосні" і "іноді голосні". В останню категорію віднесені символи 'y' і 'w'.

```
public class SymbolTest {

    public static void main(String[] args) {
        for ( int i = 0; i < 100; i++ ) {
            char c = (char)(Math.random()*26 + 'a');
            System.out.print(c + ": ");
            switch ( c ) {
            case 'a': case 'e': case 'i':
            case 'o': case 'u':
                System.out.println(" - голосна");
                break;
            case 'y': case 'w':
                System.out.println(" – інколи голосна");
                break;
            default:
                System.out.println(" - приголосна");
            }
        }
    }
}
```

В даному прикладі є кілька нових для нас елементів.

- Використовується метод random() класу Math. Подивимося документацію по класу Math і розберемося, що він робить.
- В операторі
  ```
  char c = (char) (Math.random() * 26 + 'a');
  ```
  проводиться складання арифметичного значення з символом. При такому складанні в Java символ перетворюється в число, яке дорівнює коду цього символу.
- В операторі
  ```
  System.out.print(c + ":");
  ```
  символ с приводиться до строкового типу для з'єднання з рядком.

Слід також звернути увагу на фрагменти case. Формально тут 7 таких фрагментів, але 5 з них не містять ніяких операторів. Оскільки в них немає операторів break, фактично для кількох символів виконуються однакові дії.

Введення-виведення

У цій лекції розглянемо засоби *консольного введення-виведення*. Самі по собі консольні додатки використовуються нечасто, але вивчені класи і методи дозволять в майбутньому аналогічно виконувати введення-виведення з текстовими файлами і іншими текстовими потоками, наприклад, з мережевими з'єднаннями.

Форматоване виведення за допомогою об'єкта `System.out`

Крім методів print() і println(), з якими ми вже знайомі, об'єкт System.out надає методи format() і printf() для форматованого виведення. Можливості цих методів однакові і збігаються з можливостями функції printf() мови C.

Формат виведення вказується за допомогою першого параметру – рядка, що містить *специфікатори формату*. Для цілих чисел використовуються специфікатори %d, %x; для дійсних – %f, %e, %g (незалежно від типу – float або double); для рядків – %s. Після символу % можна вказувати ширину поля виведення і кількість знаків після коми для речових значень. Наприклад:

```
double d = Math.PI;
System.out.printf("%f; %10.2f;\n %-10.0ff\n", d, d, d);
```

виведе

```
3,141593;       3,14;
 3        f
```

Якщо тип переданого значення не відповідає типу специфікатора, то генерується *виключення*. Якщо таке виключення не оброблено явно, то це приводить до аварійного завершення програми.

Для форматованого виведення в рядок можна використовувати статичний метод `String.format()`, який має аналогічні параметри. Він створює об'єкт класу String, і в нього записує відформатований текст, наприклад:

```
System.out.println( String.format("%s = %f", "Pi", Math.PI) );
```

Клас `java.util.Scanner`

В Java існує кілька способів реалізації форматованого введення. Найбільш простий з них – використовувати клас `java.util.Scanner`. Цей клас знаходиться в пакеті java.util, тому його необхідно підключити на початку програми:

```
import java.util.Scanner;  // Підключаємо клас на початку програми
```

При створенні об'єкт класа Scanner зв'язується з потоком введення, файлом чи рядком:

```
Scanner in = new Scanner(System.in);          // Стандартний потік введення
Scanner in1 = new Scanner( "123 + 125" );           // Рядок
Scanner in2 = new Scanner( new File("a.txt") );   // Файл в директорії
                                                    // проекту
```

У разі читання з файлу додатково необхідно обробляти можливі *виключення*. У всіх випадках вхідні дані розглядаються як послідовність *слів*, між якими стоять *роздільники*. За замовчанням роздільниками є пробіл, табуляція і новий рядок. Для читання даних за словами в класі Scanner є методи виду **next*Type*(),** де *Type* – один з примітивних типів:

```
String s = in.next();   // зчитати наступне слово у вигляді рядка
int a = in.nextInt();   // зчитуємо ціле число
byte b = in.nextByte(); // зчитуємо байтове число
...
double d = in.nextDouble();  //зчитуємо дійсне число
```

Читання до кінця рядка виконує метод **nextLine()**.

Приклад 1. Зчитуємо вирази виду "число операція число"

```
Scanner in = new Scanner(System.in);
int i1 = in.nextInt();          // зчитуємо ціле число a
char c = in.next().charAt(0);   // зчитуємо  знак операції
int i2 = in.nextInt();          // зчитуємо  ціле число b
String c = in.nextLine();     // пропускаємо все до кінця рядка
```

Методи **next*Type*()** мають такі особливості:

- вони блокують поточний потік виконання програми, поки в потоці введення не з'явиться наступне слово (або кінець рядка);

- всі вони, крім next() і nextLine() генерують виключення, якщо формат слова не відповідає очікуваному типу.

Тому для кожного з методів next*Type*() є метод **hasNext*Type*(),** який перевіряє, чи є в потоці слово потрібного формату, але нічого не вводить і не змінює поточну позицію в потоці. Також є методи **hasNext()** і **hasNextLine()**, які перевіряють наявність в потоці відповідно слова або кінця рядка, без урахування формату.

Метод **useDelimiter(String pattern)** дозволяє задати роздільник слів за допомогою ***регулярного виразу (Regex)***. У Java регулярні вирази використовуються дуже часто і допомагають розпізнавати рядки і виділяти значущі частини рядків. *Регулярний вираз – це рядок спеціального виду, шаблон, який описує формат тексту.* Схожі функції виконує рядок формату в методі printf(), але регулярні вирази – це набагато більш гнучкий і потужний засіб. Типовими задачами, які розв'язуються за допомогою регулярних виразів є:

- перевірка правильності введення, наприклад, перевірка правильності написання Email або номера телефону;
- пошук фрагментів тексту певного виду, наприклад, пошук всіх тегів-посилань в html-документі;
- контекстна заміна, наприклад в тексті замінити всі коми, що стоять перед великою літерою, на крапки.

Найпростішим регулярним виразом є звичайний рядок без всяких спеціальних символів, наприклад пробіл " ". За допомогою такого виразу можна знайти тільки точні співпадіння в тексті. Якщо ж потрібно знайти, наприклад, всі пробільні символи (пробіли, переклади рядків, табуляції), то потрібно використовувати спеціальний символ "\s".

Приклад 2. Зчитування послідовності чисел, розділених пробілами, комами, або крапкою з комою.

```
Scanner in = new Scanner("123, 123 345 ;   33");
in.useDelimiter("[\\s,;]+");    // Набір символів-роздільників
                                // задається регулярним виразом
while( in.hasNextDouble() ) System.out.println( in.nextDouble() );
if (in.hasNextLine()) System.out.println( in.nextLine() );
          // Останнє слово, посля якого може не быть роздільника
```

Примітка. Тут регулярний вираз `[\\s,;]` означає будь-який із символів «пробіл», «кома», «крапка з комою». Перед "\s" ставиться повторний слеш, щоб компілятор не намагався замінити ці два символи на один подібно '\n'. Символ '+' після дужок означає, що роздільник може складатися з одного або декількох символів. Якщо '+' не поставити, то два пробіли, що йдуть один за другим, розпізнаються як два окремих роздільники, між якими знаходиться порожнє слово, і це порожнє слово буде з'являтися на виході методу next().

Більш докладно регулярні вирази розглянемо далі.

Приклад 3*. Виведення вмісту файлу на екран.

```
import java.io.File;        // Підключаємо клас для роботи з файлами
// ...
try {     // Блок, в якому можливі виключення
     // Зв'язуємо Scanner з файлом
     // Тут можливі виключення, наприклад, якщо такого файлу немає
     Scanner in = new Scanner( new File("1.txt") );
     while (in.hasNextLine())                    // Поки є рядки в файлі
          System.out.println( in.nextLine() );   // Виводимо їх
} catch (Exception e) {          // Обробник виключення
     e.printStackTrace();        // Вивести інформацію про виключення
}
```

Методи

boolean hasNext(String pattern);

String findInLine(String pattern);

призначені для пошуку в потоці введення відповідно слова або під-рядка, що відповідає заданому формату. Формат задається у вигляді регулярного виразу. Для доступу до результатів порівняння використовується метод **Scanner.match()**.

Приклад 4. Нехай у файлі urls.txt записані посилання URL і коментарі до них, наприклад:

```
My references:
Reference 1:   ftp://file_server.com:21/top_secret/life_plans.pdf
https://regexone.com/lesson/introduction#section     - reference 2
file://localhost:4040/zip_file
https://s3cur3-server.com:9999/
market://search/angry%20birds
```

Необхідно обробити цей файл і вивести всі домени, на які ведуть посилання, із зазначенням портів, до яких вони звертаються. Наступний короткий фрагмент коду виконує це завдання, використовуючи клас Scanner і регулярні вирази.

```
String pattern = "(\\w+)://([\\w\\-\\.]+)(:(\\d+))?";   // Рядок-шаблон
Scanner in = new Scanner(new File("urls.txt"));         // Відкрити файл
while (in.hasNextLine()) {                          // Пока файл не закінчився
    if (in.findInLine(pattern) != null) {           // Перевірити, чи є в рядку URL
        String domain = in.match().group(2);        // Отримати частини URL
        String port = in.match().group(4);
        if (in.match().group(4) == null) {          // Перевірити наявність порта
            port = "default";
        }
        System.out.format("Domain: %s, port: %s\n", domain, port);
    }
    in.nextLine();                                  // Пропустити весь залишок рядка
}
```

Для наведеного файлу результатом роботи програми буде наступний текст:

```
Domain: file_server.com, port: 21
Domain: regexone.com, port: default
Domain: localhost, port: 4040
Domain: s3cur3-server.com, port: 9999
Domain: search, port: default
```

В цьому прикладі використано відразу кілька можливостей регулярних виразів, детальніше про які можна дізнатися з документації Regex.

Лекція 4

Примітивні типи даних

Типи даних в Java поділяються на примітивні і посилання (рис. 2.1). ***Примітивні типи*** (які також називають скалярними, а інколи простими) є найбільш простими у використанні. Вони багато в чому схожі з типами даних мови C++. Змінна такого типу являє собою іменовану комірку пам'яті, в якій зберігається деяке значення – число або символ.

Типи-посилання – це типи *об'єктів*. Будь-який об'єкт в Java-програмі може бути доступний тільки через посилання на нього, звідси і назва відповідних типів. До типів-посилань відносяться: масиви, класи і інтерфейси.

Всі типи-посилання є спадкоємцями класу Object, тобто всі вони пов'язані спільною ієрархією класів. Примітивні типи стоять особняком і в цю ієрархію не входять. Необхідно завжди пам'ятати, що робота з примітивними типами і посиланнями принципово різниться.

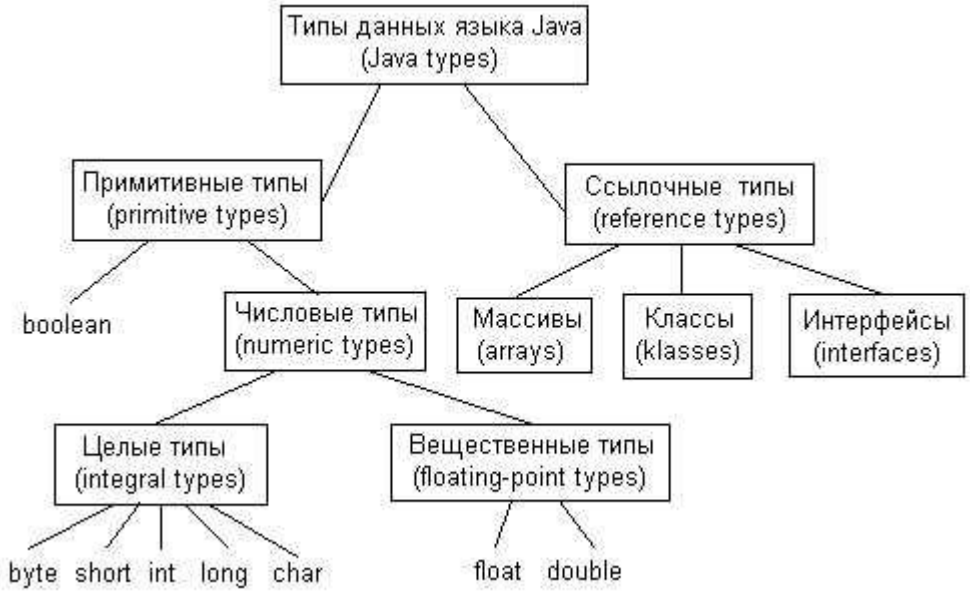

Рис. 2.1. Типи даних Java

Спочатку розглянемо примітивні типи Java (табл. 2.1). Їх всього 8: boolean, byte, char, short, int, long, float, double. Довжини і діапазони значень примітивних типів визначаються стандартом мови (а не конкретною її реалізацією: версією JDK, платформою) і однакові для всіх віртуальних машин. Вони наведені в таблиці 2.1. Тип char зробили двобайтовим, щоб він міг вміщати символи Unicode. Як вже говорилося раніше, це сильно полегшує локалізацію додатків: будь-який символ будь-якої мови має унікальний код і завжди коректно відображається. Коли складався стандарт мови Java, вже існував Unicode-16, але ще не було Unicode-32, тому символьний тип став саме двобайтовим. Оскільки в результаті не залишилося однобайтового типу, додали новий тип byte, причому в Java, на відміну від інших мов, він має знак. Зверніть увагу, в Java немає жодного беззнакового числового типу (тип char через жорсткі правила типізації з числовими типами несумісний).

Таблиця 2.1. Примітивні типи даних.

Тип	Довжина (в байтах)	Діапазон чи набір значень
boolean	1 – в масивах, 4 – в змінних	true, false
byte	1	−128..127

char	2	$0..2^{16}-1$, чи 0..65535
short	2	$-2^{15}..2^{15}-1$,чи −32768..32767
int	4	$-2^{31}..2^{31}-1$, чи −2147483648..2147483647
long	8	$-2^{63}..2^{63}-1$, чи приблизно $-9.2\cdot10^{18}..9.2\cdot10^{18}$
float	4	$-(2-2^{-23})\cdot2^{127}..(2-2^{-23})\cdot2^{127}$, чи приблизно $-3.4\cdot10^{38}..3.4\cdot10^{38}$, а також –∞, ∞, NaN
double	8	$-(2-2^{-52})\cdot2^{1023}..(2-2^{-52})\cdot2^{1023}$, чи приблизно $-1.8\cdot10^{308}..1.8\cdot10^{308}$, а також –∞, ∞, NaN

Принципи представлення значень примітивних типів в пам'яті (рис. 2.2) в цілому збігаються з тими, що використовувалися в мові C++, тому окремо зупинятися на них не будемо.

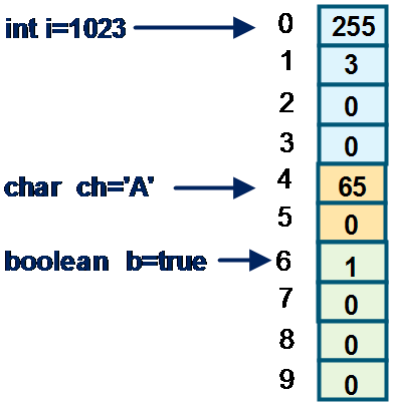

Рис. 2.2. Розташування примітивних типів в пам’яті

Важливо знати, що практично для всіх примітивних типів є також класи-обгортки, імена яких аналогічні, але починаються з великих літер (наприклад замість int – Integer, замість char – Character, замість double – Double). Ці класи підтримують ті ж операції, що і відповідні примітивні типи, а змінні цих класів у виразі автоматично приводяться до відповідних примітивних типів. В цілому можна сказати, що клас-обгортка може бути використаний скрізь, де можна використовувати примітивний тип.

Ці класи-обгортки в основному використовуються там, де за логікою програми необхідно число, але за правилами мови примітивний тип не допускається. Наприклад, в стандартний клас-список можна заносити тільки значення типів-посилань, тобто тільки об’єкти. Також класи-обгортки надають ряд нових можливостей: зокрема – містять наступні константи і методи:

1) Спеціальні значення речових типів float і double (–∞, ∞, і «не число» NaN) доступні у вигляді констант в класах Double і Float:

```
Double.POSITIVE_INFINITY,   // Плюс нескінченність
Double.NEGATIVE_INFINITY,   // Мінус нескінченність
Double.NaN;                 // “Не число”
```

Клас Float також має схожий набір констант.

Значення Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY можуть з'являтися в програмі в результаті переповнення типу double, а відповідні константи потрібні для перевірки таких ситуацій. Обидва значення «нескінченності» інтерпретуються як звичайні числа і можуть брати участь в арифметичних операціях та операціях порівняння: при цьому Double.POSITIVE_INFINITY вважається більше будь-якого іншого значення типу double, а Double.NEGATIVE_INFINITY – менше будь-якого іншого значення типу double.

Значення Double.NaN може з'являтися в програмі в результаті обчислення виразів, результат яких не визначений, наприклад ділення (0.0 / 0.0), або корінь з від'ємного числа Math.sqrt(–1). Це значення має наступні спеціальні властивості:

- NaN не дорівнює жодному іншому значенню (навіть самому собі); відповідно, найпростіший метод перевірки результату на NaN – це порівняння отриманої величини із самою собою.
- Будь-яка нетривіальна операція, яка бере NaN як операнд, завжди повертає NaN незалежно від значення інших операндів.

2) Мінімальні нормалізовані додатні значення, які приймають типи дійсних чисел float і double, наприклад:

```
Double.MIN_NORMAL;  // = 2-1022 ~= 2.2E-308
```

3) Границі (мінімальні і максимальні значення), визначені для кожного цілочислового типу – також у вигляді констант, наприклад:

```
Byte.MIN_VALUE   = -128;
Byte.MAX_VALUE   = 127;
```

4) Методи для визначення, чи є величина скінченною. Саме цими методами необхідно користуватися для перевірки значення на NaN або на нескінченність (нагадаємо, що порівняння виду `variable = Double.NaN` не має сенсу і завжди має результат false):

```
static boolean Double.isNaN(double v);
static boolean Double.isInfinite(double v);
```

5) Константи для визначення довжини представлення типа в бітах, наприклад:

```
static int Integer.SIZE  = 32;
static int Double.SIZE  = 64;
```

6) Методи, що створюють екземпляр класу-оболонки із значення примітивного типу (ці методи в виразах викликаються автоматично), наприклад:

```
static Double Double.valueOf(double d);
static Integer Integer.valueOf(int i);
```

7) Перетворення числа в рядок (також у виразах виконується автоматично):

```
static String Double.toString(double d);
static String Integer.toString(int i);
```

8) Перетвоорення рядка в число, наприклад:

```
static Double Double.valueOf(String s);
static Double Double.parseDouble(String s);
static Integer Integer.valueOf(String s)
static Integer Integer.parseInt(String s)
```

Приклад. Використання констант и методів класів Double та Integer.

```
System.out.println("inf = " + Double.POSITIVE_INFINITY );
System.out.println("MIN_DOUBLE = " + Double.MIN_NORMAL);
System.out.println("SIZE of int = " + Integer.SIZE);
double d  = Double.valueOf("123") + Double.parseDouble("12.5");
System.out.println("123 + 12.5 = " + d );   // Неявно викликається Double.toString()
```

Результат виконання цього коду буде наступним:

```
inf = Infinity
MIN_DOUBLE = 2.2250738585072014E-308
SIZE of int = 32
123 + 12.5 = 135.5
```

Перетворення типів

В Java, як і в мові С, перетворення типів при обчисленні виразів можуть виконуватися або автоматично, або за допомогою оператора приведення типу, який має вигляд:

```
(новий_тип) значення
```

Однак правила приведення типів в Java більш суворі, ніж в мові С/С++.

Автоматичне перетворення відбувається при виконанні присвоювання і арифметичних операцій між *сумісними типами*. Дозволені лише *розширюючі* автоматичні перетворення (widening conversion) в напрямку:

byte→short→int→long→float→double.

Автоматичні перетворення в зворотному напрямку (тобто *звужуючі*) заборонені, щоб не виникало неявної втрати даних.

Також необхідно пам'ятати, що типи char і boolean не сумісні з числовими типами.

Результат будь-якої операції над значеннями byte, short завжди приводиться до типу int. У цьому легко переконатися на простому на прикладі:

```
byte a = 1, b = -4;
short c = a + b;        // Помилка – спроба перетворення int->short
```

Тут у другому рядку результат складання a і b має тип int, який, в свою чергу, не може бути автоматично приведений до більш «вузького» типу short.

Операція приведення типу дозволяє виконувати звужуючі перетворення (narrowing conversion) в напрямку

byte←short←int←long←float←double,

а також перетворення між типом char і числовими типами.

Перетворення між типом boolean і числовими типами виконати неможливо.

Наприклад:

```
int i,b = 123;
long l = 38388;
i = l;              // Помилка – звужуюче перетворення повинне бути явним
i = (int) l;        // Вірно, явне звужуюче перетворення
char a = b;         // Помилка – немає автоматичного перетворення int->char
System.out.println( (char)b );    // Вірно, виведе символ "}"
System.out.println( (char)1234 ); // Недрукований символ (виводиться '?')
boolean k = (boolean)i;          // Помилка – такого перетворення немає
```

Операції над примітивними типами

Операції над примітивними типами в Java також багато в чому повторюють набір операцій мов С/С++, але є й відмінності. Розглянемо ці операції по групах:

1) Логічні операції для типу `boolean`. На відміну від С, є дві форми логічних операцій:

1. `!, &, |, ^` – повна форма, в якій обов'язково обчислюються всі операнди;
2. `&&, ||` – скорочена форма, в якій другий операнд обчислюється тільки якщо від нього залежить результат операції. Наприклад, у виразі `(2>10)&&(a>0)`

обраховувати другий операнд немає сенсу, оскільки вже перший операнд `(2>10)` визначає результат логічного «і», незалежно від значення змінної a.

2) Арифметичні операції для числових типів: `+,-,*,/,%,++,--`. Операції `++`, `--` мають префіксну і постфіксну форми, які діють аналогічно до мови C/C++.

3) Операції порівняння для числових типів: `==, !=, >, >=, <, <=`.

4) Побітові операції для цілих типів: `~, &, |, ^`.

5) Побітові зсуви <<, >>, а також беззнаковий зсув >>>. Обидві операції >> і >>> виконують побітовий зсув вправо, але відрізняються своєю поведінкою з від'ємними числами. Операція >> при зсуві копіює знаковий біт числа в молодші розряди (при цьому число залишається від'ємним), а операція >>> заповнює старші розряди нулями (у результаті число стає додатним). Таким чином, операцію >> можна використовувати для ділення на 2, а операцію >>> – ні.

6) Операції звичайного присвоєння (=) і комбінованого присвоєння: `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= ; >>>=`.

Результатом присвоювання у всіх випадках є нове значення лівої частини, тобто можливі ланцюжки виду `a=b=c`.

При комбінованому присвоєнні приведення типу виконується автоматично, наприклад:

```
byte b = 1;
b = b + 10; // Помилка! Спроба неявного перетворення із  int в byte
b += 10; // Правильно!
```

7) Умовна операція: `умова ? вираз1 : вираз 2`, наприклад

```
c = (b != 0 ? a/b : 0 );
```

У всіх операціях, крім присвоювання, операнди обчислюються зліва направо.

Пріоритет операцій визначається наступною послідовністю (від вищого до нижчого):

1)	Спеціальні операції	( ) [ ] .
2)	Унарні	~ !
3)	Арифметичні	* / %
4)	Арифметичні	+ –
5)	Зсуви	>> >>> <<
6)	Порівняння	> >= < <=
7)	Порівняння	== !=
8)	Побітові (по спаданню пріоритету)	& ^ |
9)	Логічні (по спаданню пріоритету)	&& ||
10)	Умова	?:
11)	Присвоєння	=, op=

Змінні

Як вже згадувалось, в Java всі типи змінних можна розділити на кілька великих груп (табл. 2.2).

Таблиця 2.2. Порівняння різних типів змінних Java.

Тип змінної	**Можливі значення**	**Значення за замовчуваням**
Примітивний	Визначаються границями типу	false, '\u0000', 0
Клас	null (нульове посилання), або посилання на об'єкт	null

	даного класу чи його спадкоємця	
Інтерфейс	null, або посилання на об'єкт будь-якого класу, що реалізує даний інтерфейс	null
Масив	null, або посилання на масив об'єктів відповідного типу	null
Object	null, або посилання на *будь-який* об'єкт	null

При цьому класи, інтерфейси, масиви відносяться до типів-посилань і є спадкоємцями класу Object.

Змінні примітивних типів

У багатьох мовах програмування, в залежності від області видимості, змінні поділяються на локальні і глобальні. Java має тільки один вид змінних – *локальні змінні*. ***Час життя*** будь-якої змінної в Java визначається правилом:

> *Змінна створюється в точці її опису і існує до моменту закінчення того блоку, в якому знаходиться її опис.*

Областю видимості змінної (scope) є фрагмент коду від точки її опису до кінця поточного блоку. Тобто час життя змінних в Java збігається з їх областю видимості, з урахуванням відмінності самих цих понять.

У кожному блоці може бути свій власний набір локальних змінних. При цьому у внутрішньому блоці неможна оголошувати змінні з таким же ім'ям, як і у зовнішньому по відношенню до нього блоці. Наприклад:

```
class myClass {
   int myVar=3;           // Оголошуємо змінну myVar
   public static void main (String args []){
      int myVar=5;        // Помилка! Така змінна вже існує
} }
```

Змінні-посилання

В Java для маніпулювання об'єктами в програмному коді використовуються ***посилання*** на об'єкти. Посилання зберігає в собі адресу деякого об'єкту в оперативній пам'яті. Об'єкти в програмі доступні тільки через посилання, але не може існувати посилання на примітивний тип. Клас об'єкта також є і типом посилання. Однак поняття «посилання» і «об'єкт» не рівнозначні. Так, може існувати кілька посилань на один об'єкт. На якийсь об'єкт може взагалі не бути посилань (наприклад, ми вийшли за межі області видимості всіх посилань на даний об'єкт), і тоді він для нас безповоротно втрачений. Також існує константа null – це порожнє посилання, яке не пов'язане з жодним об'єктом.

Посилання в Java в корені відрізняються від *покажчиків* в C++. Не можливо створити посилання на якусь довільну область пам'яті – за виконанням цього обмеження жорстко стежать як компілятор Java, так і JVM. Посилання може або бути пов'язане з деяким об'єктом, або дорівнювати null. При цьому компілятор і JVM також жорстко стежать за відповідністю типу об'єкта і типу посилання – в результаті можливості перетворення типів-посилань дуже обмежені. Немає ніякого способу перетворити посилання класу MyClass на посилання класу OtherClass, якщо OtherClass не є предком MyClass в ієрархії успадкування. З іншого боку, оскільки клас Object знаходиться на самій вершині ієрархії класів, то будь-яке посилання може бути перетворене на типу Object.

Всі об'єкти в Java створюються тільки явно, для чого використовується операція **new**.

```
MyType ref = new MyType();  // В дужках – параметри конструктора
```

Оператор new створює сам об'єкт і посилання на нього, причому його результатом є саме посилання.

Оскільки об'єкти доступні в програмі тільки через посилання на них, то ***область видимості об'єкта*** визначається загальною областю видимості всіх посилань на цей об'єкт. ***Час життя об'єкта*** тоді визначається наступним правилом:

Об'єкт існує, доки існує хоча б одне посилання на цей об'єкт.

Це правило, однак, не стверджує, що об'єкт буде знищений, як тільки зникне останнє посилання на нього. Просто такий об'єкт стає недоступним і може бути знищений без всяких небажаних наслідків для програми. Явно звільняти пам'ять в Java не потрібно. Об'єкти знищуються (або утилізуються) ***збирачем сміття*** (garbage collector) – окремим процесом, який завжди працює у фоновому режимі паралельно з Java-програмою.

Для пояснення сказаного розглянемо приклад:

```
SomeType globalReference = null;
{
  SomeType localReference = new SomeType();
  globalReference = localReference;
}
// ...
```

Тут всередині блоку створюється об'єкт класу SomeType, і посилання на цей об'єкт заноситься в локальну змінну localReference. Після цього те ж посилання з localReference копіюється в globalReference. При виході з блоку {} змінна-посилання localReference знищується, але змінна globalReference продовжує існувати. Відповідно, продовжує існувати і створений об'єкт.

Фінальні змінні

Змінні, оголошені з модифікатором **final**, називаються «фінальними» і відрізняються від звичайних тим, що можуть отримати своє значення тільки один раз при ініціалізації. При цьому ініціалізація може відбуватися як в момент оголошення змінної, так і пізніше, окремим оператором, наприклад:

```
final int i = 5;        // Ініціалізація при оголошенні
final String s;
s = "ddddd";            // Ініціалізація окремо від оголошення
i = 10;                 // Помилка! Надати інше значення не можна
```

Примітка. Ініціалізація фінальних членів класу повинна проводитися або при оголошенні, або в конструкторі. Ініціалізація фінальних статичних змінних класу повинна проводитися тільки при оголошенні (насправді також можливо надати їх значення в так званому статичному блоці ініціалізації, але використання цієї можливості виходить за рамки даного курсу).

Все це зовсім не означає, що фінальну змінну взагалі не можна змінити. Для змінних-посилань обмеження `final` полягає лише в тому, що в змінній не можна зберегти посилання на інший об'єкт. Але в той же час через фінальну змінну-посилання можна викликати будь-які методи пов'язаного з нею об'єкта, навіть якщо вони змінюють стан або внутрішні дані об'єкту, наприклад:

```
final SomeClass obj = new SomeClass();      // Ініціалізація при оголошенні
obj.change();                               // Виклик методу,
                                            // що змінює об’єкт
```

Статичні змінні

Статичними (**static**) в Java оголошуються тільки члени класу. Значення статичної змінної повинно бути спільним для всіх об'єктів класу, в якому вона оголошена. Звичайні змінні оголошені в класі (без модифікатора `static`) прийнято називати ***змінними реалізації (instance variables)***, тому що в кожному об'єкті класу (тобто в кожній його реалізації) знаходиться своє

значення такої змінної. Аналогічно, оскільки значення статичної змінної відноситься не до конкретного об'єкта (реалізації), а до класу в цілому, такі змінні називають ***змінними класу (class variables).***

Змінні, оголошені всередині методів, не можуть бути статичними, тобто модифікатор `static` не застосовується до локальних змінних. Більш докладно про змінні класу йтиметься в наступній лекції.

Масиви

Кожен масив в Java є об'єктом, а змінні типу масив є *посиланнями*. Масив створюється в три етапи:

1) ***Оголошення*** змінної – посилання на масив, наприклад:

```
double[] a, b;// Дві посилання на масиви
double c[], d;// Посилання  на масив a, і число d
```

2) ***Виділення пам'яті*** для масиву, в результаті чого в змінну заноситься адреса реального масиву. Цей крок можна робити одночасно з оголошенням масиву або окремо.

```
double a[];
a = new double[5];
double c[] = new double[5];
```

Після цього кроку до елементів масиву можна звертатися за допомогою операції **[ ]**. Індексація завжди починається з 0. Якщо індекс виходить за межі масиву, генерується виключення `IndexOutOfBoundsException`, але такій ситуації необхідно запобігати вже на етапі написання програми.

Розмір масиву зберігається у властивості `length`. Використовувати цю властивість завжди краще, ніж підставляти в код число або константу.

3) ***Ініціалізація*** елементів масиву. Існує два способи ініціалізації масиву:

а) ініціалізація при створенні масиву:

```
double b[] = new double[] {0.1, 0.2, -0.3, 0.45, -0.02};
double[] d = {0.01, -3.4, 2.89, 4.5, -6.7};
```

б) звертання до елементів масиву, наприклад:

```
double a[] = new int[5];
for (int i = 0; i < a.length; i++) a[i] = i*i;
```

Також можна створити *безіменний масив*, використовуючи результат операції `new` в виразі, наприклад:

```
System.out.println (new char [] {'H', 'e', '1', '1', 'o'});
```

Такий безіменний масив завжди повинен буди ініціалізований при створенні, оскільки після його створення не залишається посилання на масив.

Масиви об'єктів

Масив об'єктів – це масив посилань на об'єкти. Відповідно, потрібно створити як масив, так і самі об'єкти. І те й інше створюється за допомогою оператора **new**. Нехай у нас є певний клас SomeClass і потрібно побудувати масив з 4-х об'єктів цього класу.

```
// Варіант 1. (явне занесення об'єктів в масив)
SomeClass arr[] = new SomeClass[4];         // Створення масива
for (int j = 0; j < 4; j++ )                // Ініціалізація масива
     arr[j] = new SomeClass();              // Ініціалізація елементів

// Варіант 2. (використання списка ініціалізації)
```

```
SomeClass arr[] = new SomeClass[] {              // Створення масива
          new SomeClass(), new SomeClass(),      // Ініціалізація елементів
          new SomeClass(), new SomeClass()
     };
```

Наприклад, створимо масив із двох рядків і зразу ж його ініціалізуємо:

```
String s[] = {new String("aaa"), "bbb"};
```

Багатовимірні масиви

В Java багатовимірні масиви як окремий тип даних відсутні. Але логічно багатовимірний масив можна організувати, якщо в звичайному масиві кожен елемент є посиланням на масив меншої розмірності. Наприклад, створимо двовимірний масив цілих чисел розміру 3 * 3.

```
// Вариант 1. Просте створення масиву
int ar[][] = new int[3][3];           // Далі необхідна ініціалізація!
for(int i=0; i<ar.length; i++ ) {     // Ініціалізація
     for(int j=0; j<ar[i].length; j++) ar[i][j] = i+j;
}

// Варіант 2. Використання списка ініціалізації
int ar1[][] = new int[][] {  {1, 2, 3}, {4, 5, 6}, {7, 8, 9}  };
```

Для ініціалізації багатовимірного (як і одновимірного) масиву об'єктів кожен об'єкт повинен бути створений окремо (за допомогою `new`) і записаний в масив. Це можна зробити в списку ініціалізації або окремим циклом.

Оскільки вкладені масиви є об'єктами, вони також можуть створюватися окремо, і навіть мати різні розміри, наприклад:

```
int ma[][] = new int[4][];       // Створюємо масив посилань на масиви,
                                 // але не створюємо самі вкладені масиви
for(int i=0; i<ma.length; i++ ) {  // Для кожного елемента верхнього рівня
     ma[i] = new int[i];         // створюємо посилання на  масив-рядок
     for(int j=0; j<i; j++) ma[i] = i+j; // Ініціалізуємо масив-рядок
}
```

Операції над посиланнями і масивами

Над масивами, як і над іншими змінними-посиланнями (екземплярами класу `Object`) можна виконувати наступні загальні операції:\

1) ***Присвоєння*** a = b, яке є копіюванням посилання. В результаті обидва посилання a й b вказують на один і той самий об'єкт (або масив) в пам'яті.

2) ***Порівняння*** на рівність і нерівність за допомогою операцій = =,! =, а також методу

```
boolean Object.equals (Object obj);
```

Операції == і != зіставляють адреси, що містяться в посиланнях, а не вміст об'єктів (або масивів), тобто ми можемо дізнатися, чи не посилаються обидва посилання на один і той самий об'єкт в пам'яті. Поведінка методу `equals()` різниться для різних класів, наприклад для `String` метод `equals()` порівнює вміст рядків, а для масивів – цей же метод порівнює лише посилання, тобто працює аналогічно до порівняння ==.

Проілюструємо роботу присвоювання і обох способів порівняння на прикладі:

```
String s = new String("abc");   // Рядок
String s1 = s;                  // Копія посилання
String s2 = new String("abc");  // Іще один такий самий рядок
System.out.println( s == s1 );  // Виведе true, оскільки посилання рівні
System.out.println( s == s2 );  // Виведе false, оскільки посилання різні
```

```
System.out.println( s.equals(s2) );  // Виведе true, т. я. рядки однакові
```

Примітка. Рядок можна ініціалізувати рядковим літералом, наприклад:

```
String s3 = "abc";
```

При цьому також створюється об'єкт-рядок, і посилання на нього заноситься в змінну s3. Але якщо створити ще однин такий самий рядок:

```
String s4 = "abc";
```

компілятор оптимізує наш код, і не буде створювати ще один зайвий об'єкт, а замість цього збереже в s4 те саме посилання, що і в s3. Тому при порівнянні посилань (s3 == s4) вони виявляться рівними.

3) ***Створення копії*** об’єкта чи масиву за допомогою методу **Object.clone()** :

```
int a[] = {1,2,3}, b[];
b = a;    a.equals(b);                 // true
c = a.clone();  a.equals(c);           // false
```

Важливо пам’ятати, що при цьому якщо в масиві зберігаються посилання, то об'єкти-елементи *не копіюються*, а в новий масив вносяться лише копії посилань.

Цих загальних методів класу `Object` абсолютно недостатньо для повноцінної роботи з масивами. Тому в стандартному класі **java.util.Arrays** реалізовані додаткові методи обробки масивів. Розглянемо їх на прикладі:

```
import java.util.*;  // Підключити пакет java.util
// ...
     int a[] = new int[1000];
     // Заповнити випадковими числами в діапазоні [10..99]
     for(int i = 0; i<a.length; i++) a[i] = (int)(Math.random()*90) + 10;

     // Заповнити елементи від 2 до 5 (не включно) значенням -1
     Arrays.fill(a, 2, 5, -1);

     // Створити копію діапазону елементів від 0-го до 10-го (не включно)
     // і перетворити отриманий масив на рядок
     int b[] = Arrays.copyOfRange(a, 0, 10);
     System.out.println( Arrays.toString(b) );

     // Перевірити рівність всіх елементів для двох масивів
     Arrays.equals(Arrays.copyOfRange(a, 2, 5), new int [] {-1,-1,-1});

     // Відсортувати масив методом QuickSort
     Arrays.sort(a);  System.out.println( Arrays.toString(a) );

     // Знайти елемент у відсортованому масиві методом бінарного пошуку
     // Результат - індекс знайденого елемента або число <0
     int pos = Arrays.binarySearch(a, 20);
     System.out.println(  pos>=0? pos : "none" );
```

Всі наведені методи працюють з **одновимірними** масивами. Масив більшої розмірності вони розглядають як одновимірний масив посилань. Наприклад, код:

```
System.out.println(Arrays.toString( new int[][] {{1},{2,3}} ));
```

виведе три посилання на масиви-рядки, а не елементи 1, 2, 3:

```
[[I@8965fb, [I@867e89, [I@1dd7056]
```

Тому для роботи з багатовимірними масивами клас **java.util.Arrays** має спеціальні методи **Arrays.deepEquals()** і **Arrays.deepToString(),** які діють аналогічно Arrays.equals() і Arrays.toString(), але рекурсивно обробляють всі вкладені масиви.

Повний перелік методів класу `Arrays` і їх опис можна знайти в документації.

Порівняння різних способів копіювання і порівняння масивів.

При копіюванні і порівнянні об'єктів і масивів завжди необхідно чітко розуміти, що саме буде робити кожен конкретний метод. Якщо цих тонкощів не враховувати, то з'являються дуже важко вловимі помилки. Щоб закріпити тонкощі копіювання і порівняння в Java, спробуйте самостійно заповнити наступні дві таблиці (табл. 2.3, 2.4). Зверніть особливу увагу на випадки, де *результат може залежати від типу і розмірності масиву*.

Таблиця 2.3. Різні способи порівняння для масиву рядків: `String a[] = {"1","2"};`
і для масиву чисел: `int a[] = {1, 2};`.

B	b==a	b.equals(a)	Arrays. equals(a,b)	Arrays. deepEquals(a,b)	b[0]==a[0]
b= a	true	true			
b= a.clone()					
b= Arrays. copyOf(a,a.length)					
b={"1","2"}					

Таблиця 2.4. Різні способи порівняння для двовимірного масиву рядків:
`String a[] = { {"1","2","3"}, {"1","2"} };`.

b	b==a	b.equals(a)	Arrays. equals(a,b)	Arrays. deepEquals(a,b)	b[0]==a[0]
b= a	true	true			
b= a.clone()					
b= Arrays. copyOf(a,a.length)					
b={{"1","2",…},…}					

Передача параметрів до методів

В Java існує тільки один тип передачі параметрів – ***за значенням***. Це означає, що під час виклику методу йому передається *копія* поточного значення параметра, яка знищується при виході з методу.

При передачі параметра типу посилання створюється *копія посилання*, але не самого об'єкта. Подивимося більш детально, що це означає. Нехай в метод method() передаються посилання на об'єкт ref:

```
public static void main(String[] args) {
     SomeClass ref = new SomeClass();
     method(ref);
}
```

Всередині методу method() ми можемо змінити посилання ref (тобто присвоїти йому посилання на інший об'єкт), але це ніяк не вплине на посилання ref у методі main(). Проте якщо всередині method() змінити сам об'єкт, викликавши який-небудь його метод ref.change(), то ця зміна вплине на об'єкт ref у методі main().

Примітка. Константні посилання і підтримка так званих *const correctness* в Java відсутні. Це означає, що при оголошенні методу неможливо «заборонити» йому змінювати об'єкти через свої параметри-посилання.

Приклад. Маємо методи, що змінюють значення своїх параметрів:

```
static void change(String s) {  s = "bbb"; }
```

```
static void change(int a[]) {
     if(a!=null) for(int i = 0; i<a.length; i++ ) a[i] = 0;
}
static void change(int a[][]) {
     if(a!=null)
          for(int i = 0; i<a.length; i++ )
               for(int j =0; j<a[i].length; j++ ) a[i][j] = 0;
}
```

Також маємо локальні змінні:

```
String s = "aaa";
int x[] = {1,2,3}, y[][] = {{1,2,3},{4,5,6},{7,8,9}};
```

Тоді в результаті викликів:

```
change(s); change(a); change(y);
```

Рядок s залишиться незмінним, оскільки в метод change() передається копія посилання s, і ця копія отримує нове значення (тобто посилання на інший рядок).

Вміст масивів x і y зміниться, оскільки при виклику методу change() вміст цих масивів не буде копіюватися і доступний зсередини методу change() за посиланням.

Під час викликів

```
change(x.clone()); change(y.clone());
```

явно створюються копії масивів, разом із збереженими в них значеннями. Зміна елементів масиву-копії x.clone() ніяк не впливає на вміст вихідного масиву **x**. З масивом **y** все складніше, оскільки копія y.clone() – це одновимірний масив посилань на менші масиви-рядки. При цьому з масиву **y** копіюються всі посилання на рядки, але не самі масиви-рядки. Тому всередині change() копія масиву **y** містить посилання на ті ж самі об'єкти-рядки, що і сам масив **y**. Через ці посилання вміст масиву **y** змінюється зсередини методу change().

Тема 3. Класи в Java

Лекція 5

Опис класу

Загальна форма оголошення класу виглядає наступним чином:

```
модифікатори class им'я_класу
     extends ім'я_базового_класу                      // не обов'язково
     implements ім'я_інтерфейсу1, ім'я_інтерфейсу2    // не обов'язково
{
     // Оголошення змінних (полів)
     модифікатори  тип  ім'я_змінна1;
     модифікатори  тип  ім'я_змінна2 = ініц_знач;

     // Оголошення методів
     модифікатори тип ім'я_методу1(список_параметрів) { тіло_методу; }

}
```

Опис класу починається з ключового слова `class`, після якого записується ім'я класу. Загальноприйняті угоди (Code Conventions) рекомендують починати ім'я класу з великої літери, на відміну від імен методів і змінних, які починаються з маленької літери.

Перед словом `class` можуть знаходитись *модифікатори* класу (class modifiers) – ключові слова `public`, `abstract`, `final`, `strictfp` – які змінюють властивості класу. Ці модифікатори обговорюються далі в цій лекції.

Тіло класу виділяється фігурними дужками і містить всі змінні (поля), методи, вкладені класи і інтерфейси.

Оголошення *змінної* (поля) в класі відбувається так само, як і всередині методу: вказується її тип, ім'я та, можливо, початкове значення. Оголошення змінної може починатися з одного або декількох необов'язкових модифікаторів: `public`, `protected`, `private`, `static`, `final`, `transient`, `volatile`. З модифікаторами ми будемо знайомитися в міру необхідності.

При описі *методу* вказується тип значення, що повертається, або слово `void`, потім, через пробіл, ім'я методу, потім, в дужках, список параметрів. Після цього в фігурних дужках пишуться оператори, які повинні виконуватися, коли викликається метод.

Опис методу може починатися з модифікаторів `public`, `protected`, `private`, `abstract`, `static`, `final`, `synchronized`, `native`, `strictfp`. Їх ми також будемо розглядати по необхідності. У списку параметрів через кому перераховуються тип і ім'я кожного параметра. Список параметрів може бути відсутнім, але дужки зберігаються.

Ключове слово **extends** служить для *успадкування* існуючого класу. У Java заборонено *множинне успадкування*. Будь-який клас неявно успадковує клас **Object.**

Модифікатори класу

При оголошенні класу можливо використання наступних модифікаторів:

- **public** – відкритий клас, який доступний ззовні пакета, в якому він визначений. Такий клас повинен бути оголошений у файлі .java, ім'я якого збігається з ім'ям класу. За замовчанням (якщо модифікатор `public` не вказано) клас доступний тільки зсередини свого пакета. *Пакетом* в Java називається іменована група файлів. Використання пакетів ми розглянемо в наступній лекції.
- **final** – фінальний клас, успадкування якого заборонено. Наприклад, фінальними є багато класів стандартної бібліотеки (такі як System, String).
- **abstract** – абстрактний клас, в якому хоча б один метод не реалізований.

При оголошенні вкладеного класу можна також використовувати модифікатори `protected`, `private`, `static`.

Модифікатори способу доступу

Java підтримує 4 різних рівня доступу, але має всього 3 модифікатори способу доступу. Ці модифікатори вказуються для кожної змінної (поля) в класі і методу окремо і дозволяють задавати 4 варіанти прав доступу:

- **public** – означає, що даний елемент доступний без будь-яких обмежень;
- **private** – доступ дозволений тільки з даного класу;
- **protected –** доступ дозволений з даного класу і з усіх класів-нащадків, а також з усіх класів даного пакету.
- *без модифікатора* – якщо жоден з модифікаторів public, private, protected не вказано, то доступ дозволений з усіх класів даного пакету.

Методи

Як і C++, Java підтримує ***перевантажені методи***. Вимоги до перевантаження методів ті ж.

Всі методи, крім статичних і конструкторів, є ***віртуальними***. Тому вони можуть бути перевизначені в похідних класах.

За аналогією зі змінними нестатичні методи називаються також *методами реалізації*, а статичні – *методами класу*.

У всіх методах реалізації можна використовувати ключове слово **this** – воно позначає посилання на поточний об'єкт (для якого викликаний даний метод). Зазвичай посилання this використовується в двох випадках:

1) якщо метод повинен повернути посилання на об'єкт, для якого він був викликаний (`return this;`), наприклад:

```
class SomeClass {

     int state = 0;                        // Стан об’єкта
     // ...
     SomeClass change( int newState ) {  // Метод, що змінює стан об’єкта
          state  =  newState;              // Змінити поточний об’єкт
          return this;                     // Повернути посилання на нього
     }
}

public static void main(String[] args) {
     SomeClass obj = new SomeClass();     // Використовуємо клас SomeClass
     obj.change(1).change(2);             // Перший виклик change() повертає
                                          // посилання на об’єкт, який можна
                                          // далі використовувати
}
```

2) для звертання до полів об’єкта, якщо їх імена співпадають з іменами формальних параметрів методу, наприклад:

```
class SomeClass {

     int state = 0;                        // Стан об’єкта
     // ...
     void change( int state ) {           // Ім’я параметра співпадає з іменем поля
          this.state  =  state;           // Зліва – поле; зправа - параметр
     }
}
```

Ключове слово **super** можна використовувати в методах реалізації довільного класу (не статичних) для посилання на змінні (поля) і методи базового класу. Наприклад:

```
class My {                    // extends Object
     public String toString() { return "My : " super.toString(); }
     public out() { System.out.println( super.toString); }
}
```

Конструктори класу

Конструктор класу (class constructor) – це спеціальний метод, ім'я якого співпадає з іменем класу, і який відповідає за створення об'єктів класу.

Особливості конструктора:

- Конструктор є в будь-якому класі. *Конструктор за замовчанням* (default constructor) створюється автоматично, але такий конструктор не робить нічого, крім виклику конструктора базового класу. Нагадаємо, що за замовчанням всі класи успадковуються від Object.
- Конструктор виконується автоматично при створенні екземпляра класу.
- Конструктор не повертає ніякого значення. Тому в його описі не пишеться навіть слово `void`. У той же час конструктор може мати модифікатор способу доступу (`public`, `protected` або `private`).
- Конструктор не є віртуальним методом, тому його не можна успадковувати або перевизначити в похідному класі.
- Тіло конструктора, крім звичайних операторів, може починатися:
  - з виклику одного з конструкторів базового класу, для цього записується слово `super()` з параметрами в дужках, якщо вони потрібні;
  - з виклику іншого конструктора того ж класу, для цього записується слово `this()` з параметрами в дужках, якщо вони потрібні.
- У класі може бути кілька перевантажених конструкторів, які відрізняються типом та/або кількістю параметрів.

У всьому іншому конструктор подібний до звичайного методу: в ньому дозволяється записувати будь-які оператори, навіть оператор `return`, але тільки порожній, без всякого значення, що повертається.

При ***створенні об'єкта*** (наприклад, `new MyClass(prm)`) відбуваються наступні дії:

- виділяється пам'ять під об'єкт і створюється посилання на нього;
- викликається конструктор базового класу (якщо явного виклику `super()` немає, то автоматично викликається конструктор базового класу без параметрів);
- поля (змінні реалізації) об'єкта похідного класу ініціалізуються значеннями, зазначеними при оголошенні (за замовчанням – нулями);
- виконується тіло конструктора похідного класу (якщо воно є);
- створене посилання на об'єкт використовується у виразі, наприклад присвоюється змінній.

Таким чином, якщо в класі не оголошено жодного конструктора, і не вказано базовий клас (`extends`), то при створенні екземплярів викликався конструктор без параметрів класу Object.

Приклад. Клас дробів, що має кілька конструкторів. Змінні реалізації цього класу зберігають чисельник і знаменник дробу, а конструктори ініціалізують ці змінні.

```
public class Ration
     // extends Object                 // Наслідується за замовчанням
{
     protected int num = 0, den = 1;  // Змінні реалізації
     Ration() {}                      // Конструктор за замовчанням нічого не робить
     Ration(int num, int den) {       // Конструктор з параметрами
          // super();                 // Тут можна викликати конструктор базового
класу
          this.num = num;
          this.den = den==0? 1 : den;
     }
     public Ration( Ration r ) {      // Копіюючий конструктор
          this( r.num, r.den );       // просто викликає конструктор з параметрами
     }
```

```
    public String toString() {       // Перетворення дробу на рядок
        return "" + r.num + "/" + r.den;
    }

    // Створення об'єктів-дробів
    public static void main(String[] args) {
        Ration r1 = new Ration();       // Викликано конструктор без параметрів
        Ration r2 = new Ration(1,2);    // Викликано конструктор з параметрами
        Ration r3 = new Ration(r2);     // Створено копію r2 з допомогою
                                        // копіюючого конструктора
        System.out.println(r3);         // Виводимо "1/2"
    }
}
```

У цьому прикладі реалізований один з методів класу `Object – toString()`. Він не має параметрів і викликається автоматично всякий раз, коли необхідно перетворити об'єкт нашого класу на рядок. Наприклад в виразі `System.out.println(r3);` спочатку об'єкт `r3` приводиться до типу `String`, а потім викликається метод `ptrintln()`.

Метод finalize()

Метод **finalize()** визначений в класі `Object` і успадковується всіма іншими класами. Цей метод для кожного об'єкта викликається автоматично перед утилізацією цього об'єкта збирачем сміття. Зазвичай в методі `finalize()` звільняються ресурси, зайняті об'єктом (файли, сокети і т. п.). Як типовий приклад, розглянемо клас, який в конструкторі відкриває файл, і в методі `finalize()` звільняє цей файл.

```
class My {
    FileInputStream aFile = null;         // Потік читання із файла

    My(String filename) {                 // Конструктор
        try {
            aFile = new FileInputStream(filename);  // Відкриття файла
        } catch (java.io.FileNotFoundException e) {
            System.err.println("Could not open file " + filename);
        }
    }

    protected void finalize() throws Throwable {
        // Обов'язково викликаємо finalize() для базового класу, інакше об'єкт
        // може некоректно видалитись
        super.finalize();
        // Звільняємо всі ресурси, виділені в похідному класі
        if (aFile != null) aFile.close();
    }
}
```

Модифікатор static

Змінні і методи, які оголошені без модифікатора **static**, називають ***змінними (методами) реалізації (instance variables)***.

Змінні і методи, оголошені як **static**, називають ***змінними (методами) класу (class variables).*** Статичний метод може звертатися тільки до статичних полів і методів.

Статичними оголошуються змінні і методи, які не залежать від конкретного об'єкта. Тому до статичних змінних і методів можна звертатися, не створюючи жодного екземпляра класу. Наприклад, ми можемо користуватися методами класу `Math`, не створюючи його екземпляри, а просто записуючи `Math.abs(x)`, `Math.sqrt(x)`. З цієї ж причини метод `main()` завжди оголошується статичним. Також ми багато разів використовували статичну змінну `out` в класі `System`, наприклад: `System.out.println()`.

Приклад. Для створеного раніше класу дробів `Ration` реалізуємо метод складання дробів двома способами: методом класу і методом реалізації.

```
public Ration add(Ration b) {          // Метод реалізації
     den = den * b.den;                // Змінюємо поточний об’єкт (this)
     num = den*b.num + num*b.den;
     simpl();                          // Спрощуємо дріб
     return this;                      // Повертаємо посилання на об’єкт
}

public static Ration add(Ration a, Ration b) {          // Метод класу
     // Створюємо новий об’єкт як копію a, змінюємо і повертаємо його
     return new Ration(a).add(b);
}

// ...

public static void main(String[] args) {
     Ration r1 = new Ration(1,2);
     Ration r2 = new Ration(3,2);
     Ration r3 = new Ration(1/6);
     // Визиваємо метод реалізації (від об’єкта r3)
     System.out.println( r3.add(r1) );  // Змінюється r3 і виводиться “2/3”
     // Викликаємо метод класу (від самого класу)
     System.out.println( Ration.add(r1, r2) );    // r1 не змінюється,
                                                  // і виводиться “2/1”
}
```

Для того, щоб статичний метод не залежав від об'єкта класу, він повинен відповідати кільком вимогам:

- в статичному методі не можна використовувати посилання `this` і `super`;
- в статичному методі не можна використовувати нестатичні поля і методи;
- статичні методи не можуть бути абстрактними;
- статичні методи можуть бути перевизначені в похідних класах тільки як статичні.

Модифікатор final

Цей модифікатор, в залежності від того, де він використаний, може мати різний зміст. Коротко розглянемо всі можливі випадки.

1. Для змінних реалізації `final` означає, що ініціалізація змінної повинна проводитися або при оголошенні, або в конструкторі. Після ініціалізації значення фінальної змінної не може бути змінено.

2. Для змінних класу (статичних) `final` має схоже значення, але така змінна повинна ініціалізуватись тільки при оголошенні. На практиці статичні фінальні змінні використовуються для оголошення констант. Наприклад, в класі дробів може бути оголошена константа `PI` – раціональне число, близьке до реального значення π :

```
public static final Ration PI = new Ration(31415, 10000);
```

3. Для параметра методу модифікатор `final` означає, що даний параметр не можна змінювати всередині методу.

4. Для методу `final` означає, що цей метод не може бути перевизначений в похідних класах. Якщо `final` поставити перед визначенням класу, то всі методи цього класу стають фінальними.

Навіщо ж буває необхідно позначати метод або клас модифікатором `final` і тим самим забороняти їх перевизначення в підкласах? Зазвичай це робиться з метою безпеки: ви можете бути впевнені, що метод не буде перевизначений, і завжди буде виконувати саме ті дії, які ви задали. Зрозуміло, такий метод не може бути абстрактним. Наприклад, фінальними визначені класи `Math`, `String`, `Integer`. В результаті, наприклад, ми можемо бути впевнені, що метод `Math.cos(x)` обчислює саме косинус числа x, і цю поведінку методу неможливо змінити за допомогою

успадкування та поліморфізму. Для повної безпеки, поля, оброблювані фінальними методами, слід зробити закритими (`private`).

Примітка: Якщо перед оголошенням деякої змінної або методу потрібно поставити кілька модифікаторів, то специфікація мови (JLS) рекомендує перераховувати їх в наступному порядку:

```
public, protected, private, static, final, transient, volatile.
```

Успадкування і поліморфізм

В Java успадкування і поліморфізм використовуються так само, як і в інших об'єктно-орієнтованих мовах (C++, C#), тому розглянемо їх дуже коротко.

Успадкування – це відношення між класами, при якому один клас (*нащадок або підклас*) розширює функціональність іншого (*батька, суперкласу або базового класу*). Це означає, що нащадок автоматично переймає всі поля і методи батька, а також додає деякі свої.

Успадкування зазвичай виникає, коли всі об'єкти одного класу одночасно є об'єктами іншого класу (відношення *загальне-часткове*). Наприклад, всі об'єкти класу Студент є об'єктами класу Людина. У цьому випадку говорять, що клас Студент успадковується від класу Людина.

Зауважимо, що якщо клас A є нащадком класу B, а клас B є нащадком класу C, то клас A є також нащадком класу C. Наприклад, в Java всі класи безпосередньо або опосередковано успадковані від класу `Object`.

Успадкування спрощує роботу програміста. Наприклад, якщо в програмі необхідно ввести новий клас «Привілегійований користувач», то його можна створити на основі вже існуючого класу «Користувач», не програмуючи заново всі поля і методи, а лише додавши ті, яких не вистачало в базовому класі.

Для того, щоб один клас був нащадком іншого, необхідно при його оголошенні після імені класу вказати ключове слово `extends` і назву суперкласу.

Щоб не виникало неоднозначностей з успадкуванням методів, в Java заборонено *множинне успадкування.*

Приклад. На базі класу дробів Ration створимо новий клас, в якому змінимо спосіб перетворення дробу на рядок.

```
public class RationEx extends Ration
{
     // Змінні num і den наслідуються із класу Ration

     // Конструктори не наслідуються – їх треба описувати заново,
     // але можна викликати конструктор базового класу
     RationEx(int num, int den) {         // Конструктор з параметрами
          super(num, den);                // Виклик конструктора базового класу
     }
     public RationEx( Ration r ) {        // Копіюючий конструктор
          super( r );                     // Виклик конструктора базового класу
     }

     // Перевизначений метод
     @Override
     public String toString() {           // Нове перетворення дробу в рядок
          return (double)r.num / r.den;
     }
}
```

У цьому прикладі перевизначенний метод позначений анотацією `@Override`. Ця анотація не обов'язкова, але дуже корисна. Якщо її поставити, компілятор буде перевіряти правильність перевизначення (тобто чи збігається сигнатура нового методу з методом в базовому класі), і це дозволить уникнути важко вловимих помилок.

Використання поліморфізму також розглянемо на простому прикладі. Збережемо різні об'єкти класів `Ration` і `RationEx` в один масив і виведемо весь вміст цього масиву. Виводитися наші об'єкти будуть по-різному, але в коді програми про це піклуватися не потрібно.

```
public static void main(String[] args) {
     // Створюємо масив об’єктів різних класів
     // Це можна зробити тільки якщо всі вони наслідуються від класу,
     // який використано в оголошені масиву!
     Ration rr[] = { new Ration(1,2), new RationEx(1,2),
                     new Ration(3,2), new RationEx(3,2) };
     // Виводимо всі об’єкти незалежно від типу
     for (Ration r : rr) {
          System.out.println( r );
     }
}
```

Масив типу `Ration` може містити як об'єкти класу `Ration`, так і будь-якого похідного класу. У циклі ми отримуємо посилання на черговий елемент масиву типу `Ration`, але, якщо за цим посиланням знаходиться об'єкт класу `RationEx`, то для перетворення на рядок буде викликаний перевизначений метод з класу `RationEx`, в чому можна переконатися, виконавши цей приклад.

Клас Object

Як уже неодноразово говорилося, клас `Object` є коренем ієрархії класів Java. Він містить загальні методи, які застосовуються для всіх класів. Ці методи дуже важливі, тому що вони визначають основи поведінки об'єкта. Нові класи повинні перевизначити методи `Object`, якщо їх не влаштовує стандартна реалізація. Як це зробити, ми розглянемо на прикладі нашого класу дробів.

`public boolean equals(Object obj);`

Цей метод порівнює поточний об'єкт з об'єктом `obj`. Він реалізований в класі `Object` як тотожність: об'єкт дорівнює тільки самому собі. Будь-яка реалізація цього методу повинна встановлювати певне *відношення еквівалентності* об'єктів.

Наприклад, для класу `Ration` метод `equals()` можна реалізувати наступним чином:

```
@Override
public boolean equals(Object obj) {
     if (this == obj)           // Об’єкт порівнюється сам з собою
          return true;
     if (obj == null)           // Об’єкт порівнюється з с null
          return false;
     if (getClass() != obj.getClass())      // obj – класу Ration?
          return false;
     Ration other = (Ration) obj;           // Приводим тип до Ration
     if (den != other.den)                  // Тепер можна порівняти поля
          return false;
     if (num != other.num)
          return false;
     return true;
}
```

Зверніть увагу, що `obj` має тип `Object`, тому перш ніж звертатися до його полів, потрібно привести його до типу `Ration`. Однак в якості `obj` може бути переданий об'єкт абсолютно іншого класу, або `null` – обидва ці випадки потрібно заздалегідь перевірити, інакше можливі помилки.

`protected native Object clone() throws CloneNotSupportedException;`

Цей метод створює копію об'єкта. Реалізація за замовчанням *захищена* (`protected`) і ззовні класу не доступна. Так зроблено тому, що в класі `Object` нічого копіювати, і, відповідно метод `clone()` просто не потрібен.

Якщо у вашому класі метод `clone()` не перевизначений зі способом доступу `public`, то ваші об'єкти також не можна буде копіювати (при спробі виклику `clone()` буде генеруватися виключення).

Існує угода, що `clone()` повертає новий об'єкт, такий, що виконуються дві умови:

```
x.clone().getClass() == x.getClass()
x.clone().equals(x) == true
```

тобто «клон» повинен бути того ж класу, що й вихідний об'єкт і повинен бути «рівним» йому в розумінні методу `equals()`.

Друга умова означає, що при перевизначенні `clone()` також потрібно перевизначити і метод `equals()`, і так, щоб ці два метода один одному відповідали.

Приклад. Реалізація `clone()` в класі `Ration`.

```
@Override
public Object clone() {
     return new Ration(this);
}
```

```
public native int hashCode();
```

Цей метод використовується класами-колекціями (`HashMap`, `HashSet`) для отримання ***хеш-коду*** об'єкта, тобто цілого числа, яке можна використовувати для швидкого пошуку цього об'єкта. Необхідність перевизначати `hashCode()` виникає, якщо планується зберігати об'єкти в ***хеш-таблиці***.

Метод `hashCode()` завжди повинен бути узгоджений з методом `equals()` – інакше можлива втрата даних з хеш-таблиці:

- рівні об'єкти повинні повертати однакові хеш-коди;
- однак для різних об'єктів коди можуть збігатися (така ситуація називається колізією).

Сама реалізація `hashCode()` повинна відповідати таким основним вимогам – інакше зберігання об'єктів в хеш-таблиці буде неефективним:

- кожен біт даних об'єкта повинен істотно впливати на хеш-код;
- хеш-коди всіх можливих об'єктів повинні бути розподілені по всьому діапазону типу `int` рівномірно;
- кількість колізій повинна бути мінімальна.

Перевизначити `hashCode()` можна або вручну, або скориставшись засобами генерації вихідного коду в InelliJ IDEA: *Alt-Insert → Generate hashCode() and equals().*

Наприклад, для класу Ration буде згенеровано наступний код:

```
@Override
public int hashCode() {
     final int prime = 31;
     int result = 1;
     result = prime * result + den;
     result = prime * result + num;
     return result;
}
```

Тут результат формується шляхом «перемішування» бітів з двох полів об'єкта-дробу.

Якщо клас містить дійсну змінну реалізації `double d;` то в `hashCode()` будуть використані всі біти цієї змінної:

```
     temp = Double.doubleToLongBits(d);
```

```
result = prime * result + (int) (temp ^ (temp >> 32));
```

Якщо ваш клас має поля-масиви, наприклад `double[] doubles`, то можна використовувати готову реалізацію `hashCode()` в класі `Arrays`:

```
result = prime * result + Arrays.hashCode(doubles);
```

`public String toString();`

Цей метод `Object` вже розглядався. Він перетворює об'єкт на рядок. Реалізація за замовчанням формує рядок, що містить позначення типу та хеш-код об'єкта. Наприклад: `MyClass@e53108`, `[I@e53108`. У багатьох класах цей метод потрібно перевизначити.

Приклад перевизначення `toString()` в класі `Ration`:

```
@Override
public String toString() {
     return super.toString() +
          " : Ration [num=" + num + ", den=" + den + "]";
}
```

`protected void finalize() throws Throwable;`

Перевизначення цього методу було розглянуто раніше. Метод повинен звільняти ресурси, зайняті об'єктом. В нашому прикладі клас `Ration` не використовує ніяких зовнішніх ресурсів, тому він не потребує реалізації методу `finalize()`.

Рефлексія і клас Class

Змінна-посилання може містити посилання на об'єкти різних класів, пов'язаних успадкуванням. ***Рефлексією*** називається здатність об'єкта описувати самого себе, і зокрема – повідомляти свій клас. Ця здатність внесена в самий базовий клас `Object`, який має:

- Поле **`class`**
- Метод **`public final native Class getClass()`**

Метод `getClass()` повертає об'єкт типу `Class` – клас даного об'єкта.

Також для об'єктів класу **`Class`** визначені

- Оператор **`insanceof`**;
- Метод `boolean isInsance (Object obj)`,

які використовуються для перевірки приналежності об'єкта класу.

Приклад використання поліморфізма і рефлексії

```
// Збережемо посилання на об’єкти різних класів в одному масиві
Object a[] = {new String("qqqq"),  new int[] {1,2,3}, new Object() };

// Обробимо цей масив, використовуючи поліморфізм
for( Object i : a) {
     // Вивести назву класу для кожного об’єкта
     System.out.print( i.getClass().toString() + "   ");

     // три способи перевірити клас об’єкта
     // if( i.getClass() == String.class ) {...}      // 1) i – рядок?
     // if ( String.class.isInstance(i) )  {...}      // 2) те ж

     if (i instanceof String ) {      // 3) i – типу String або спадкоємець String?

          // Щоб працювати  як із рядком потрібне приведення типу
          System.out.println( ((String)i).charAt(0));
```

```
    } else if (i instanceof int[]) {     // Якщо i - масив

        // Приведення типу до типу-масиву та обробка масиву
        // Не можна привести до типу Object[] !!!
        System.out.println(Arrays.toString( (int[]) i ));
    }
}
```

Примітка: щоб перевірити, чи є посилання масивом (будь-якого типу), можна використовувати конструкцію виду:

```
if( i.getClass().isArray() ) { ... }
```

Абстрактні класи та інтерфейси

Абстрактний клас - це клас в якому частина методів не реалізована. Він оголошується з модифікатором **`abstact`**. Цей же модифікатор повинен використовуватися для нереалізованих методів, наприклад:

```
public abstract class D {        // Абстрактний клас
   . . .
    protected int g1(int s) {    // Звичайний метод
       . . .
    }

    public abstract void g2(String str);          // Абстрактний метод
    . . .
}
```

Абстрактні класи використовуються, якщо деякі методи потрібно оголосити як загальні для всіх класів-нащадків, але надати загальну реалізацію неможливо або недоцільно. Нехай, наприклад, маємо клас «Фігура», і від нього успадковані класи «Прямокутник» і «Трикутник». В даному випадку доречно оголосити операцію знаходження площі в базовому класі «Фігура», але реалізовуватися вона буде в кожному похідному класі по-різному.

Всі абстрактні класи мають такі особливості:

- *Не можна створити об'єкт абстрактного класу* (наприклад «Фігура»). Необхідно, використовуючи цей клас як базовий, створити інший клас (наприклад «Трикутник»), в якому визначити всі абстрактні методи. Тоді можна буде створювати об'єкти похідного класу «Трикутник».
- З іншого боку, *не заборонено описувати змінні абстрактного класу*. Наприклад, користуючись поліморфізмом, можемо зберігати посилання типу «Трикутник» або «Прямокутник» в змінній типу «Фігура».

Інтерфейс – це повністю абстрактний клас, який не містить ніяких полів, крім констант (`static final` поля). Інтерфейс оголошується подібно класу, але з ключовим словом **interface.** Клас може реалізувати один або кілька інтерфейсів – для цього інтерфейси вказуються після ключового слова **implements**. Розглянемо приклад:

```
public interface Figure {        // Інтерфейс
   . . .
   double square();              // Абстрактні методи
}

// Клас, що реалізує інтерфейс
public class Triangle implements Figure {
   public double square() {      // Реалізація абстрактного методу
      // ...
   }
}
```

Усередині неабстрактного класу, що реалізує деякий інтерфейс, повинні бути реалізовані всі методи, описані в цьому інтерфейсі. Також можна створити абстрактний клас, який реалізує тільки деякі методи інтерфейсу. Як і з абстрактними класами, не можна створювати об'єкти інтерфейсів, але можна описувати змінні типу інтерфейсів, наприклад:

```
Figure f1 = new Triangle(), s1 = new Rectangle();
```

Отже, інтерфейси схожі з абстрактними класами, але, на відміну від них:

- інтерфейс не може містити ніяких полів, крім констант, а також неабстрактних методів;
- інтерфейси допускають множинне успадкування;
- всі методи інтерфейсу за замовчуванням вважаються `public`, але в похідному класі вказувати `public` обов'язково.

Інтерфейси для порівняння об’єктів

У Java є два стандартних інтерфейси **java.lang.Comparable** і **java.util.Comparator**, які суттєво спрощують пошук і сортування об'єктів в масивах і колекціях. Обидва інтерфейси забезпечують можливість порівнювати об’єкти між собою на «більше-менше». Розглянемо детально один з них – **Comparable**.

В інтерфейсі **Comparable** оголошений всього один метод

```
int compareTo(Object obj)
```

Він повинен порівнювати поточний об'єкт (`this`) з об'єктом obj. На відміну від методу `equals()`, який повертає `true` або `false`, `compareTo()` повертає:

• 0, якщо об'єкти рівні;
• Від’ємне значення, якщо поточний об'єкт менше параметра;
• Додатне значення, якщо поточний об'єкт більше параметра.

Стандартні класи, такі як `Integer`, `Long`, `Double`, `String`, `Date` та ін., вже реалізують інтерфейс `Comparable`.

Як параметр в `compareTo()` може передаватися об'єкт будь-якого класу. Це не дуже зручно, оскільки перед порівнянням потрібно перевіряти тип цього об'єкта.

Є також друга форма цього інтерфейсу, в якій явно вказується клас, з яким буде проводитися порівняння: `Comparable<Class>`.

Розглянемо обидва варіанти на прикладі класу дробів.

Приклад. Клас дробів, що реалізує інтерфейс `Comparable`. Тут порівняння дробів зведено до порівняння двох об'єктів класу `Double`.

```
class Ration
     implements Comparable        // Для порівняння з будь-якими об’єктами
{
     int num = 0, den = 1;        // Змінні реалізації
     // ...

     public int compareTo(Object r) {          // Реалізація Comparable
          // Необхідна перевірка типу і приведення типу перед порівнянням!
          if( !(r instanceof Ration) ) return -1;
          Ration rr = (Ration)r;
          return Double.valueOf((double)num/den).compareTo( (double)rr.num/rr.den );
     }
}
```

Приклад. Клас дробів, що реалізує інтерфейс `Comparable<Ration>`. У цьому випадку метод `compareTo()` виглядає простіше, тому що в ньому не потрібна перевірка типу параметру.

```
class Ration
     implements Comparable<Ration>    // Для порівняння тільки з об’єктами Ration
{
     int num = 0, den = 1;            // Змінні реалізації
     // ...

     // Реалізація Comparable<Ration>
     public int compareTo(Ration r) {
          return  Double.valueOf( (double)num/den ).compareTo(
                         (double)r.num/r.den );
     }
}
```

Тема 4. Класи стандартної бібліотеки

Лекція 6

Бібліотеки і пакети

Бібліотека Java – це набір скомпільованих класів. Зазвичай бібліотека – це .jar-файл (rt.jar, наприклад). Але свою особисту бібліотеку можна зробити і просто в будь-якому каталозі. Крім того, бібліотека може бути zip-файлом.

Щоб використовувати будь-який готовий клас, потрібно підключити бібліотеку, в якій цей клас знаходиться. Компілятор Java знаходить шлях до бібліотек класів в змінній оточення CLASSPATH. У IntelliJ IDEA бібліотеки можна додавати в діалозі Project Structure (Ctrl-Alt-Shift-S) на вкладках Libraries і Global Libraries. Саме там вказується використовувана бібліотека JDK, що містить всі стандартні пакети Java.

Пакет – це група класів всередині бібліотеки. Ім'я пакета визначає загальний простір імен для цих класів. Наприклад, ми можемо створити власний клас `String` у власному пакеті `myclasses`, і тоді він не буде конфліктувати зі стандартним класом `String`.

Стандартна бібліотека Java містить наступні ***основні пакети***:

- `java.lang` – основний пакет, який підключається автоматично і містить найбільш загальні класи: `Object`, `System`, `String`, класи виключень, потоків і т.д;
- `java.io` – пакет підтримки введення/виведення, що містить класи для роботи з файлами і потоками введення-виведення;
- `java.util` – містить безліч корисних класів, що полегшують написання програм. У цьому пакеті містяться класи-колекції, класи по роботі з датою-часом, і ін .;
- `java.awt` – базові класи для створення графічного інтерфейсу;
- `javax.swing` – розширений пакет класів візуальних компонент (з'явився в Java 2.0);
- `java.applet` – класи для створення аплетів.

Щоб використовувати клас з пакета, його потрібно підключити за допомогою оператора **import**:

```
import java.util.ArrayList;
. . .
ArrayList objList = new ArrayList();
```

Можна одним рядком підключити всі класи з пакета, наприклад,

```
import java.util.*;
```

Пакети можуть бути вкладені один в другий. Оскільки в проекті кожен пакет являє собою папку з java-файлами, то вкладені пакети представляють собою вкладені папки. Ім'я вкладеного пакету складається за принципом <пакет>. <вкладений_пакет>. <ще_вкладений_пакет>.

Створення пакета

Для того щоб створити власний пакет необхідно виконати наступні дії:

1. Вибрати ім'я пакета. Згідно загальноприйнятому правилу, в імені пакета повинна бути присутня web-адреса (домен) фірми-розробника (або електронна адреса окремого розробника) в зворотному порядку. Наприклад, якщо адреса розробника my@prov.ua, то для пакета його допоміжних класів він використовує ім'я ua.prov.my.util. Загальною угодою є іменування пакетів тільки маленькими буквами.

2. Створити необхідну структуру каталогів: а) створити каталог бібліотеки, наприклад, c:\javaproj; б) в ньому створити підкаталоги, що відповідають ієрархії пакетів, наприклад c:\javaproj\ua\ prov\my\util.

3. Підключити бібліотеку до проекту (або задати шлях до бібліотеки в змінній classpath). Якщо пакет створюється засобами IDE безпосередньо в тому ж проекті, де він буде використовуватися, то цей крок не потрібен.

4. На початку кожного java-файлу пакета вказати оператор **package**, наприклад:

```
package ua.prov.my.util;
class MyClass {public void use() {} }
```

5**.** Підключити і використовувати клас із пакета, наприклад:

```
import ua.prov.my.util.MyClass;
...
MyClass a = new MyClass();   a.use();
```

Основні класи пакета java.lang

Клас Math

Клас `Math` необхідний для виконання математичних операцій. Також він містить дві константи: `E` (константа Ейлера) і `PI` (число π). Всі методи цього класу статичні. Нижче наведені деякі з них.

– abs – повертає абсолютне значення числа;
– sin, cos, tan, asin, acos, atan – обчислюють тригонометричні функції від числа;
– atan2 (x, y) – повертає арктангенс числа x/y;
– ceil – найближче більше ціле число;
– floor – найближче менше ціле число;
– rint – округлює число;
– round – округлює число до значення в типі int або long;
– sqrt, pow, exp, log – обчислюють стандартні математичні функції;
– min, max – вибирають мінімальне/максимальне з двох чисел;
– random – повертає випадкове число від 0 до 1.

Клас System

Клас `System` представляє саму JVM. Конструктор `System()` оголошений `protected`, тому створювати об'єкти класу `System` не можна. Всі методи і властивості цього класу є статичними:

– стандартні потоки введення, виведення і помилок.

```
System.in, System.out, System.err
```

– поточний час і лічильник часу високої точності

```
// Час в мілісекундах з 1.01.1970
long t = System.currentTimeMillis();
// Кількість наносекунд з моменту запуску ОС.
// Використовується тільки для точного заміру продуктивності
long t = System.nanoTime();
```

– метод для отримання змінних стану середовища із ОС `getenv()`. Наприклад, отримання значення змінної PATH виглядає наступним чином

```
Map<String,String> env =  System.getenv();
String val =  System.getenv( "PATH" );
```

– управління параметрами самої JVM за допомогою методів:

```
String val =  getProperty( "key" );
```

```
setProperty( "key", "value" );
```

Тут `key` може приймати значення:

java.class.path – Java class path;
java.library.path – List of paths to search when loading libraries;
os.name – Operating system name;
os.version – Operating system version;
file.separator – File separator ("/" on UNIX);
path.separator – Path separator (":" on UNIX);
line.separator – Line separator ("\n" on UNIX);
user.name – User's account name;
user.home – User's home directory;
user.dir – User's current working directory;

Клас String

`String` – це дуже корисний клас, який надає методи для роботи з рядками. Перелічимо найбільш важливі з його методів:

1. Клас `String` реалізує всі методи `Object`;
2. `int length()` – повертає довжину рядка (кількість символів в ньому);
3. `boolean isEmpty()` – перевіряє чи порожній рядок;
4. `String replace(a, b)` – повертає рядок, де символ a (літерал або змінна типу `char`) замінений на символ b;
5. `String toLowerCase(); StringtoUpperCase()` – повертає рядок, де всі символи початкової строки перетворені до рядкових (прописних);
6. `int indexOf(ch); int lastIndexOf(ch)` – повертає індекс першого (останнього) входження символу ch в рядок, або –1 якщо символ не знайдений.
7. `int indexOf(ch,n)` – те ж, починаючи з індексу n.
8. `char charAt(n)` – повертає код символу, що знаходиться в рядку під індексом n.

Розглянемо тепер більш детально деякі найбільш часто використовувані методи.

`public char charAt (int index)`

Повертає символ із зазначеним зміщенням в рядку. Відлік йде від 0. Для вилучення одного символу використайте `getChars()`. Приклад:

```
String testString = "Котеня";
char myChar = testString.charAt(2);
tv.setText(Character.toString(myChar)); // виводить третій символ - т
```

`public int compareTo(String string)`

Порівнює два рядки на «більше-менше» в алфавітному порядку символів Unicode. Може використовуватися при сортуванні. Якщо рядки співпадають, то результат дорівнює 0; якщо перший рядок менший за другий, то результат менше нуля, і нарешті, якщо перший рядок більший за другий, то результат більше нуля. Регістр символів враховується: символи верхнього регістру (великі літери) вважаються «менше» символів нижнього регістру (малих). Приклад:

```
String testString = "Котеня";
if (testString.compareTo("котеня") == 0) {           // Умова не виконується
    tvInfo.setText("Рядки рівні");
} else {
    tvInfo.setText("Рядки не рівні. Повернуто "
            + testString.compareTo("котеня"));       // повертає -32
}
```

Тепер відсортуємо масив рядків за допомогою бульбашкового методу .

```
String[] text = { "First", "Second", "Third", "Fourth" };
for (int j = 0; j < text.length; j++) {
     for (int i = j + 1; i < text.length; i++) {
          if (text[i].compareTo(text[j]) < 0) {
               String temp = text[j];
               text[j] = text[i];
               text[i] = temp;
          }
     }
     System.out.print(text[j] + " ");
}
```

В результаті отримаємо:

```
First Fourth Second Third
```

`public void getChars(int start, int end, char[] buffer, int index)`

Цей метод призначений для вилучення під-рядка із рядка. Треба вказати індекс початку під-рядка, що вилучається, (start) та індекс символу, наступного за кінцем під-рядка (end). Масив, який приймає виділені символи, знаходиться в параметрі buffer. Індекс в масиві, починаючи з якого буде записуватися під-рядок, передається в параметрі index. Масив повинен бути достатнього розміру, так щоб в ньому помістилися всі символи зазначеного під-рядка.

```
String unusualCat = "Котеня по імені Мяв";
int start = 5;
int end = 12;
char[] buf = new char[end - start];
unusualCat.getChars(start, end, buf, 0);
System.out.println(new String(buf));
```

`public String replace(CharSequence target, CharSequence replacement)`
`public String replaceAll (String regularExpression, String replacement)`
`public String replaceFirst (String regularExpression, String replacement)`

Ці методи міняють символ чи послідовність символів target на replacement, наприклад:

```
String testString = "кит";
System.out.println(testString.replace("и", "і"));     // результат – "кіт"
String s = "001234-cat";
String s = s.replaceFirst ("^0*", "");                // результат – "1234-cat"
```

Розглянемо наступний приклад роботи з рядками і символами. Нехай необхідно згенерувати випадковий рядок із заданого набору символів.

```
// Набір символів
private static final String mCHAR = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
private static final int STR_LENGTH = 9; // довжина генерованого рядка

Random random = new Random();
StringBuffer randStr = new StringBuffer();
for (int i = 0; i < STR_LENGTH; i++) {
    int number = random.nextInt(mCHAR.length()-1);
    char ch = mCHAR.charAt(number);
    randStr.append(ch);
}
System.out.println(randStr.toString());
```

Класи – колекції

Для зберігання великої кількості однотипних даних можуть використовуватися масиви, але вони не завжди є ідеальним рішенням. По-перше, довжина масиву задається заздалегідь. Якщо кількість елементів заздалегідь невідома, доведеться або виділяти пам'ять «із запасом», або робити складні дії з виділення наново пам’яті для масиву. По-друге, елементи масиву мають жорстко задане розміщення в його комірках, тому, наприклад, видалення елемента з масиву не є простою операцією.

У програмуванні широко використовуються такі структури даних як стек, черга, список, множина і т. д., об'єднані загальною назвою колекція. ***Колекція*** – це група елементів з операціями додавання, вилучення та пошуку елемента. Механізм цих операцій істотно розрізняється залежно від типу колекції. Наприклад, елементи *стека* строго впорядковані, додавання нового елемента може відбуватися тільки в кінець послідовності, а отримати можна тільки той елемент, що знаходиться в кінці (тобто той, що був доданий останнім). *Черга*, навпаки, дозволяє отримати лише перший елемент (елементи додаються в один кінець послідовності, а вилучаються з іншого). Інші колекції (наприклад, *список*) дозволяють вилучити елемент з будь-якого місця послідовності, а *множина* взагалі не впорядковує елементи і дозволяє (крім додавання і видалення) тільки дізнатися, чи міститься в ній даний елемент, але не дізнатися номер його позиції.

Java має бібліотеку стандартних колекцій, які зібрані в пакеті `java.util`, тому немає необхідності програмувати їх самостійно.

При роботі з колекціями головне уникати помилки початківців – треба користуватися найбільш універсальною колекцією – наприклад, писати код з розрахунком на обробку будь-якого списку, замість того щоб розраховувати тільки на випадок стека. Але якщо для логіки роботи програми критично, що дані зберігаються саме в стеці (тобто з'являються і обробляються в зворотній послідовності), слід використовувати саме стек. Тоді не можливо буде порушити логіку обробки даних, звернувшись безпосередньо до середини послідовності, а значить, шанс появи важко вловимих помилок, різко зменшується.

Щоб вибрати колекцію, яка найкраще підходить до кожної задачі, необхідно знати особливості кожної з них. Ці знання є обов'язковими для будь-якого програміста, оскільки без застосування тих чи інших колекцій не обходиться жодна сучасна задача.

Отже, всі класи колекцій визначені в пакеті `java.util`. Кожна колекція – це ***шаблон*** (generic type), тому може зберігати елементи будь-якого ***типу-посилання***. Примітивні типи в колекції зберегти неможливо, замість них потрібно використовувати класи-обгортки (наприклад `Double` замість `double`, `Integer` замість `integer` і т. д.).

Колекції об'єктів розбиті на кілька великих категорій, описаних наступними **інтерфейсами** (рис. 4.1):

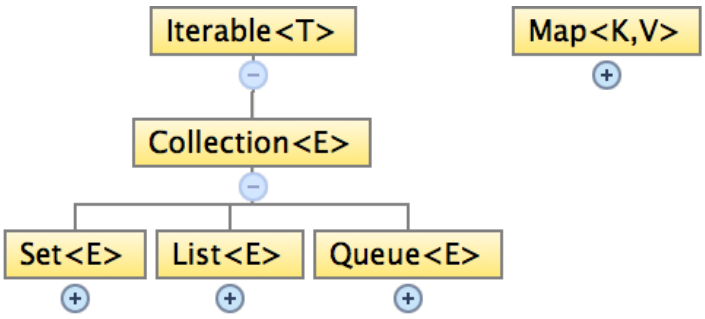

Рис. 4.1 Інтерфейси колекцій JCF

Існує два базових інтерфейси, реалізації яких представляють сукупність всіх класів колекцій: `Collection` і `Map`.

Collection<E> – це колекція, що містить набір об'єктів (елементів). Тут визначено основні методи для маніпуляції елементами, такі як вставка (*add, addAll*), видалення (*remove, removeAll, clear*), пошук (*contains*).

Інтерфейс `Collection<E>` визначає абстрактну колекцію, яка підтримує (але не реалізує) такі основні операції:

- додати елемент в колекцію `add(E e);`
- отримати розмір колекції `size();`
- перевірити, чи є елемент в колекції (рівний в сенсі `e.equals()`) `contains (E e);`
- вибрати елемент за номером `get(int i);`
- замінити елемент за номером `set(int i, E e);`
- видалити з колекції елемент за значенням `remove(Object o);`
- перетворити колекцію на масив `E arr[] = (E[]) coll.toArray();`
- отримати ітератор `iterator();`

Також є можливість перебрати елементи будь-якої колекції за допомогою циклу *for each*.

Цей інтерфейс уточнюється трьома більш конкретними інтерфейсами: List, Set і Queue.

- **List<E>** – описує упорядкований список. Елементи списку пронумеровані, починаючи з нуля, і до будь-якого елементу можна звернутися за його номером у списку (індексом). Інтерфейс `List` є спадкоємцем інтерфейсу `Collection`, тому містить всі його методи і додає до них кілька власних:

`add (int index, Object item)`	вставляє елемент item в позицію index, при цьому список розсовується (всі елементи, починаючи з позиції index, збільшують свій індекс на 1);
`get (int index)`	повертає об'єкт, що знаходиться в позиції index;
`indexOf (Object obj)`	повертає індекс першої появи елемента obj в списку;
`lastIndexOf (Object obj)`	повертає індекс останньої появи елемента obj в списку;
`add (int index, Object item)`	замінює елемент, що знаходиться в позиції index, іншим об'єктом item;
`subList (int from, int to)`	повертає новий список, що є частиною даного (починаючи з позиції from до позиції (to-1) включно).

- **Set<E>** – множина об'єктів (невпорядкований список без повторень).

У множині:

а) повторне додавання об'єкта не змінює множину;

б) порядок елементів при будь-якому способі перебору множини – довільний.

Інтерфейс `Set` успадкований від інтерфейсу `Collection`, але ніяких нових методів не додає. Змінюється тільки поведінка методу `add (Object item)` – він не стане додавати об'єкт item, якщо той вже присутній у множині.

- **Queue<E>** – інтерфейс, який описує чергу. Елементи можуть додаватися в чергу тільки з одного кінця, а виходити з іншого (аналогічно черзі в магазині). Інтерфейс `Queue` так само успадкований від інтерфейсу `Collection`. Специфічні для черги методи:

`poll()`	повертає перший елемент і видаляє його з черги;
`peek()`	повертає перший елемент черги, не видаляючи його;
`offer(Object obj)`	додає в кінець черги новий елемент і повертає true, якщо вставка вдалася.

На рис. 4.2 показані інтерфейси, які стосуються `Collection`, та їхні методи.

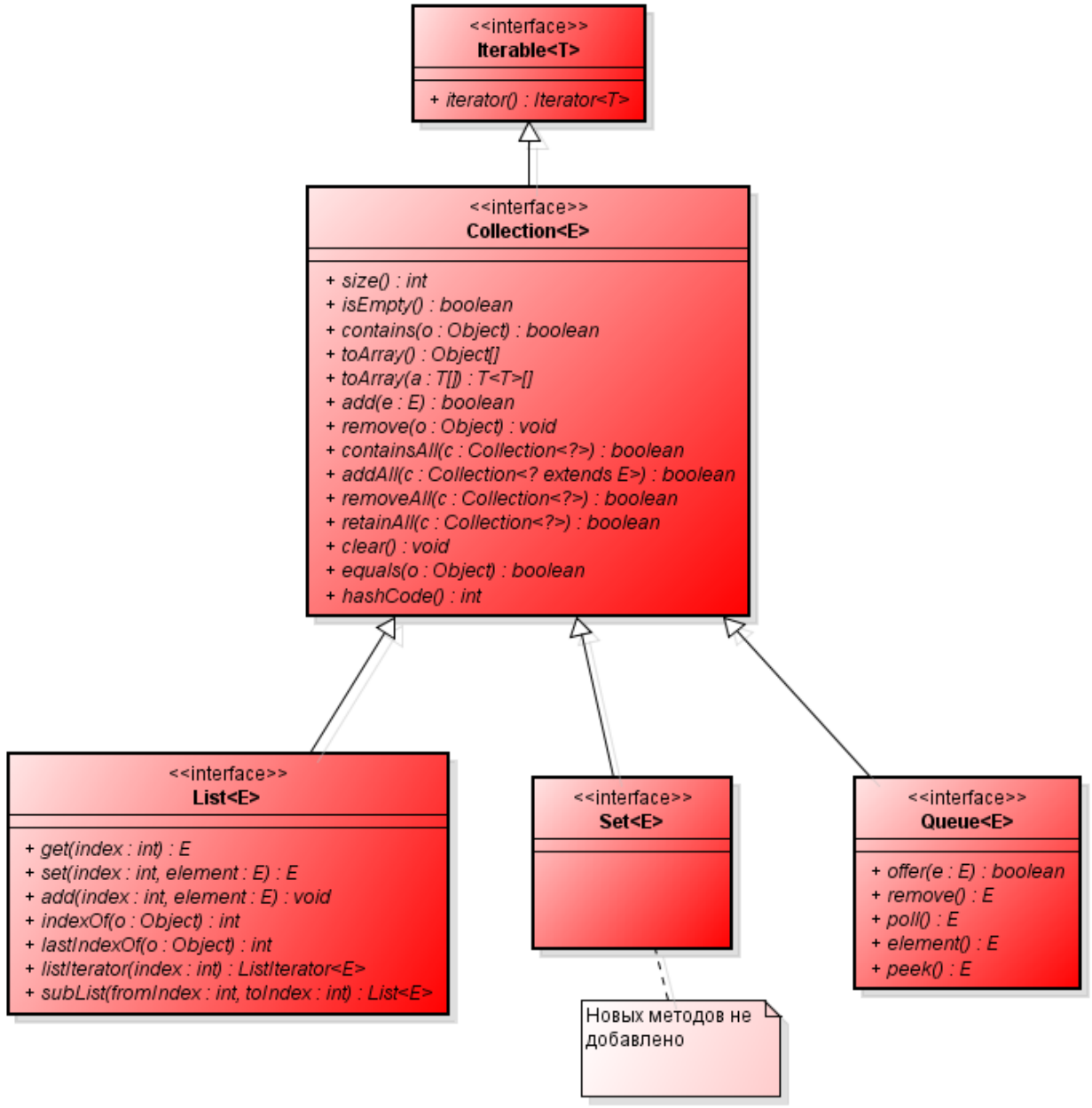

Рис. 4.2. Верхня частина ієрархії `Collection`

Map <K, V> описує колекцію, що складається з пар "*ключ - значення*". Для кожного ключа зберігається тільки одне значення, що відповідає математичному поняттю однозначної функції або відображення. Тут K вказує тип ключів, а V – тип збережених значень. Таку колекцію часто називають ще *словником* (dictionary) або *асоціативним масивом* (associative array). Зверніть увагу, що `Map` не відноситься до інтерфейсу `Collection` і є самостійним інтерфейсом. В інтерфейсі `Map` основною операцією є пошук *значення по ключу* – і ця операція виконується дуже швидко, за постійний час, так само як і додавання/видалення пар ключ-значення. Також із відображення (`Map`) можна отримати множину (`Set`) ключів і список (`List`) значень.

Кожен інтерфейс колекції реалізується декількома класами:

- Інтерфейс `List<E>` реалізується класами `LinkedList<E>` і `ArrayList<E>`, які ми розглянемо далі.
- Інтерфейс `Set<E>` реалізується класами `HashSet<E>` (звичайна множина) і `TreeSet<E>` (впорядкована множина).
- Інтерфейс `Map<K, V>` реалізується класами `HashMap<K, V>`, `LinkedHashMap<K, V>`, `TreeMap<K, V>`.

Класи, що реалізують один інтерфейс, мають однаковий набір методів, і тому використовуються в програмі абсолютно аналогічно. Але, з іншого боку, кожен клас використовує власне внутрішнє представлення даних, через що він може більш ефективно виконувати одні операції, а інші – менш ефективно. Ці особливості кожного класу потрібно враховувати при виборі, який з них використовувати, оскільки в разі невдалого вибору робота програми може сповільнитися в десятки і навіть сотні разів. Далі розглянемо найбільш часто використовувані класи-колекції.

ArrayList<E>

Це найбільш часто використовувана колекція. `ArrayList` реалізує роботу зі списком за допомогою звичайного масиву, довжина якого автоматично збільшується при додаванні нових елементів. Оскільки `ArrayList` використовує масив, то час доступу до елементу за індексом мінімальний (на відміну від `LinkedList`).

З іншого боку, при видаленні довільного елемента зі списку, всі елементи, що знаходяться «правіше», зміщуються на одну комірку масиву вліво. Це досить тривала операція. Важливо пам'ятати, що при видаленні і додаванні елементів реальний розмір масиву (його ємність, capacity) не змінюється. Якщо при додаванні елемента виявляється, що масив повністю заповнений, буде створено новий масив розміром (n * 3) / 2 + 1, де n – попередній розмір масиву, в нього будуть поміщені всі елементи зі старого масиву і додано один новий елемент.

Таким чином `ArrayList` має постійний час доступу за номером і лінійний час для всіх інших операцій. Розглянемо роботу зі списком `ArrayList` на прикладі.

Приклад. Створимо список дробів (тобто об'єктів створеного раніше класу `Ration`) і покажемо різні способи перебору списку а також сортування та випадкове перемішування елементів.

```
ArrayList<Ration> lst = new ArrayList<Ration>(10);
for(int i=0; i<5; i++ )
     lst.add( Ration.random(10) );
Ration r1  = lst.get(0);              // Отримати перший елемент
lst.add(2, new Ration (0,1) );        // Вставити в 3-ю позицію
lst.add(2, new Ration (0,1) );        // Вставити в 3-ю позицію
lst.remove(2);                        // Видалити 3-й елемент
lst.set( lst.size()-1, new Ration (1,1) );  // Заміна елемента
out( lst.toString() );


// Пройти по колекції, використовуючи цикл for
for( Ration r : lst) {
     r.add(new Ration(1,1));                 // Елемент колекції змінюється
     r = Ration.add(r, new Ration(1,1));     // Не впливає на елементи
                                             // колекції
}
out( lst.toString() );

// Перебрати колекцію, використовуючи ітератор
Iterator<Ration> it = lst.iterator();
while(it.hasNext()) {
     it.next().add(new Ration(1,1));
}
out( lst.toString() );

Collections.shuffle(lst);    // Перемішати елементи
out( lst.toString() );

Collections.sort(lst);       // Відсортувати у відповідності до compareTo()
out( lst.toString() );
```

LinkedList<E>

Реалізує ті ж методи, що і `ArrayList`, але для внутрішнього представлення даних використовує двонаправлений список. Доступ до довільного елементу за номером в такому списку вимагає переглядати список від самого початку, і тому здійснюється за лінійний час. При цьому доступ до першого і останнього елементів списку завжди здійснюється за константний час, оскільки посилання на перший та останній елементи постійно зберігаються в об'єкті `LinkedList`. В цілому ж, як за швидкістю роботи, так і по споживаній пам'яті `LinkedList` в більшості випадків програє `ArrayList`, але цей клас незамінний у випадках, коли часто доводиться вставляти і видаляти елементи в середині списку, або ж загальна кількість елементів змінюється в широких межах.

HashMap<V,K>

Як вже говорилося, даний клас реалізує *асоціативний масив*, тобто набір пар виду *ключ-значення*. При цьому ключі не повторюються. У будь-який момент можна отримати елемент-значення, асоційоване (тобто що знаходиться в парі) із заданим ключем. У вигляді асоціативного масиву зручно зберігати, наприклад, рядок, отриманий з бази даних. В цьому випадку ключем буде назва стовпця, а значенням – значення цього стовпця в даному рядку. Можна сприймати асоціативний масив як різновид звичайного масиву, в якому індексами, за якими здійснюється доступ до елементів, є не цілі числа, а довільні об'єкти – найчастіше рядки.

Інтерфейс `Map` містить методи для роботи з асоціативним масивом:

`size()`	повертає кількість елементів (пар) в масиві;
`containsKey(Object key)`	перевіряє, чи існує в масиві елемент з ключем key;
`containsValue(Object value)`	перевіряє, чи існує в масиві елемент зі значенням value;
`get(Object key)`	повертає значення, що відповідає ключу key;
`put(Object key, Object value)`	додає в масив елемент з ключем key і значенням value. Якщо елемент з таким ключем вже існує в масиві, то його значення просто змінюється;
`values()`	повертає значення всіх елементів масиву у вигляді колекції (тобто, результат, що повертається, має тип `Collection`);
`remove(Object key)`	видаляє елемент з ключем key, повертаючи значення цього елемента (якщо він є) і null, якщо такого елемента немає;
`clear()`	очищає масив;
`isEmpty()`	перевіряє, чи не порожній масив.

Покажемо основні прийоми роботи на прикладі колекції, в якій ключами є об'єкти нашого класу дробів `Ration`, а значеннями – рядки. Зверніть увагу, для того щоб клас можна було використовувати в якості ключа `Map`, в ньому повинні бути перевизначені методи `hashCode()` і `equals()`, як обговорювалося в попередній лекції.

Приклад. Прийоми роботи з `Map`

```
// Створюємо Map
HashMap<Ration, String> map = new HashMap<Ration, String>(20);
Ration r;

// Додавання пар
for(int i=0; i<5; i++ )
     map.put( r = new Ration(i,i+1), r.toString() );

// Заміна і видалення значень
r = new Ration(3,4);
map.put(r, "aaa");
map.remove( new Ration (2,3));
out( map.toString() );
```

```
// Map не являється Collection, тому для нього не можна використовувати
// ітератори і цикл виду for( Ration r : map)

// Знайти значення за ключем
if ( map.containsKey(r) )
     out( map.get( r ) );

// Прохід за допомогою ключів
Set<Ration> set = map.keySet();
for(Ration rr : set) {
     out("key = " + rr + " value = " + map.get(rr) );
}

// Пошук за значенням
out( "" + map.containsValue( "3/4" ) );

// Прохід по значенням
// (отримувана колекція містить копії посилань на значення)
Collection<String> col = map.values();
String ss = "";
for(String s : col) ss += " " + s;
out( ss );

// Зміна map через колекцію col
col.remove( "3/4" );
out( map.toString() );
```

Застарілі колекції

Наступні класи колекцій є застарілими, і їх використання не рекомендується, але не забороняється.

1. `Enumeration` – аналог інтерфейсу `Iterator`.

2. `Vector` – аналог класу `ArrayList`; підтримує впорядкований список елементів, що зберігаються у "внутрішньому" масиві.

3. `Stack` – клас, похідний від `Vector`, в який додані методи включення (push) і виключення (pop) елементів, так що список може трактуватися в термінах, прийнятих для опису структури даних стека.

4. `Dictionary` – аналог інтерфейсу `Map`, хоча є абстрактним класом, а не інтерфейсом.

5. `Hashtable` – аналог реалізації `HashMap`.

Всі методи `Hashtable`, `Stack`, `Vector` є ***синхронізованими***, що робить їх менш ефективними в додатках з одним потоком виконання.

Класс java.util.Collections

Містить декілька зручних методів для роботи з колекціями:

`emptyList(); emptyMap(); emptySet();` – створення порожніх колекцій, тільки для читання;

`min(collection); max(collection);` – пошук мінімального або максимального елемента в колекції. Елементи порівнюються за допомогою методу `compareTo()`, тобто вони повинні реалізовувати інтерфейс `Comparable`;

`sort(lst);` – сортування елементів колекції в порядку, що встановлюється методом порівняння `compareTo()`;

`shuffle (lst);` – випадкове перемішування елементів колекції;

`binarySearch(lst,key);` – бінарний пошук елемента в колекції. Працює набагато швидше звичайного пошуку, але застосовується тільки в тому разі, якщо колекція попередньо відсортована;

`indexOfSublist(lst,lstSub);` – знайти заданий підсписок lstSub в списку lst;

`unmodifiableList (list); unmodifiableSet (list); unmodifiableMap (map);`
– повертають *незмінне представлення* колекції.

**** Синхронізовані колекції***

Отримати синхронізовані об'єкти колекцій можна за допомогою статичних методів `synchronizedMap()` і `synchronizedList()` класу `Collections`.

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
```

Синхронізовані обгортки колекцій `synchronizedMap()` і `synchronizedList()` іноді називають *умовно* потоко-безпечними: всі методи цих класів окремо потокобезпечні, тобто не можуть бути перервані іншим потоком. Однак необхідно пам'ятати, що *послідовність викликів цих методів* цілком може бути перервана іншим потоком, що може стати причиною конкуренції за дані або порушення цілісності даних. Тому таку послідовність викликів слід завжди розташовувати всередині синхронізованої секції (`synchronized`). Синхронізація потоків і критичні секції розглядаються далі в цьому курсі.

Таким чином, умовна безпека потоку, яку забезпечують `synchronizedMap()` і `synchronizedList()` представляє приховану загрозу: розробники вважають, що, раз ці колекції синхронізовані, значить, вони повністю потоко-безпечні, і нехтують належною синхронізацією складних операцій. В результаті, хоча такі програми і працюють при легкому навантаженні, але при серйозному навантаженні вони можуть почати викидати виключення `NullPointerException` або `ConcurrentModificationException`.

Виключення

Виключеннями або винятковими ситуаціями (станами) називаються помилки, що виникли в програмі під час її роботи (рис. 4.3).

Всі виключення в Java є об'єктами. Тому вони можуть породжуватися не тільки автоматично при виникненні виняткової ситуації, але і створюватися самим розробником.

Рис. 4.3. Типи помилок в програмі

Всі виключення мають спільного предка – клас **Throwable** (рис. 4.4). Його нащадками є підкласи **Exception** і **Error.**

Виключення (**Exception**) є результатом проблем в програмі, які в принципі можна вирішити і передбачити. Наприклад, якщо має місце цілочислове ділення на нуль, то цьому завжди можна запобігти, перевіривши операнди перед діленням.

Помилки (**Error**) представляють собою більш серйозні проблеми, які, відповідно до специфікації Java, не слід намагатися обробляти у власній програмі, оскільки вони пов'язані з проблемами рівня JVM. Наприклад, виключення такого роду виникають, якщо закінчилася пам'ять, доступна віртуальній машині. Навіть якщо програма містить обробку такої ситуації, вона все одно не зможе забезпечити додаткову пам'ять для JVM.

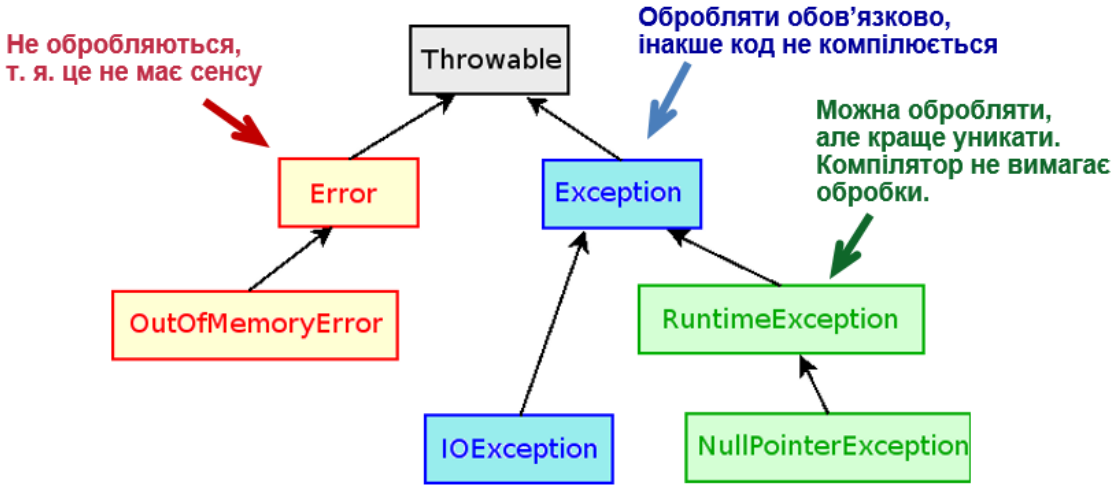

Рис. 4.4. Загальна ієрархія класів виключень

В Java всі виключення поділяються на два типи:

1) ***неконтрольовані виключення (unchecked),*** до яких відносяться помилки (клас **Error**) і виключення часу виконання (клас **RuntimeExceptions**, один з нащадків класу `Exception`). Неконтрольовані виключення не вимагають обов'язкової обробки, проте, при бажанні, можна обробляти виключення класу `RuntimeException`.

2) ***контрольовані виключення (checked)*** – це помилки, які можна і потрібно обробляти в програмі. Це все нащадки класу `Exception`, крім `RuntimeException` і його нащадків (рис. 4.5).

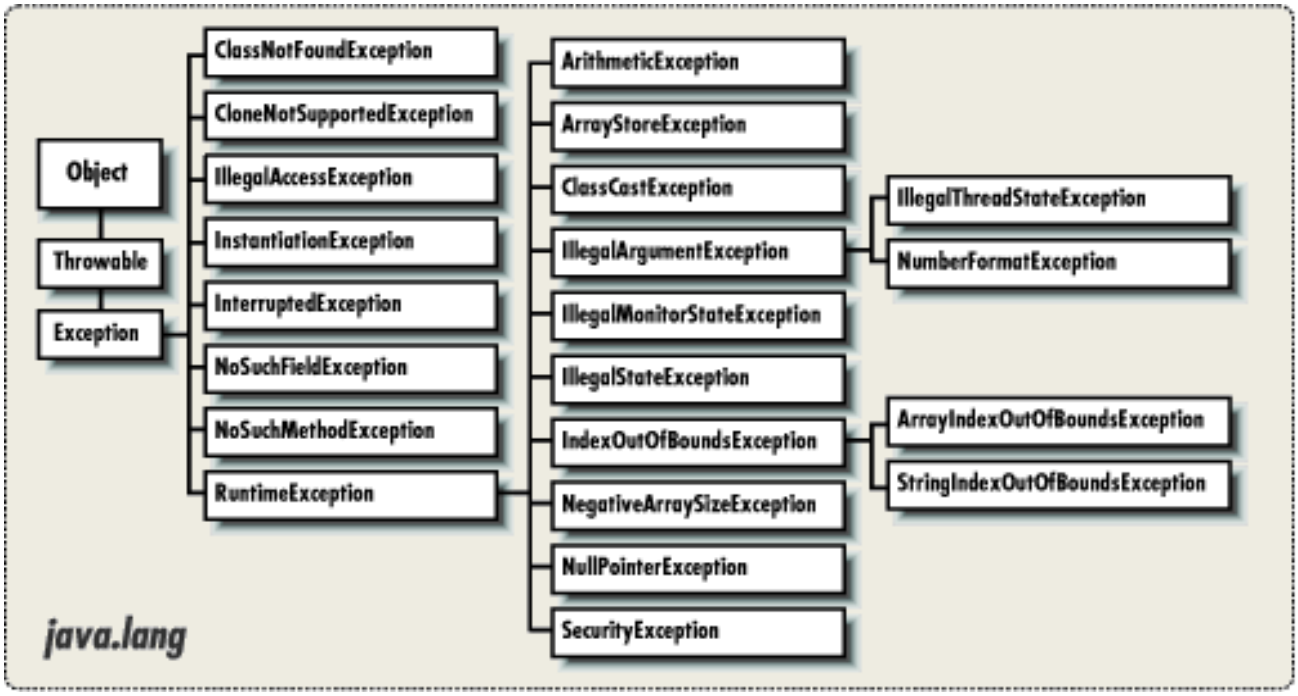

Рис. 4.5. Класи виключень

Наприкладр, код

```
int a = 4;
System.out.println(a/0);
```

генерує виключення класу `ArithmeticException`:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:4)
```

З повідомлення видно клас виключення, що відбулося – `ArithmeticException` і рядок коду, де воно відбулося.

До механізму обробки виключень в Java мають відношення 5 ключових слів: – **try, catch, throw, throws** і **finally**. Схема роботи цього механізму наступна. Ви намагаєтеся (`try`) виконати блок коду, і якщо при цьому виникає помилка, то система генерує (`throw`) виключення. Залежно від типу виключення ви можете його перехопити (`catch`) або передати далі за кодом. Також можна окремо визначити дії, які будуть виконуватися гарантовано (`finally`), незалежно від того, відбулася помилка чи ні.

Загальна форма блока з обробкою виключень має наступний вигляд:

```
try {
     // блок кода
} catch (ТипВиключення e1) {
     // обробник виключень типу ТипВиключення1
} catch (ТипВиключення2 e2) {
     // обробник виключень типу ТипВиключення2
     throw(e2);   // можливе повторне збудження виключення
} finally {
     // код, який виконується в будь-якому випадку
}
```

Приклад обробки різних видів виключень:

```
int[] m = {-1,0,1};
int a = 1;
Scanner sc = new Scanner(System.in);
try {
    a = sc.nextInt();
    m[a-1] = 4/a;
    System.out.println(m[a]);
} catch (ArithmeticException e) {
    System.out.println("Недопустима арифметична операція");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Недопустимий індекс масиву");
} catch (Exception e) {
    e.printStackTrace();
}
```

Якщо користувач введе 0, то виникне виключення класу `ArithmeticException` (ділення на нуль), і воно буде оброблено першим блоком catch.

Якщо користувач введе 3, то виникне виключення класу `ArrayIndexOutOfBoundsException` (вихід за межі масиву), і воно буде оброблено другим блоком `catch`.

Якщо користувач введе нецілe число, наприклад, 3.14, то виникне виключення класу `InputMismatchException` (невідповідність типу введеного значення). Спеціального блоку `catch` для такого класу виключень в нашій програмі немає, але воно буде оброблено в останньому блоці `catch(Exception e)`, оскільки клас `Exception` є батьківським для всіх контрольованих виключень, і, відповідно об'єкт класу `InputMismatchException` також є об'єктом класу `Exception`.

Оскільки весь механізм виключень побудований на ієрархії класів, *то спочатку треба намагатися обробити більш спеціалізовані виключення і лише потім більш загальні.*

Зупинимося трохи докладніше на блоці **finally**. Він може знаходиться після блоків `try` і `catch`. Вміщені в нього команди будуть виконуватися в будь-якому випадку, незалежно від того, чи відбулося виключення чи ні, і чи було воно оброблене. Наприклад, якщо виключення не було оброблене, то решта програми не виконуватиметься, і при цьому можуть залишитися незвільнені ресурси (незакриті потоки, розпочаті і не закінчені транзакції доступу до БД і т. п.). При цьому в блоці `finally` можна гарантовано звільнити всі ресурси.

Ключове слово **throws** вказується при оголошенні методу, щоб повідомити клієнту, які виключення цей метод може генерувати. Якщо `throws` зазначено, то при використанні методу цей тип виключення обов'язково потрібно обробляти, тобто виклик методу повинен обов'язково знаходитись всередині секції `try ... catch`. Наприклад, в класі `java.io.File` є метод створення нового файлу, який оголошений таким чином:

```
public boolean createNewFile() throws IOException
```

Він генерує виключення класу IOException, якщо файл не вдалося створити. IOException відноситься до контрольованих виключень, і тому його:

– потрібно обов'язково вказати при описі методу: `throws IOException`;
– обов'язково потрібно обробляти, наприклад:

```
File f = null;
Try {
   f = new File("test.txt");     // Створюємо об’єкт-файл

   f.createNewFile();            // Намагаємось створити фізичний файл
                                 // з іменем test.txt
} catch(IOException e){          // Обробляємо можливі помилки
   e.printStackTrace();
}
```

Тема 5. Робота з файловою системою

Лекція 7

Потоки введення – виведення

Програми, написані в попередніх розділах, обмінювалися інформацією з користувачем тільки через консоль. Однак у багатьох випадках потрібно виводити результати роботи програми в файл, базу даних або передавати мережею. Вихідні дані теж часто доводиться завантажувати з файлу, бази даних або з мережі. При цьому програмісту зовсім не обов'язково піклуватися про те, з яким саме фізичним джерелом даних він працює. Він може користуватися набором абстракцій (тобто спеціальних класів і інтерфейсів Java), які приховують відмінності між роботою з консоллю, файлом, мережевим з'єднанням або іншим джерелом даних.

Для того щоб програма не залежала від особливостей конкретних пристроїв введення/виведення, в Java використовується поняття ***потоку*** (stream). Вважається, що в програму йде *вхідний потік* (input stream) символів Unicode або просто байтів, що сприймається в програмі методами `read()`. З програми методами `write()` або `print()`, `println()` виводиться вихідний потік (output stream) символів або байтів. При цьому неважливо, куди спрямований потік: на консоль, в файл або в мережу, – методи `read()`, `write()` і `print()` в усіх випадках працюють однаково.

Можна уявити собі потік як трубу, якою в одному напрямку послідовно "течуть" символи або байти, один за іншим. Методи `read()`, `write()`, `print()`, `println()` взаємодіють з одним кінцем труби, а інший кінець з'єднується з джерелом або приймачем даних конструкторами класів, в яких реалізовані ці методи.

Три стандартні потоки доступні через статичні змінні класу `System`: `in`, `out` і `err`. Їх можна використовувати без всяких додаткових визначень, що і робилось раніше. Вони називаються відповідно потоками *стандартного вводу* (stdin), *стандартного виведення* (stdout) і *стандартного виведення повідомлень про помилки* (stderr). Ці стандартні потоки можуть бути з'єднані з різними конкретними пристроями введення і виведення.

Потік `out` – це екземпляр класу `PrintStream`, що організує вихідний потік байтів. Він виводять інформацію на консоль методами `print()`, `println()` і `write()`, які мають близько двадцяти перевантажених версій для різних типів параметрів.

Потік `in` – це екземпляр класу `InputStream`. Він призначений для клавіатурного введення з консолі методами `read()`. Клас `InputStream` абстрактний, тому насправді зажди використовується якийсь із його підкласів.

Поняття потоку виявилося настільки зручним і настільки полегшило програмування введення-виведення, що його застосування виходить далеко за межі класичного введення-виведення. Так, в Java можна створити потік, що направляє символи або байти не на зовнішній пристрій, а до масиву, тобто до певної області оперативної пам'яті. Більш того, можна створити потік, пов'язаний зі об’єктом-рядком типу `String`, що знаходиться, знову-таки, в оперативній пам'яті. Крім того, можна створити *канал* (pipe) обміну інформацією між паралельними потоками (threads) програми, який передає дані також за принципом потоку (stream).

Ще один вид потоку дозволяє читати або записувати об'єкти Java, перетворюючи їх на послідовність байтів. За допомогою такого потоку можна будь-який об'єкт зберегти в файл або передати мережею. Подібне перетворення об’єкта на потік байтів називається ***серіалізацією*** (serialization). Потім серіалізований об'єкт можна відновити з потоку – ця операція називається ***десеріалізацією*** (deserialization).

Всі класи, що відносяться до потоків, зібрані в класи пакету `java.io`.

У Java є цілих чотири ієрархії класів для роботи з потоками. На чолі ієрархії чотири класи, що безпосередньо успадковані від `Object`:

• ***Reader*** – абстрактний клас, в якому зібрані найзагальніші методи текстового введення;

• ***Writer*** – абстрактний клас, в якому зібрані найзагальніші методи текстового виведення;
• ***InputStream*** – абстрактний клас із загальними методами байтового введення
• **OutputStream** – абстрактний клас із загальними методами байтового виведення.

Класи вхідних потоків `Reader` і `InputStream` мають по чотири базових методи введення:

- `read()` – повертає один символ або байт, взятий з вхідного потоку, у вигляді цілого значення типу `int`; якщо потік вже закінчився, повертає –1;
- `read(char[] buf)`, `read(byte[] buf)` – заповнює заздалегідь визначений масив `buf` відповідно символами або байтами з вхідного потоку; метод повертає фактичне число взятих з потоку символів (байтів), або –1, якщо потік вже закінчився;
- `read(char[] buf, int offset, int len)`, `read(byte[] buf, int offset, int len)` – заповнює частину символьного або байтового масиву `buf` довжиною `len`, починаючи з індексу `offset`; метод повертає фактичне число взятих з потоку елементів, або –1, якщо потік вже закінчився.
- `skip(long n)` – пропускає n символів або байтів потоку, починаючи з поточної позиції. Ці байти або символи відкидаються, і наступні виклики методів `read()` їх вже не читають. Метод повертає реальне число пропущених елементів, яке може відрізнятися від n, наприклад, якщо потік закінчився.

Ці методи генерують виключення `IOException` в разі помилки введення/виведення.

Класи вихідних потоків `Writer` і `OutputStream` мають по три схожих методи виведення:

- `write(char[] buf)`, `write(byte[] buf)` – виводять масив у вихідний потік; `OutputStream` працює з масивом байт `byte[]`, а `Writer` – з масивом символів `char[]`;
- `write(char[] buf, int offset, int len)` та `write(byte[] buf, int offset, int len)` – виводять len елементів масиву buf, починаючи з елемента з індексом offset;
- `write(int elem)` – цей метод в класі `Writer` виводить 16 молодших бітів значення `elem`, а в класі `OutputStream` – 8 молодших бітів.

У класі `Writer` є ще два методи для виведення рядків:

- `write(String s)` – виводить рядок s у вихідний потік;
- `write(String s, int offset, int len)` – виводить `len` символів рядка s, починаючи з символу з номером `offset`.

Існує багато підкласів `Writer` і `OutputStream`, які здійснюють *буферизоване* виведення. При цьому елементи спочатку накопичуються в буфері, в оперативній пам'яті, і виводяться у вихідний потік тільки після того, як буфер заповниться. Це зручно для вирівнювання швидкостей виведення з програми і виведення потоку, але часто треба вивести інформацію в потік ще до заповнення буфера. Для цього передбачений метод `flush()`. Даний метод відразу ж виводить весь вміст буфера в потік.

Нарешті, після закінчення роботи з потоком, його необхідно закрити методом `close()`.

Класи, що входять до ієрархії потоків введення-виведення, показані на рис. 5.1 і рис. 5.2.

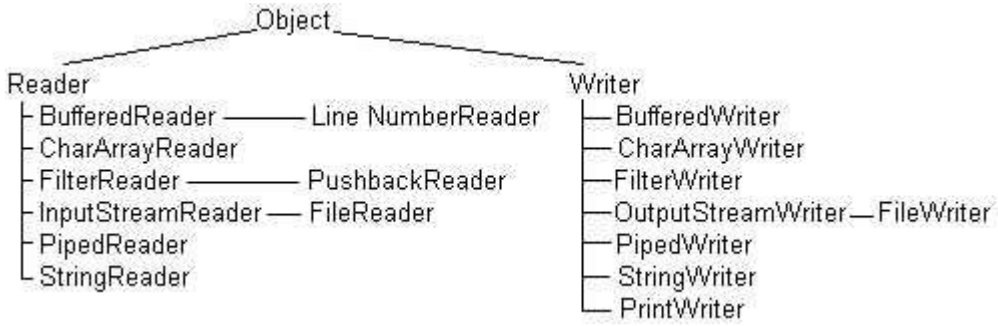

Рис 5.1. Класи символьних потоків

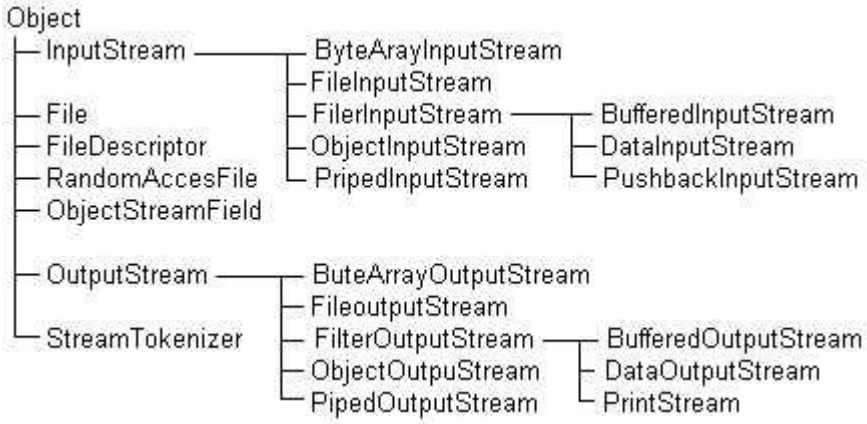

Рис. 5.2. Класи байтових потоків

Всі класи пакету java.io можна розділити на дві групи: класи, що створюють потік (data source/sink), і класи, керуючі потоком (data processing).

Перша група призначена для *створення потоків*, що пов'язані в різноманітними джерелами інформації. Їх можна розділити на п'ять груп (зліва вказані класи символьних потоків, праворуч – класи байтових потоків):

- потоки, пов'язані з файлами на диску:
  `FileReader` `FileInputStream`
  `FileWriterFile` `Outputstream`
  `RandomAccessFile`

- потоки, пов'язані з масивами в оперативній пам'яті програми:
  `CharArrayReader` `ByteArrayInputStream`
  `CharArrayWriter` `ByteArrayOutputStream`

- канали обміну інформацією між підпроцесами:
  `PipedReader` `PipedInputStream`
  `PipedWriter` `PipedOutputStream`

- символьні потоки, пов'язані з об'єктом-рядком:
  `StringReader`
  `StringWriter`

Класи другої групи не пов'язані напряму ні з яким джерелом інформації, а *виконують певні перетворення чи додаткові операції над існуючим потоком*. Всі ці класи отримують в своїх конструкторах посилання на вже наявний потік, і створюють на його основі новий, перетворений потік, з іншими властивостями. Перелічимо найбільш важливі з цих класів.

- Маємо чотири класи, які виконують буферизацію даних в потоках (як байтових, так і символьних):
  `BufferedReader` `BufferedInputStream`

`BufferedWriter` `BufferedOutputStream`

- Також маємо два класи для форматованого текстового виведення, аналогічного виведенню за допомогою об'єкта `System.out` (але не в консоль, а до будь-якого потоку):
  `PrintWriter` `PrintStream`

- Наступні два класи надбудовуються над будь-яким байтовим потоком і дозволяють зчитувати і записувати значення примітивних типів:
  `DataInputStream` `DataOutputStream`

- Нарешті, маємо два класа для передачі об'єктів Java через байтові потоки:
  `ObjectInputStream`
  `ObjectOutputStream`

Цей огляд класів введення-виведення прояснює співвідношення між ними, але не пояснює, як їх використовувати. Перейдемо до розгляду практичних задач введення-виведення.

Файлове введення-виведення

У всіх сучасних операційних системах файли в загальному випадку розглядаються як послідовність байтів. Тому то для файлового введення-виведення використовуються байтові потоки `FileInputStream` і `FileOutputStream`. Це особливо зручно для бінарних файлів, що зберігають дані, архіви, зображення, звук і т. п.

У текстових файлах послідовність байтів кодує певні символи, і зазвичай в Java використовується кодування символів Unicode. Для роботи з такими файлами призначені спеціалізовані класи `FineReader` і `FileWriter`. Дані класи розширюють класи `InputStreamReader` і `OutputStreamWriter`, відповідно, організовуючи перетворення потоку: з боку програми потік символьний, а з боку файлу – байтовий.

Незважаючи на відмінність між байтовими і текстовими потоками, використання класів файлового введення-виведення дуже схоже. У конструкторах всіх чотирьох файлових потоків задається ім'я файлу у вигляді рядка або посилання на об'єкт класу `File`. Конструктори не тільки створюють об'єкт, але також відразу відшукують файл і відкривають його. Наприклад:

```
FileInputStream fis = new FileInputStream( new File("MyFile.dat"));
FileReader fr = new FileReader("D:\\jdkl.3\\src\\PrWr.Java");
```

Кожен з цих конструкторів викидає виключення, якщо файл не знайдений або не вдається отримати доступ до нього – найчастіше `FileNotFoundException` або більш загального вигляду `IOException`.

При відкритті вихідного потоку `FileWriter` або `FileOutputStream` для непорожнього файлу весь вміст цього файлу стирається. Для дописування в кінець файлу в обох класах передбачений конструктор з двома параметрами. Якщо другий параметр дорівнює true, то відбувається запис в кінець файлу, якщо false, то файл заповнюється новою інформацією з початку. Наприклад:

```
FileWriter fw = new FileWriter("notes.txt", true);              // Дозапис
FiieOutputstream fos = new FileOutputStream("D:\\newfile.txt"); // Перезапис
```

Відразу після виконання конструктора можна читати файл:

```
fis.read(); fr.read();
```

чи записувати в нього:

```
fos.write((char)c);
fw.write((char)c);
```

Після закінчення роботи з файлом потік слід закрити методом `close()`.

Примітка. Перетворення потоків у класах `FileReader` і `FileWriter` виконується згідно кодової таблиці, яка визначається встановленою *локаллю* (locale). Тому для правильного введення кирилиці треба застосовувати `FileReader`, а не `FileInputStream`. Якщо файл містить текст у кодуванні, відмінному від локального кодування, то необхідно застосовувати додаткове перетворення потоку – перекодування. Таке перетворення може забезпечити клас `InputStreamReader`, наприклад:

```
InputStreamReader isr = new InputStreamReader(fis, "KOI8_R")); // fis створений вище
```

Приклад. За допомогою потоків введення-виведення `FileInputStream` і `FileOutputStream` скопіюємо файл. Для цього відкриємо існуючий файл для читання, а новий файл – для запису. Потім будемо переписувати дані з одного файлу в інший порціями по 1024 байти.

```
byte b[] = new byte[1024];   // Буфер для копійованих даних
int cnt;                     // Кількість прочитаних до буфера даних
try {

     // Відкриваємо існуючий файл для читання, а новый файл – для запису
     InputStream in = new FileInputStream("/path/name.txt");
     OutputStream out = new FileOutputStream("1.txt",false);

     // Переписуємо дані із одного файла до другого порціями по 1024 байти
     while( (cnt = in.read(b)) > 0 ) {      // Поки ще є дані в in
          // в cnt повертається колькість реально прочитаних байт
          // - це число може бути меншим, чим 1024
          out.write( b, 0, cnt );           // записуємо їх в out
          System.out.write(b, 0, cnt );     // і виводимо на консоль
     }
     out.close();  in.close();              // Закриваємо файли
} catch( Exception e ) { e.printStackTrace(); }
```

Примітка. Роботу даної програми можна істотно прискорити, якщо використовувати класи буферизованих потоків `BufferedInputStream`, `BufferedOutputStream`, як буде показано нижче.

Буферизоване введення-виведення

Операції введення-виведення в порівнянні з операціями в оперативній пам'яті виконуються дуже повільно. Для компенсації цього недоліку зазвичай використовується наступний прийом: в оперативній пам'яті виділяється деяка проміжна область – ***буфер***, у якій поступово накопичується інформація. Коли буфер заповнено, його вміст швидко переноситься до файлу, після чого буфер очищається і знову готовий до заповнення інформацією.

Наочною демонстрацією ідеї буфера може служити звичайний поштовий ящик, в якому накопичуються листи. Ми кидаємо в нього лист і йдемо по своїх справах, не чекаючи приїзду поштової машини. Поштова машина періодично звільняє поштову скриньку, забираючи одразу велику кількість листів. Уявіть собі місто, в якому немає поштових скриньок, і натовп людей з листами в руках чекає приїзду поштової машини. Саме така ситуація складається в багатьох програмах, якщо не використовувати механізм буферизації.

Класи файлового введення-виведення не займаються буферизацією. Для цієї мети є чотири класи, що перетворюють потік: **BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter**.

Вони надають ті ж базові методи для введення-виведення, але за рахунок буферизації значно збільшується швидкість роботи.

Об'єкти цих класів не є самостійними, а завжди приєднуються до готового (вже відкритого) потоку введення-виведення, наприклад:

```
Reader br = new BufferedReader( new FileReader("/path/name.txt") );
```

```
InputStream bis = new BufferedInputStream( System.in );
```

Розглянемо приклад буферизованного файлового введення-виведення.

Приклад. Наступна програма читає текстовий файл, створений у кодуванні CP866, і записує його вміст до іншого файлу в кодуванні KOI8_R. При читанні і записі застосовується буферизація. Ім'я вихідного файлу задається в командному рядку параметром args[0], ім'я копії – параметром args[l].

```
import java.io.*;
class DOStoUNIX{
     public static void main(String[] args) throws IOException{
          if (args.length != 2){
               System.err.println("Треба вказати імена файлів");
               System.exit(0);
          }

          // Створюємо буферизований потік введення
          // на основі звичайного потоку введення, пов’язаного з файлом
          BufferedReader br = new BufferedReader(
               new InputStreamReader(
                    new FileInputStream(args[0]), "Cp866"));

          // Створюємо буферизований потік виведення
          // на основі звичайного потоку виведення, пов’язаного з файлом
          BufferedWriter bw = new BufferedWriter(
               new OutputStreamWriter(
                    new FileOutputStreamtargs[1]), "KOI8_R"));

          int c = 0;                              // Переписуємо дані,
          while ((c = br.read()) != -1) {         // вже не піклуючись про буферизацію
               bw.write((char)c);
          }
          br.close(); bw.close();                 // Закриваємо файли
     }
}
```

Робота з бінарними файлами. Потік примітивів

Два класи **DataInputStream** і **DataOutputStream** – використовуються сумісно для читання-запису значень примітивних типів. Вони містять відповідно методи:

```
writeInt(i);  writeDouble(d); writeBoolean(b); writeChar(c);
readInt();  readDouble(); readBoolean(); readChar();
```

Також в `DataOutputStream` можна записувати рядки:

```
writeBytes(String str);     // Запис молодших байт-символів
writeChars(String str);     // Запис символів по 2 байти
```

Користуючись цими методами необхідно пам'ятати, що перший з них записує кожен символ рядка в один байт, відкидаючи старший байт кодування Unicode (якщо він є), а другий метод записує символи кодування Unicode повністю, так само, як це робить метод `writeChar()`.

У класі `DataInputStream` немає симетричного методу для читання рядків, тому послідовність символів з бінарного файлу можна прочитати тільки як масив `byte[]` методом `read()` базового класу `InputStream`.

Ці класи взагалі не підходять для роботи з рядками, оскільки рядок (String) не є примітивом. Для рядків і інших об'єктних типів передбачений інший механізм введення-виведення – серіалізація, – про який йтиметься далі.

Приклад запису та читання примітивів із файлу:

```
// Записуємо дані різних примітивних типів в файл
DataOutputStream dos = new DataOutputStream( new FileOutputStream("1.dat") );
dos.writeInt(100);
dos.writeDouble(1.2);
dos.writeBytes("aabbbb");
dos.close();

// Читаємо дані
DataInputStream dis = new DataInputStream( new FileInputStream("1.dat") );
System.out.println( dis.readInt() + " " + dis.readDouble() );
// Читаємо рядки
byte bb[] = new byte[256];
int n = dis.read( bb );
System.out.println( new String( bb, 0 , n ) );
dis.close();
```

Робота з текстовими файлами

Для роботи з текстовими файлами можна використовувати різні класи-потоки, але найбільш простим способом читання текстових даних є клас **Scanner**, який розглядався раніше.

Приклад: читання чисел із файлу.

```
// Створити Scanner із зазначенням кодування файлу
Scanner sc = new Scanner( new File("numbers.txt"),"cp1251" );
// Створити Scanner, що використовує кодування за замовчанням
Scanner sc1 = new Scanner( "numbers.txt" );
while( sc.hasNext() ) {
     if( sc.hasNextInt() ) System.out.print( sc.nextInt() + " " );
}
```

Класи **PrintStream, PrintWriter** надають дуже зручні методи форматованого текстового виведення: `print(), println(), printf(), format()`. Відмінність між цими двома класами полягає тільки в тому, що `PrintWriter` може створювати на основі існуючого об’єкта `Writer`, а `PrintStream` – на основі одного з підкласів `OutputStream`..

Приклад. Використаємо потік `PrintStream` для дозапису тексту в кінець файлу. Для цього спочатку необхідно відкрити файл для дозапису як об'єкт `FileOutputStream`. Якщо відкрити файл по-іншому (наприклад, так: `new PrintStream ("1.txt")`), то весь вміст файлу буде видалено.

```
PrintStream o = new PrintStream( new FileOutputStream("1.txt",true) );
o.println("Appended text ");
o.close();
```

Серіалізація

Серіалізация – це процес збереження стану довільного об'єкта в послідовність байтів; ***десеріалізація*** – це процес відновлення об'єкта з такої послідовності байтів.

Механізм серіалізації в Java застосовний до об'єктів будь-якого класу, причому для того, щоб користуватися серіалізацією, в самому класі не потрібно практично нічого змінювати. Це дуже зручно, оскільки можна легко зберегти стан програми (тобто всі створені в ній об'єкти) в файл, або ж передати цю інформацію мережею.

Java має два способи серіалізації об’єктів:

- ***Стандартний*** – реалізувати інтерфейс `java.io.Serializable`, який не має методів, але дозволяє читати і записувати об'єкти класу в спеціальні потоки `ObjectOutputStream` і `ObjectInutStream`.

- ***Розширений*** – реалізувати в своєму класі інтерфейс `java.io.Externalizable`, який містить два явних методи для серіалізації і десеріалізації:
  `writeExternal(ObjectOutput oo)`
  `readExternal(ObjectInput oi).`

Розглянемо перший метод, який є простішим і підходить для більшості практичних задач.

Отже, для серіалізації об'єктів маємо спеціальні класи-потоки: `ObjectInputStream` і `ObjectOutputStream`. Подібно до того, як методи класів `DataInputstream` і `DataOutputstream` читають і записують дані простих типів, методи `ObjectInputStream` і `ObjectOutputStream` дозволяють читати і записувати в потік будь-які об'єкти (в тому числі масиви і рядки).

Користуватися цими класами настільки ж легко, як і потоками введення-виведення примітивних типів, однак серіалізація має багато тонкощів, які потрібно розуміти для уникнення важко вловимих помилок.

За замовчанням для об'єктів класу серіалізація недоступна. Так зроблено тому, що серіалізація об'єкта порушує його безпеку. Для пояснення сказаного уявімо собі ситуацію, коли користувач вашого класу серіалізує об'єкт в масив (тобто зберігає його у вигляді послідовності байтів), а потім переписує деякі елементи масиву, що представляють закриті (private) поля об'єкта і десеріалізує об'єкт зі зміненими полями, і таким чином змінює стан об’єкта неприпустимим способом.

На цьому прикладі видно, що серіалізація крім користі може нести і загрози. Тому програміст повинен явно повідомляти компілятору, що він хоче зробити клас серіалізованим. Для цього достатньо реалізувати інтерфейс **Serializable**. Цей інтерфейс не містить методів, тому окрім слів `implements Serializable` більше нічого писати не треба.

Розглянемо приклад:

```
// Клас, що серіалізується
class My implements Serializable {

     // Будь-які поля і методи

}
```

Тепер для збереження стану об’єкта `MyClass` в файл достатньо створити потік `ObjectOutputStream` і скористатися його методом `writeObject()`.

```
// ...
// Використання класу MyClass
MyClass a = new MyClass(), b;              // Створюємо об’єкт  a та посилання b

// Серіалізація
FileOutputStream fos =                     // Створюємо потік виведення в файл
      new FileOutputStream("temp.out");
ObjectOutputStream oos =                   // Зв’язуємо з ним потік серіалізації
      new ObjectOutputStream(fos);

oos.writeObject(b);                        // Записуємо об’єкт в потік
oos.close();                               // Закриваємо потік
```

Десеріалізація відбувається так само просто, як і серіалізація. Потрібно тільки дотримуватися порядку читання об’єктів з потоку (тобто читати об'єкти в тому ж порядку, в якому вони записувалися в потік). Необхідно створити потік `ObjectOutputStream` і скористатися його методом `readObject()`, який створює об'єкт на підставі даних з потоку і повертає посилання типу `Object`. Перш ніж присвоювати отримане посилання змінній типу `MyClass`, необхідно виконати приведення типу:

```
// Десеріалізація
```

```
ObjectInputStream ois = new ObjectInputStream(       // Створюємо потік
                                new FileInputStream("temp.out") );
b = (MyClass) ois.readObject( ois );      // Зчитуємо  об'єкт
                                          // Перетворення типу обов'язкове
ois.close();                              // Закриваємо потік
```

В процесі серіалізації в вихідний потік виводяться всі нестатичні поля об'єкта, незалежно від прав доступу до них, а також відомості про клас цього об'єкта, необхідні для його правильного відновлення при десеріалізації. Байт-коди методів класу не серіалізуются.

Якщо об'єкт має посилання на інші об'єкти, то ці об'єкти теж повинні бути серіалізовані. Ці об'єкти можуть також мати посилання на нові об'єкти, і тоді всі пов'язані з ними об'єкти, знову-таки, повинні бути серіалізовані, і так далі. Таким чином серіалізація одного об'єкта часто призводить до серіалізації цілого дерева об'єктів, пов'язаних між собою складним чином. Метод `writeObject()` розпізнає ситуацію, коли в дереві два посилання вказують на один об'єкт, і виводить такий об'єкт в вихідний потік тільки один раз. До того ж він розпізнає посилання, замкнуті в кільце, і уникає зациклення при серіалізації. Але все ж, залишається багато випадків, в яких дерево об'єктів розростається настільки, що робить серіалізацію неприйнятною – наприклад, так відбувається при спробі серіалізувати компоненти форм GUI. Тому стандартний механізм серіалізації не є універсальним рішенням.

Всі класи об'єктів, що входять в описане вище дерево, а також всі їх внутрішні класи, повинні реалізувати інтерфейс `Serializable`, інакше буде викинуто виключення класу `NotSerializableException`, і процес серіалізації перерветься.

Багато стандартних класів SDK реалізують інтерфейс `Serializable`. Необхідно також враховувати, що всі нащадки таких класів також успадковують реалізацію. Наприклад, клас `java.awt.Component` реалізує інтерфейс `Serializable`, значить, все графічні компоненти можна серіалізувати (але з зазначених вище причин робити це не варто).

Процес серіалізації можна повністю налаштувати під свої потреби, перевизначивши методи введення-виведення або скориставшись допоміжними класами. Можна також взяти весь процес на себе, реалізувавши не інтерфейс `Serializable`, а інтерфейс `Externalizable`. В табл. 5.1 наведені відмінності між `Serializable` і `Externalizable`.

Табл. 5.1. Способи серіалізації.

Стандартний – реалізувати інтерфейс java.io.Serializable, який не має методів, дозволяє читати і записувати об'єкт в потоки `ObjectOutputStream` і `ObjectInutStream`.	***Розширений*** – реалізувати інтерфейс java.io.Externalizable, який містить два явних методи: `writeExternal(ObjectOutput)` і `readExternal(ObjectInput)`.

`class My implements Serializable { … }`
`My a = new My(), b;`
`FileOutputStream fos =`
`  new FileOutputStream("temp.out");`
`ObjectOutputStream oos =`
`  new ObjectOutputStream(fos);`
`oos.writeObject(b);`
`oos.close();`
 	
`ObjectInputStream ois = new ...;`
`b = (My) ois.readObject( ois );`
`ois.close();`
	`class My implements Externalizable {`
	` ...`
	` public void writeExternal(ObjectOutput o){`
	`  o.writeInt( i );`
	`  o.writeObject( str );  }`
	`}`
	`My a = new My(), b;`
	`ObjectOutputStream oos = new ...;`
	`  b.writeExternal( oos );`
	`  oos.close();`
	`ObjectInputStream ois = new ...;`
	`  b = new My().readExternal( ois );`
	`  ois.close();`

При використанні стандартного способу серіалізації (`Serializable`) необхідно завжди враховувати його приховані ***особливості***:

1. Він використовує рефлексію для виділення полів даних з об'єкта класу. За рахунок цього працює повільно.

2. Він не викликає конструктор об'єкта, що десеріалізується.

3. Якщо батьківський клас не реалізує `Serializable`, то його поля не зберігаються при серіалізації, а при десеріалізації для нього викликається конструктор без параметрів.

4. При десеріалізації необхідно контролювати, щоб набір полів (і методів) в класі не змінився**.** Якщо в новій версії програми змінити що-небудь в оголошенні класу (навіть порядок оголошення методів), то об'єкти, серіалізовані більш ранніми версіями цієї ж програми, будуть відновлюватися з помилками. Для вирішення цієї проблеми в кожен клас, який реалізує інтерфейс `Serializable`, на стадії компіляції додається ще одне поле

```
private static final long serialVersionUID;
```

Це поле містить унікальний ідентифікатор версії серіалізованого класу. Воно обчислюється за вмістом класу – полями, порядком їх оголошення, методами і порядком оголошення методів. Відповідно, за будь-якої зміни в оголошенні класу це поле змінить своє значення. При десеріалізації об'єкта значення цього поля автоматично порівнюється з наявним у класі. Якщо значення не збігаються, то генерується виключення `java.io.InvalidClassException`.

При використанні **Externalizable** ситуація інша. У цьому випадку вся логіка серіалізації реалізується явно. При десеріалізації спочатку автоматично *викликається конструктор без параметрів*, а потім вже для створеного об'єкта викликається метод `readExternal()`, який повинен заповнити всі поля даних з потоку. Докладний розгляд використання цього механізму виходить за межі даного курсу.

Додаткові посилання за темою:

Базова серіалізація: http://habrahabr.ru/post/60317/

Тонкощі Srializable і Externalizable: http://www.skipy.ru/technics/serialization.html

Маніпулювання файлами і директоріями

В Java існує два способи маніпулювати файлами і директоріями:

1) за допомогою класу java.io.File,

2) за допомогою класу `java.nio.file.Files` і інтерфейсу `java.nio.file.Path`

Клас **File** представляє абстрактний шлях до файлу або директорії. Цей шлях зберігається у вигляді рядка (сам файл при цьому не обов'язково існує фізично).

Створення. У конструкторі цього класу вказується шлях до файлу або каталогу:

```
File f = new File( "path/name.ext" );
```

Конструктор не перевіряє, чи існує файл з таким ім'ям, тому після створення об'єкта слід це перевірити за допомогою метода `exists()`.

Виділення частин шляху:

Клас `File` має кілька методів для отримання різних частин шляху до файлу, наприклад:

```
String s = f.getAbsolutePath() + " " + f.getPath() + " " + f.getName();
```

Ці методи не мають відношення до фізично існуючих тек на диску, а лише розбирають рядок, що зберігається всередині змінної `File`. Тому, наприклад, виклик

```
new File("../").getParent()
```

поверне `null`, т. я. в даному випадку об'єкт `File` не зберігає імені батьківської директорії.

Отримати повний шлях до файлу:

```
String s1 = f.getCanonicalPath();
```

Цей метод, на відміну від попередніх, викликає функції операційної системи, щоб перетворити відносний шлях в повний фізичний шлях. Тому він може генерувати виключення.

Отримання інформації про файл

Клас `File` містить багато методів, що дозволяють дізнатися про різні властивості файлу або директорії.

Перш за все, логічними методами `isFile()`, `isDirectory()` можна з'ясувати, чи є шлях, вказаний в конструкторі, шляхом до файлу або директорії (папки).

Для директорії можна отримати її вміст – список імен файлів і вкладених директорій методом `list()`, що повертає масив рядків `String[]`. Можна отримати той же список у вигляді масиву об'єктів `File[]` методом `listFiles()`. Можна вибрати зі списку тільки деякі файли, реалізувавши інтерфейс `FileNameFilter` і звернувшись до методу `list(FileNameFilter filter)`.

Для файлу можна отримати його довжину в байтах методом `length()`, час останньої модифікації в секундах з 1 січня 1970 р методом `lastModified()`. Якщо файл не існує, ці методи повертають нуль.

Логічні методи `canRead()`, `canWrite()`, `canExecute()` показують права доступу до файлу.

Покажемо роботу цих методів на прикладі:

```
File f = new File( "path/name.ext" );
f.exists();                  // Чи існує файл/директорія з таким іменем?
f.isDirectory();             // Перевірка на директорію
f.length();                  // Для файла – його розмір, для директорії - 0
f.lastModified();            // Дата останньої зміни (в мс. з 1.01.1970)
f.canRead(), f.canWrite(), f.canExecute();   // Перевірка дозволів доступу
f.isHidden();                // Чи є файл прихованим?
File ff[] = f.listFiles();   // Для директорій – отримати список файлів
```

Операції над файлами та порожніми директоріями:

Якщо директорія із зазначеним в конструкторі `File()` шляхом не існує, її можна створити логічним методом `mkDir()`. Цей метод повертає true, якщо директорію вдалося створити. Логічний метод `mkDirs()` створює також і всі неіснуючі директорії, зазначені у шляху до файлу.

Порожня директорія видаляється методом `delete()`. Якщо директорія непорожня, то `delete()` нічого не робить – спершу необхідно видалити весь вміст директорії.

Файл можна перейменувати логічним методом `renameTo(File newName)` або видалити логічним методом `delete()`. Ці методи повертають true, якщо операція пройшла вдало.

Якщо файл з вказаним в конструкторі шляхом не існує, його можна створити логічним методом `createNewFile()`, що повертає true, якщо файл не існував, і його вдалося створити, і false, якщо файл вже існував.

Розглянемо тепер кілька прикладів використання методів роботи з файлами і директоріями для вирішення найбільш поширених задач.

Приклад 1. Виведемо основні властивості файлу і директорії. Результат запуску цієї програми показаний на рис. 5.3. Зверніть увагу, що для директорії ця програма виводить тільки вкладені файли першого рівня вкладеності.

```
import java.io.*;
class FileTest {
```

```
    public static void main(String[] args) throws IOException{
         // Змінюємо кодування потоку виведення
         PrintWriter pw = new PrintWriter(
              new OutputStreamWriter(System.out, "Cp866"), true);

         // Створюємо шлях до файлу
         File f = new File("FileTest.Java");
         pw.println();

         // Отримуємо властивості файла
         pw.println("Файл \"" + f.getName() + "\" " +
              (f.exists()?"":"не ") + "існує");
         pw.println("Ви " + (f.canRead()?"":"не ") + "можете читати файл");
         pw.println("Ви " + (f.canWrite()?"":"не ") + "можете записувати в файл");
         pw.println("Розмір файла " + f.length() + " б");
         pw.println() ;

         // Створюємо шлях до директорії
         File d = new File(" D:\\jdk1.3\\MyProgs ");
         pw.println("Вміст каталога:");
         // Можна запитувати вміст тільки якщо
         // d існує та являється директорією
         if (d.exists() && d.isDirectory()) {
              String[] s = d.list();
              for (int i = 0; i < s.length; i++)
                   pw.println(s[i]);
              }
         }
    }
}
```

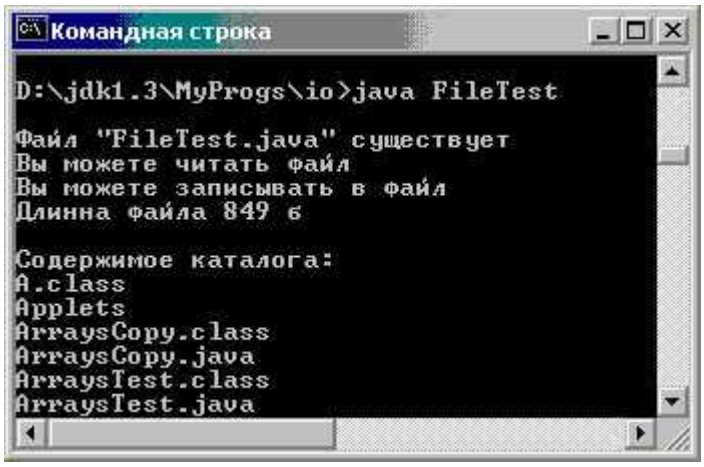

Рис. 5.3 Результат роботи програми

Приклад 2. Виведемо на екран вміст директорії з урахуванням всіх її піддиректорій і визначимо її загальний розмір (в байтах). Будемо виводити назви файлів і кількість днів з часу їх останньої зміни.

```
    try {
      // Створюємо об'єкт File і зв'язуємо його з директорією нашого проекту
      File dir = new File( "../" );
      System.out.println( dir.getCanonicalPath() );
      Викликаємо рекурсивну функцію перегляду вмісту директорії list()
      System.out.println( "Total size - "+list( dir, 0 )+" "+dir.length());
    } catch (Exception e) { e.printStackTrace(); }

    // Рекурсивна функція перегляду вмісту директорії
    static long list( File f, int depth ) {

         // Виводимо пробіли щоб показати рівень вкладеності
```

```
        for( int i = 0; i<depth; i++ ) System.out.print("  ");

        // Визначаємо кількість днів з часу останньої зміни файлу
        long days = (System.currentTimeMillis() - f.lastModified())
                          / 1000 / 60 / 60/ 24;
        System.out.println( f.getName() + " - " + days + " days ago" );

        // Якщо це директорія, то рекурсивно викликаємо list() для кожного
        // вкладеного файла
        if( f.isDirectory() ) {
             long sz = 0;
             for( File f1 : f.listFiles() )
                  sz += list( f1, depth+1 );   // Сумуємо розміри вкладених файлів
             return sz;                        // і повертаємо результат
        } else {
             return f.length();                // Для звичайного файлу –
                                               // повертаємо його розмір
        }
    }
```

Приклад 3. Напишемо метод видалення непорожньої директорії. Перш ніж викликати метод delete() для директорії перевіримо, чи має вона вкладені файли і директорії. Якщо має – то рекурсивно запустимо наш метод для кожного вкладеного файлу і директорії.

```
static void deleteDir( File dir ) {
    try {
        if( dir.isDirectory() )
             for( File f : dir.listFiles() )  deleteDir( f );
        dir.delete();
    } catch (Exception e) {
        System.out.println("Cannot delete " + dir.getAbsolutePath());
        // getAbsolutePath() не генерує виключень
    }
}
```

Клас `File` не містить методу копіювання файлу. Копіювання реалізується через потокові операції читання-запису (за допомогою класів `FileInputStream` і `FileOutputStream`, як це робилося на початку цієї лекції), або за допомогою класу `Files`, як буде показано нижче.

В цілому клас **Files** надає можливості, аналогічні `File`, але має наступні відмінності:

- для представлення файла (а точніше шляху до файлу) використовує інтерфейс **Path**;
- не потрібно створювати об'єкти класу `Files`, оскільки всі його методи є статичними;
- `Files` має деякі додаткові методи, аналогів яким немає в класі `File`, наприклад:
  `copy(), move();`
- багато методів мають додаткові параметри, що управляють їх роботою, наприклад:

```
Files.move(srcPath, destPath, REPLACE_EXISTING);
```

Приклад 4. Скопіюємо файл з допомогою класу `Files`.

```
static void copyFile( String name, String newName ) {
    try {

        // Створюємо об'єкти Path і зв'язуємо їх із шляхами
        Path p1 = new File(name).toPath(), p2 = new File(newName).toPath();

        // Перевіряємо, чи не є один із шляхів директорією
        if( Files.isDirectory(p1) || Files.isDirectory(p2) ) return;

        // Копіювати можна тільки якщо вихідний файл існує
        if( Files.exists(p1) )
             // Якщо другий файл вже існує, то його треба спочатку видалити
```

```
            if( Files.exists(p2) ) Files.delete(p2);
            // Тепер можна копіювати
            Files.copy( p1, p2 );   // FileNotFoundExxeption,
                                    // FileAlreadyExistsException !!!!
    } catch (Exception e) { e.printStackTrace(); }
}
```

Тема 6. Створення GUI. Бібліотека Swing

Лекція 8

Можливості та особливості різних бібліотек GUI

Раніше ми створювали програми, які були пов'язані з консоллю і запускались з командного рядка. Такі програми називаються *консольними додатками*. Вони розробляються для виконання на серверах, там, де не потрібен інтерактивний зв'язок з користувачем. Програми іншого призначення тісно взаємодіють із користувачем, сприймають сигнали від клавіатури і миші, працюють в графічному середовищі – це *додатки з графічним інтерфейсом користувача* (GUI).

Кожний GUI-додаток повинен створити хоча б одне вікно, в якому буде відбуватися його робота, і зареєструвати його в графічній оболонці операційної системи, щоб вікно могло взаємодіяти з операційною системою і іншими вікнами: перекриватися, переміщатися, змінювати розміри, згортатися до панелі задач і т. п.

Існує багато графічних систем: MS Windows, X Window System, Mac OS X та ін. Кожна з них має свої правила побудови вікон і їх компонентів: меню, полів введення, кнопок, списків, смуг прокрутки. Ці правила складні і заплутані, а графічні API зазвичай містять сотні функцій. Для полегшення створення вікон і їх компонентів для різних мов програмування написані спеціальні бібліотеки класів: MFC, Motif, OpenLook, Qt, Tk, Xview, OpenWindows та безліч інших. Кожен клас такої бібліотеки описує відразу цілий графічний компонент, керований методами цього та інших класів.

В технології Java справа ускладняється тим, що додатки повинні працювати однаково в будь-якому графічному середовищі (або по меншій мірі у багатьох таких середовищах). Таким чином, потрібна бібліотека класів, незалежна від конкретної графічної системи.

У першій версії JDK задачу вирішили наступним чином: були розроблені інтерфейси, що містять методи роботи з графічними об'єктами. Програми Java можуть їх використовувати для створення вікон, розміщення і переміщення графічних об'єктів, зміни їх розмірів, організації взаємодії графічних об'єктів. З іншого боку, для роботи з екраном в конкретній ОС і конкретній графічній оболонці ці інтерфейси реалізовані по-різному, з використанням API цієї оболонки, а також за допомогою графічних бібліотек даної операційної системи. Такі інтерфейси, за якими скриті різні реалізації для кожної графічної системи, були названі ***peer-інтерфейсами***.

Бібліотека класів Java на основі peer-інтерфейсів отримала назву **AWT** (Abstract Window Toolkit). При виведенні на екран об'єкта, створеного в додатку Java за допомогою peer-інтерфейсу, насправді створюється та відображається на екрані парний йому (peer-to-peer) об'єкт графічної підсистеми ОС. Ці два парних об'єкти тісно взаємодіють під час роботи програми. Тому графічні об'єкти AWT в кожному графічному середовищі мають вигляд, характерний для цього середовища: в MS Windows, Motif, OpenLook, OpenWindows – всюди вікна, створені в AWT, виглядають як "рідні" вікна відповідного графічного середовища.

Саме через таку реалізацію peer-інтерфейсів та інших методів, написану, головним чином, на мові C++, доводиться для кожної платформи випускати свій варіант JDK.

Наслідком описаних особливостей AWT є і основні її недоліки:

- AWT може використовувати тільки стандартні елементи ОС, в зв'язку з чим AWT не дозволяє створювати віджети довільної форми;
- набір елементів управління, загальних для всіх ОС, невеликий.

У версії JDK 1.1 бібліотека AWT була перероблена. Додана можливість створення компонентів, повністю написаних на Java і не залежних від peer-інтерфейсів. Такі компоненти стали називати ***"легкими"*** (lightweight) на відміну від компонентів, реалізованих через peer-інтерфейси, названих ***"важкими"*** (heavy).

В той же час була створена потужна бібліотека "легких" компонентів Java, названа **Swing**. У ній були переписані всі реалізації інтерфейсів бібліотеки AWT, так що бібліотека Swing може

використовуватися самостійно не дивлячись на те, що всі класи з неї розширюють класи і інтерфейси бібліотеки AWT.

"Легкі" компоненти всюди виглядають однаково, зберігають заданий при створенні вигляд (***look and feel***). «Look» визначає зовнішній вигляд компонентів, а «Feel» – їх поведінку. Більш того, є можливість вже після запуску додатку змінити його зовнішній вигляд на інший, обравши інший компонент look-and-feel. Ця цікава особливість "легких" компонентів отримала назву PL&F (Pluggable Look and Feel або “plaf”). Основні стандартні *LookAndFeel* компоненти наступні:

- *CrossPlatformLookAndFeel* – вигляд компонентів не залежить від ОС та її налаштувань;
- *SystemLookAndFeel* – всі компоненти використовують налаштування L&F, встановлені в ОС.

Все це стало можливим, тому що компоненти Swing є «Lightweight», тобто створюються як зображення на поверхні батьківського вікна без використання компонентів ОС. Як наслідок, в Swing-додатку вікнами є тільки компоненти верхнього рівня (наприклад, JFrame), і всі віджети малюються на них. Природньо, що при такому способі відтворення Swing працює повільніше, ніж AWT і SWT. На сьогоднішній день проблеми з продуктивністю Swing зведені до мінімуму. Тому зараз Swing є основною бібліотекою користувацького інтерфейсу в Java. Використання Swing майже завжди краще, за винятком тих випадків, коли потрібна зворотна сумісність з Java 1.

Також в Java 2 бібліотека AWT значно розширена додаванням нових засобів малювання, виведення текстів і зображень, які отримали назву Java 2D, і засобів, що реалізують переміщення об’єктів методом Drag and Drop. Ці ж засоби малювання використовуються і в Swing-додатках. Крім того, в Java 2 включені нові методи введення-виведення Input Method Framework і засоби зв'язку з додатковими пристроями введення-виведення, такими як світлове перо або клавіатура Брайля, названі Accessibility.

Разом всі ці засоби Java 2: AWT, Swing, Java 2D, DnD, Input Method Framework і Accessibility – склали бібліотеку графічних засобів Java, названу **JFC (Java Foundation Classes)**. Опис кожного з цих засобів досить великий, тому розглянемо лише основи роботи з бібліотекою Swing.

Перш ніж йти далі, потрібно відзначити, що існує і третя стандартна бібліотека графічних компонентів – ***SWT (Standard Widget Toolkit)***. Вона не є самостійною графічною бібліотекою в повному розумінні, а являє собою крос-платформову оболонку для графічних бібліотек конкретних платформ. Наприклад, під Linux вона використовує бібліотеку Gtk+. SWT отримує доступ до ОС-бібліотек через Java Native Interface і API оболонки. За рахунок цього SWT дозволяє отримати звичний зовнішній вигляд віджетів для кожної ОС. Використання SWT робить Java-додаток більш ефективним, але робить його більш залежним від операційної системи і устаткування, вимагає ручного звільнення ресурсів і в деякій мірі порушує концепцію платформи Java.

Компонент і контейнер

Основне поняття графічного інтерфейсу користувача (ГІК) – ***компонент*** (component) графічної системи. В загальному розумінні це слово означає просто складову частину, елемент чогось, але в GUI це поняття набагато конкретніше: воно означає окремий, повністю визначений елемент, який можна використовувати в графічному інтерфейсі незалежно від інших елементів. Прикладами компонентів є: поле введення, кнопка, рядок меню, смуга прокрутки, радіо-кнопка. Саме вікно програми – теж є компонентом. Компоненти можуть бути і невидимими, наприклад, панель, яка об'єднує компоненти, сама теж є компонентом.

В AWT компоненти представлені об'єктами класу **`Component`** та його численних нащадків. У класі `Component` зібрані загальні методи роботи з будь-яким компонентом GUI. Цей клас є центром бібліотеки AWT, над якою, в свою чергу, надбудовується бібліотека Swing (рис. 6.1).

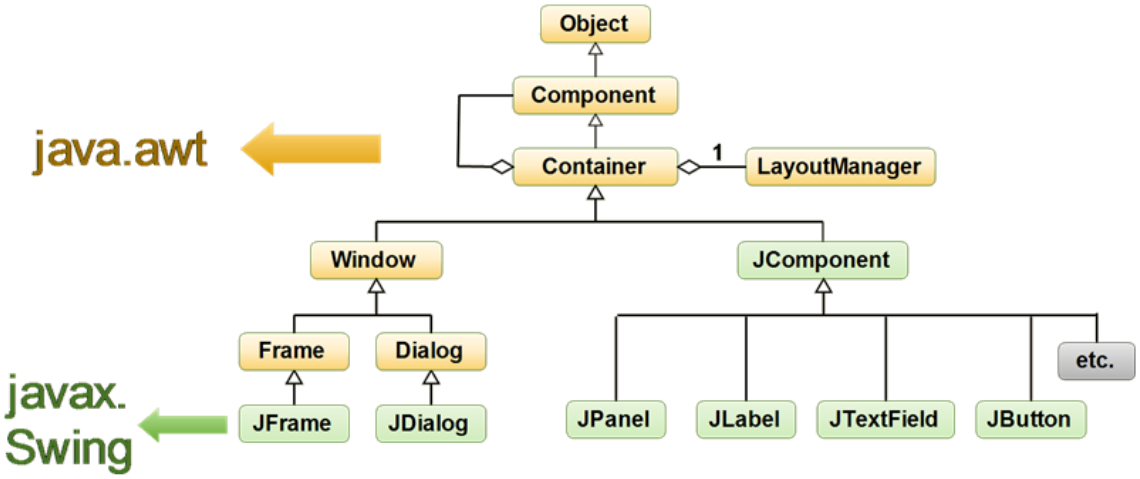

Рис. 6.1. Ієрархія класів компонентів

Кожен компонент перед виведенням на екран поміщається в *контейнер* (`Container`). Контейнер "знає", як розмістити дочірні компоненти на екрані. Зверніть увагу, що прямий спадкоємець цього класу – клас `JComponent` – є вершиною ієрархії компонентів бібліотеки Swing. Це означає, що *всі компоненти Swing є контейнерами* і можуть містити дочірні компоненти.

Створивши компонент (тобто об'єкт класу `Component` або його нащадка), слід додати його до попередньо створеного об'єкту класу `Container` або його розширення одним з методів `add()`. Оскільки клас `Container` розширює клас `Component`, в контейнер поряд з компонентами можна поміщати інші контейнери, досягаючи таким чином більшої гнучкості розташування компонентів.

Основне вікно програми, що активно взаємодіє з ОС, необхідно побудувати за правилами графічної системи. Воно повинно переміщатися по екрану, змінювати розміри, реагувати на дії миші і клавіатури. У вікні повинні бути, як мінімум, такі стандартні компоненти.

- *Рядок заголовка* (title bar), з лівого боку якого необхідно розмістити кнопку контекстного меню, а з правого – кнопки згортання і розгортання вікна і кнопку закриття програми.
- Необов'язковий *рядок меню* (menu bar) з випадними пунктами меню.
- Горизонтальна і вертикальна *смуги прокрутки* (scrollbars).
- Вікно повинне бути оточене *рамкою* (border), що реагує на дії миші.

Вікно зі всіма цими компонентами описано в класі `Frame`, а Swing-клас `JFrame` безпосередньо його успадковує. Таким чином, щоб створити вікно, потрібно або успадкувати власний клас від `JFrame`, або просто створити об'єкт класу `JFrame` в своєму класі програми. В обох випадках потім обов'язково окремо створити дочірні компоненти і помістити їх у вікно. У першому випадку це робиться в конструкторі вашого класу-вікна, а в другому – відразу після створення об'єкта `JFrame` (зазвичай в конструкторі класу додатка або в методі `main()`). Далі докладно розглянемо обидва способи.

Вікно JFrame

Клас `JFrame` представляє собою вікно і може використовуватися як для головного віна програми, так і для інших допоміжних вікон, що відкриваються під час роботи програми. Вікно `JFrame` має рамку і заголовок (з кнопками «Згорнути», «На весь екран» і «Закрити»). Воно може змінювати розміри і переміщатися екраном.

Конструктор `JFrame()` без параметрів створює порожнє вікно; інший конструктор `JFrame(String title)` створює порожнє вікно з заголовком title. Щоб написати просту програму, що виводить на екран порожнє вікно, необхідні ще три методи:

`setSize(int width, int height)` – встановлює розміри вікна. Якщо не поставити розміри, вікно буде мати нульову висоту незалежно від того, що в ньому знаходиться і користувачеві після запуску доведеться розтягувати вікно вручну. Розміри вікна включають не тільки робочу область, але також і рамку та рядок заголовка.

`setDefaultCloseOperation (int operation)` – дозволяє вказати дію, яку необхідно виконати, коли користувач закриває вікно натисканням на хрестик. Зазвичай в програмі є одне або кілька вікон, при закритті яких програма припиняє роботу. Для того, щоб запрограмувати таку поведінку, слід в параметр `operation` передати константу `EXIT_ON_CLOSE`, описану в класі `JFrame`.

`setVisible(boolean visible)` – цей метод керує видимістю вікна. Нове створене вікно за замовчанням невидиме. Щоб відобразити його на екрані, треба викликати даний метод з параметром true. Якщо викликати його з параметром false, вікно знову стане невидимим.

Тепер можемо написати програму, яка створює вікно, виводить його на екран і завершує роботу після того, як користувач закриває вікно. Зверніть увагу, при створенні GUI-додатків в кожній програмі нам буде потрібно імпортувати класи пакетів `java.swing` і `java.awt`.

```
import javax.swing.*;
public class MyClass {
     public static void main (String [] args) {
          JFrame myWindow = new JFrame("Пробне вікно");
          myWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          myWindow.setSize(400, 300);
          myWindow.setVisible(true);
     }
}
```

Як правило, перед відображення вікна необхідно зробити набагато більше дій, ніж в цій простій програмі. Необхідно створити елементи управління, налаштувати їх зовнішній вигляд, розмістити в потрібних місцях вікна. Крім того, в програмі може бути багато вікон і налаштовувати їх всі в методі `main()` незручно і неправильно, оскільки це порушує принцип інкапсуляції. Логічніше було б, щоб кожне вікно займалося своїми розмірами і вмістом самостійно. Тому класична структура програми з вікнами виглядає наступним чином:

В файлі SimpleWindow.java оголошуємо клас вікна:

```
public class SimpleWindow extends JFrame {
     SimpleWindow(){
          super("Пробне окно");
          setDefaultCloseOperation(EXIT_ON_CLOSE);
          setSize(250, 100);
     }
}
```

В головному класі програми створюємо екземпляр вікна та відображаємо його:

```
public class Program {
     public static void main (String [] args) {
          JFrame myWindow = new SimpleWindow();
          myWindow.setVisible(true);
     }
}
```

З цього спрощеного прикладу видно, що вікно описується в окремому класі, який є спадкоємцем `JFrame` і налаштовує свій зовнішній вигляд і поведінку в конструкторі (першою командою викликається конструктор суперкласу). Метод `main()` міститься в іншому класі, відповідальному за управління ходом програми. Кожен з цих класів дуже простий, кожен займається своєю справою, тому в них легко розбиратися, і такий код легко супроводжувати (тобто удосконалювати за необхідності).

Зверніть увагу, що метод `setVisible()` не викликається в класі `SimpleWindow`, що цілком логічно: за тим, де розташовані дочірні компоненти і які розміри вони мають, стежить саме вікно, а от приймати рішення про те, в який момент вікно виводиться на екран – це вже справа головного класу програми.

Розглянемо тепер приклад, більш наближений до реальних додатків.

Приклад. Каркас мінімального GUI-додатка.

```
public class MyWnd {              // Клас додатка

     private JFrame frame;        // Головне (і єдине) вікно

     public static void main(String[] args) {
          EventQueue.invokeLater(new Runnable() {          // Для створення вікна
               public void run() {                         // як правило запускається
                    try {                                  // окремий потік.
                         MyWnd window = new MyWnd();       // Створюємо вікно
                         window.frame.setVisible(true);    // та відображаємо його.
                    } catch (Exception e) {
                         e.printStackTrace();
                    }
               }
          });
     }

     public MyWnd() {   initialize(); }         // Конструктор класу додатку

     private void initialize() {                // Створення вікна та компонентів
          frame = new JFrame();                 // Створюємо саме вікно
          frame.setBounds(100, 100, 450, 300);
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          // Тут код створення дочірніх компонентів та додавання для них обробників
     }
}
```

Зверніть увагу, що зазвичай процес створення вікна запускається окремим потоком, тому що створення складного вікна може займати багато часу, і це не повинно впливати на роботу основного потоку програми.

Контейнери

Swing має контейнери верхнього рівня, тобто вікна, які відповідають вікнам в ОС, і прості контейнери, які не відповідають ніяким об'єктам ОС, а просто вимальовуються на поверхні контейнерів верхнього рівня

Контейнери верхнього рівня:

- `JFrame` – вікно додатку;
- `JDialog` – діалог додатку;
- `JColorChooser` – діалог вибору кольору;
- `JFileChooser` – діалог вибору файлів та директорій;
- `FileDialog` – діалог вибору файлів та директорій (awt компонент);
- `JApplet` – головне вікно апплета.

Прості контейнери:

- `Jpanel` – проста панель для групування елементів, включаючи вкладені панелі;
- `JtoolBar` – панель інструментів (як правило це кнопки);
- `JScroolPane` – панель прокрутки, що дозволяє продивлятись вміст дочірнього елемента;
- `JDesktopPane` – контейнер для створення віртуального робочого стола або додатків на основі MDI (multiple-document interface);
- `JEditorPane`, `JTextPane` – контейнери для відображення складного документа як HTML або RTF;
- `JTabbedPane` – контейнер для керування закладками;

- `JSplitPane` – контейнер, що має дві області, розділені перегородкою, та дозволяє змінювати їх розмір, переміщуючи перегородку.

Все контейнери є нащадками класу `Container` та наслідують від нього ряд корисних методів:

- `add(Component component)` – добавляє компонент до контейнера;
- `remove(Component component)` – видаляє компонент з контейнера;
- `removeAll()` – видаляє всі компоненти з контейнера;
- `getComponentCount()` – повертає кількість компонентів у контейнері.

В класі `Container` визначено близько двох десятків методів для управління набором дочірніх компонентів. Ці методи схожі на методи класу-колекції. По суті контейнер і є колекцією, але колекцією візуальних компонентів, яка крім зберігання елементів займається також їх просторовим розташуванням і відображенням. Зокрема, будь-який контейнер має метод `getComponentAt(int x, int y)` – він повертає компонент, до якого належить точка з заданими координатами відносно лівого верхнього кута контейнера. Немає потреби розглядати абстрактний контейнер, тому відразу перейдемо до його найбільш часто використовуваного нащадка – класу `JPanel`.

Панель вмісту

Елементи управління ніколи не розміщуються безпосередньо у вікні. Для їх розміщення служить панель вмісту, що займає весь робочий простір вікна. Посилання на цю панель можна отримати за допомогою методу `getContentPane()` класу `JFrame`. За допомогою методу `add()` можна додати на неї будь-який елемент управління. Метод `setContentPane(JPanel panel)` дозволяє замінити панель вмісту вікна.

В наступних прикладах для простоти будемо використовувати тільки один елемент управління – кнопку. Цей елемент описується класом `JButton` і створюється конструктором з параметром типу `String` – надписом. Додамо кнопку в панель вмісту нашого вікна командами:

```
JButton newButton = new JButton(); getContentPane().add(newButton);
```

В результаті отримаємо вікно з кнопкою, що зображене на рис. 6.2. Кнопка займає всю доступну площу вікна. Такий ефект корисний не у всіх програмах, і тому далі ми розглянемо різні способи керування розташуванням елементів у контейнері.

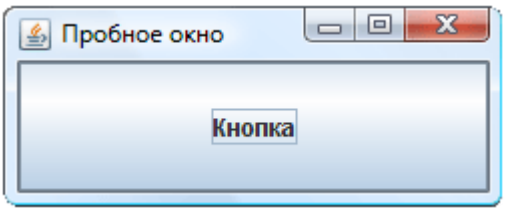

Рис. 6.2. Просте вікно із кнопкою

Клас JPanel

Панель `JPanel` – це прямокутний простір, на якому можна розміщувати інші компоненти за допомогою методів, успадкованих від класу `Container`.

У попередньому прикладі додана на панель вмісту кнопка зайняла весь її простір. Це відбувається не завжди. Насправді кожна панель має так званий *менеджер розміщення*, який визначає стратегію взаємного розташування елементів, що додаються на панель. Менеджер розміщення можна замінити методом `setLayout(LayoutManager manager)`. Але щоб передати в цей метод потрібний параметр, необхідно знати класи ієрархії `LayoutManager`.

Верстка форми і класи розміщень (layouts)

Для автоматичного позиціонування і розрахунку розмірів дочірніх елементів контейнери використовують спеціальні об'єкти – *менеджери розміщення* (або просто «розміщення»). Нижче наведено список стандартних класів розміщень:

- *FlowLayout* – розміщує елементи по порядку в тому ж напрямку, що і орієнтація контейнера (зліва направо за замовчанням), застосовуючи один з п'яти видів вирівнювання, зазначеного при створенні менеджера. Даний менеджер використовується за замовчанням в більшості контейнерів;
- *GridLayout* – розміщує елементи у вигляді таблиці. Кількість стовпців і рядків вказується при створенні менеджера. За замовчанням він має один рядок та число стовпців, рівне кількості елементів;
- *BorderLayout* – розміщує кожен елемент в одну з п'яти областей, яка зазначається при додаванні елемента в контейнер: "North", "South", "East", "West" і "Center";
- *SpringLayout* – розміщує елементи у відповідності з обмеженнями, які задані в кожному з них.

Якщо встановити для контейнера в якості розміщення порожній об'єкт викликом `setLayout(null)`, то всі дочірні компоненти необхідно буде позиціонувати вручну.

Менеджер послідовного розміщення FlowLayout

Найпростшим менеджером розміщення є `FlowLayout`. Він розміщує компоненти строго по черзі, зліва направо і зверху вниз, подібно тексту на сторінці, і в залежності від розмірів панелі. Якщо черговий елемент не поміщається в поточному рядку, то він переноситься на наступний рядок. Розглянемо приклад, в якому змінимо конструктор класу `SimpleWindow` (див. приклад вище) наступним чином:

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    panel.setLayout(new FlowLayout());
    panel.add(new JButton("Кнопка"));
    panel.add(new JButton("+"));
    panel.add(new JButton("-"));
    panel.add(new JButton("Кнопка з довгим написом"));
    setContentPane(panel);
    setSize(250, 100);
}
```

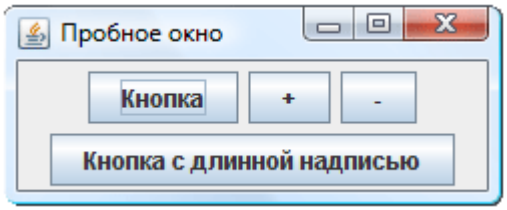

Рис. 6.3. Демонстрація FlowLayout

У вікні, що з'являється після запуску програми (рис. 6.3), чотири кнопки розташовані як слова у тексті із вирівнюванням по центру. Ефект буде краще помітний, якщо змінювати розміри вікна під час роботи програми.

Проаналізуємо текст прикладу. Об’єкт `FlowLayout` створюється конструктором без параметрів. Слід звернути увагу, що цей об’єкт не зберігається у змінну. Це виправдано в тих випадках, коли в подальшому не буде потреби звертатися до створюваного об'єкту. В нашому випадку після зв'язування об’єкта `FlowLayout` із панеллю вони надалі взаємодіятимуть між собою автоматично. Так само додаємо на панель нові кнопки. Подібна конструкція, коли об’єкт

створюється тільки для одноразового використання, дуже поширена в програмах із GUI. Якщо потрібно буде звертатися до створених кнопок в інших методах програми, то їх потрібно зберегти в змінні в класі `SimpleWindow`.

Менеджер граничного розміщення BorderLayout

Цей менеджер розділяє панель на п'ять областей: центральну, верхню, нижню, праву і ліву. У кожну з цих областей можна додати рівно по одному компоненту, причому компонент буде займати всю відведену для нього область. Верхні і нижні компоненти будуть розтягнуті по ширині, правий і лівий – по висоті, а компонент, для якого відведено центр, буде розтягнутий до повного заповнення простору, що залишився.

При додаванні елемента на панель з менеджером розміщення `BorderLayout` необхідно в методі `add()` додатково вказувати область, в якій він розміститься (за замовчуванням елемент буде поміщено до центральної області). П'ять областей `BorderLayout` умовно позначаються назвами сторін світу: "North", "South", "East", "West" і "Center"; їм відповідають константи, визначені в класі `BorderLayout`: `NORTH`, `SOUTH`, `EAST`, `WEST` і `CENTER`.

Панель вмісту в першому прикладі має саме таке розміщення. Кнопка за замовчуванням була додана в центральну область, саме тому вона займала весь робочий простір вікна. Додамо тепер кнопки в кожну з п'яти областей (рис. 6.4):

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    getContentPane().add(new JButton("Кнопка"), BorderLayout.NORTH);
    getContentPane().add(new JButton("+"), BorderLayout.EAST);
    getContentPane().add(new JButton("-"), BorderLayout.WEST);
    getContentPane().add(
           new JButton("Кнопка з довгим написом"), BorderLayout.SOUTH);
    getContentPane().add(new JButton("В ЦЕНТР!"));
    setSize(250, 100);
}
```

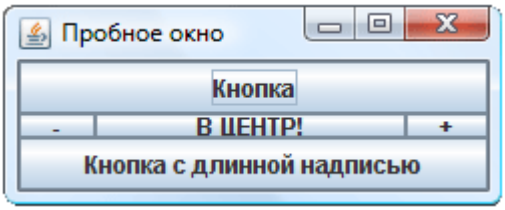

Рис. 6.4. Демонстрація BorderLayout

Принцип роботи `BorderLayout` – добре видно, якщо змінювати розміри вікна. Дане розміщення не випадково використовується в панелі вмісту за замовчанням. Більшість програм користуються областями по краях вікна, щоб розташувати в них панелі інструментів, рядок стану і т. п. При цьому обмеження на один компонент в центральній області абсолютно не суттєве, адже цим компонентом може бути панель з багатьма дочірніми компонентами і з будь-яким власним менеджером розташування.

Менеджер табличного розміщення GridLayout

`GridLayout` розбиває контейнер на комірки однакового розміру і контейнер стає схожим на таблицю. Кожен дочірній компонент займає одну комірку. Комірки заповнюються по черзі, починаючи з верхньої лівої (рис. 6.5).

Такий менеджер, на відміну від розглянутих вище, створюється конструктором з параметрами (чотири цілих числа). Необхідно вказати кількість стовпців, рядків, а також відстань між комірками по горизонталі і по вертикалі, наприклад:

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(2, 3, 5, 10));
    panel.add(new JButton("Кнопка"));
    panel.add(new JButton("+"));
    panel.add(new JButton("-"));
    panel.add(new JButton("Кнопка з довгим написом"));
    panel.add(new JButton("ще кнопка"));
    setContentPane(panel);
    setSize(250, 100);
}
```

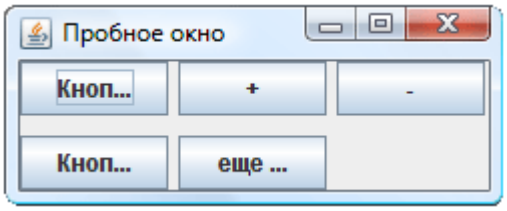

Рис. 6.5. Демонстрація GridLayout

Менеджер блочного розміщення BoxLayout і клас Box

Менеджер `BoxLayout` розміщує елементи в рядок або в стовпчик. Зазвичай для роботи з даним сервісом використовують допоміжний клас `Box`, що представляє собою панель, для якої вже налаштоване блочне розміщення. Створюється така панель не конструктором, а одним з двох статичних методів, визначених в класі `Box`: `createHorizontalBox()` і `createVerticalBox()`.

Елементи, додані на панель з блоковим розміщенням, розташовуються один за іншим. Відстань між елементами за замовчанням нульова. Однак замість компонента можна додати невидимий «буфер», єдине завдання якого – розсовувати сусідні елементи, забезпечуючи між ними задану відстань. Горизонтальний буфер створюється статичним методом `createHorizontalStrut(int width)`, а вертикальний – методом `createVerticalStrut(int height)`. Обидва названі методи визначені в класі `Box`, а цілочисловий параметр в кожному з них визначає розмір буфера.

Крім того, до `BoxLayout` можна додати ще один спеціальний елемент – своєрідну «пружину». Якщо розмір контейнера буде більшим, ніж необхідно для оптимального розміщення всіх елементів, ті з них, які здатні розтягуватися, будуть намагатися заповнити додатковий простір собою. Якщо ж розмістити серед елементів одну або кілька «пружин», додатковий вільний простір буде розподілятися на ці проміжки між елементами. Горизонтальна і вертикальна пружини створюються відповідно методами `createHorizontalGlue()` і `createVerticalGlue()`.

Особливості роботи цього менеджера покажемо на наочному прикладі. Розташуємо чотири кнопки вертикально, поставивши між двома центральними «пружину», а між іншими – буфери в 10 пікселів (рис. 6.6).

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Box box = Box.createVerticalBox();
    box.add(new JButton("Кнопка"));
    box.add(Box.createVerticalStrut(10));
    box.add(new JButton("+"));
    box.add(Box.createVerticalGlue());
    box.add(new JButton("-"));
    box.add(Box.createVerticalStrut(10));
    box.add(new JButton("Кнопка з довгим написом"));
    setContentPane(box);
    setSize(250, 100);
```

```
}
```

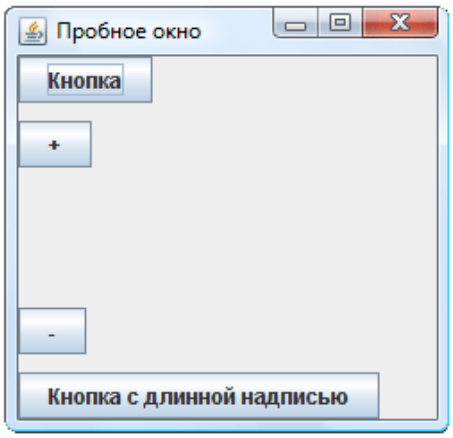

Рис. 6.6. Демонстрація `BoxLayout`

Особливості вирівнювання елементів

У попередньому прикладі з вертикальною панеллю всі кнопки виявилися вирівняні по лівому краю. Таке вирівнювання по горизонталі прийнято за замовчанням. Для того, щоб встановити інші правила вирівнювання компонентів, використовуються методи `setAlignmentX(float alignment)` – вирівнювання по горизонталі і `setAlignmentY(float alignment)` – вирівнювання по вертикалі. Як параметр найпростіше використовувати константи, визначені в класі `JComponent`. Для вирівнювання по горизонталі служать константи `LEFT_ALIGNMENT` (по лівому краю), `RIGHT_ALIGNMENT` (по правому краю) і `CENTER_ALIGNMENT` (по центру). Для вирівнювання по вертикалі – `BOTTOM_ALIGNMENT` (по нижньому краю), `TOP_ALIGNMENT` (по верхньому краю) і `CENTER_ALIGNMENT` (по центру).

Однак вирівнювання працює трохи інакше, ніж очікується. Щоб це виявити, змінимо попередній приклад, вирівнявши третю кнопку по правому краю. Для цього замінимо рядок

```
box.add(new JButton("-"));
```

на три інших:

```
JButton rightButton = new JButton("-");
rightButton.setAlignmentX(JComponent.RIGHT_ALIGNMENT);
box.add(rightButton);
```

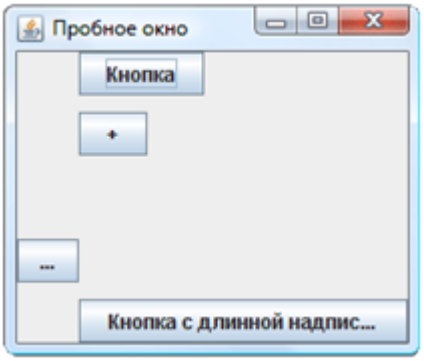

Рис. 6.7. Демонстрація вирівнювання компонентів

Після запуску програми побачимо вікно (рис. 6.7), в якому кнопки розташовані не так, як очікувалося. Ми звикли, що вирівнювання по правому краю притискає об'єкт до правого краю контейнера, але в даному випадку перемістилися всі елементи, причому кнопка з вирівнюванням

по правому краю виявилася самою лівою. Насправді при вирівнюванні по правому краю компонент не притискається до правого краю контейнера; замість цього він притискається правим краєм до невидимої лінії вирівнювання. Всі інші компоненти притискаються до цієї лінії своїм лівим краєм (за замовчуванням), тому і виходить спостережуваний ефект.

Єдина складність для розробника в тому, що не завжди легко зрозуміти, де саме пройде умовна лінія вирівнювання. Її положення залежить від розмірів і параметрів вирівнювання всіх елементів контейнера. Однак корисно запам'ятати просте правило: якщо всі елементи в контейнері вирівняні однаково, то лінія вирівнювання знаходиться біля краю контейнера (як це і було в попередньому прикладі).

Можна також поекспериментувати з вертикальною панеллю, задаючи різне вирівнювання для її елементів: логіка вирівнювання буде схожою.

Розміщення компонентів вручну

Якщо встановити для контейнера в якості розміщення порожній об'єкт викликом `setLayout(null)`, то положення дочірніх компонентів не буде обраховуватись автоматично і тоді можна вручну задати їхні розміри та положення. При цьому координати кожного компонента необхідно вказати явно, оскільки вони ніяк не залежать від розмірів панелі і від положення інших компонентів. За замовчанням координати компонента встановлюються рівними нулю (тобто він розташований в лівому верхньому кутку контейнера). Розмір елемента (ширина і висота) за замовчанням також дорівнюють нулю, тому якщо не задати розміри явно, елемент відображатися не буде.

Для задання положення і розмірів будь-який компонент має наступні методи:

```
setLocation(int x, int y);
setLocation(Point point);
setSize(int width, int height),
setSize(Dimension size)
```

Обидва метода `setLocation()` задають положення лівого верхнього кута компонента відносно лівого верхнього кута контейнера. Координати точки кута можна задати двома способами: двома цілими числами, або об'єктом класу **Point**. Об'єкт `Point` має дві властивості – `x` і `y` – і створюється конструктором `Point(int x, int y)`. Останній спосіб є найбільш вживаним, оскільки багато методів графічної бібліотеки видають координати точок у вигляді об'єктів `Point`. Наприклад, за допомогою методу `getLocation()` можна дізнатись поточне положення будь-якого компонента. Тоді наступний рядок помістить елемент b в точності в те місце, яке займає елемент a:

```
b.setLocation(a.getLocation());
```

В методах `setSize()` також маємо два способи задання розмірів. Об'єкт **Dimension**, аналогічно `Point`, зберігає два цілі числа, має дві властивості – `width` і `height`, – а також конструктор з двома параметрами `Dimension(int width, int height)`. Отримати поточний розмір компонента можна методом `getSize()`, що повертає об'єкт класу `Dimension`. Тоді наступний рядок зробить елемент b точно такого ж розміру, як і елемент a:

```
b.setSize(a.getSize());
```

Автоматичне визначення розмірів компонентів

Всі менеджери розміщення ігнорують явно задані координати і розміри дочірніх компонентів. Менеджер сам визначає їх координати і розміри. Але все ж деякі властивості компонентів можуть впливати на те, як саме менеджер буде їх розміщувати.

Ми бачили раніше, що в деяких випадках компоненти заповнюють весь доступний їм простір: в `BorderLayout` компонент заповнював всю центральну область, а в разі `GridLayout` – всю комірку таблиці. У разі `FlowLayout`, навпаки, компоненти не збільшувалися понад деякого

розміру, навіть якщо поруч залишався вільний простір. Подібними аспектами поведінки компонентів найлегше керувати, використовуючи їх властивості.

Кожен компонент має три пари розмірів: *мінімально допустимий, максимально допустимий і бажаний*. Вони доступні через властивості `minimumSize`, `preferredSize` і `maximumSize` типу `Dimension` (а значить є відповідні методи – «геттери» і «сеттери»). Наприклад, кнопка `JButton` за замовчуванням має мінімальний розмір – нульовий, максимальний розмір – не обмежений (що позначається у властивості `maximumSize` як `width = -1` і `height = -1`), а бажаний розмір залежить від напису на кнопці (обчислюється як розмір тексту напису плюс розміри полів).

Різні менеджери розміщення використовують ці властивості по-різному:

`FlowLayout` завжди використовує бажані розміри елементів (`preferredSize`);

`BorderLayout` використовує бажану ширину для правого і лівого, а також бажану висоту для верхнього і нижнього компонентів; інші розміри підганяються під розміри контейнера;

`GridLayout` намагається підігнати розміри всіх компонентів під розмір комірок;

`BoxLayout` орієнтується на бажані розміри.

У випадках, коли компонент повинен заповнити весь доступний простір, або навпаки, коли місця недостатньо, більшість (але не всі) менеджери розміщення враховують властивості `minimumSize` і `maximumSize`. Ці властивості за правильного налаштування обмежують стиснення компонентів при зменшенні розмірів контейнера, в якому вони знаходяться, а також обмежують їх розтягнення, коли розмір контейнера збільшується.

Багато компонентів, наприклад кнопки, взагалі не повинні збільшуватися і зменшуватися, а повинні зберігати свій бажаний розмір. Для них використовується наступний простий прийом:

```
component.setMinimumSize(component.getPreferredSize());
component.setMaximumSize(component.getPreferredSize());
```

Розрахунок ризміру вікна

У попередніх прикладах розмір вікна задавався явно методом `setSize()`. Однак на практиці важко визначити найбільш підходящі розміри вікна, особливо якщо вікно має багато дочірніх компонентів, і їх розміри визначаються різними менеджерами розміщення. Безумовно, найбільш підходящим буде варіант, за якого всі елементи вікна мають бажані розміри або близькі до них.

Всі контейнери-вікна мають метод `pack()`, який автоматично підбирає і встановлює розміри вікна в залежності від його вмісту. При цьому враховуються бажані розміри всіх дочірніх компонентів, вимоги менеджерів розміщення, і вікно приймає мінімальний розмір, достатній для розміщення всіх дочірніх компонентів.

Зауважте, якщо головна панель вікна (властивість `contentPane`) не має менеджера розміщення (`setLayoutManager(null)`), то вікно не має алгоритму для обчислення свого бажаного розміру і в цьому випадку метод `pack()` не працює.

Рамки

Зазвичай рамка використовується щоб візуально відокремити компонент від оточення. Рамку можуть мати панелі, написи, кнопки і багато інших компонентів. Деякі з них мають рамку за замовчуванням, інші – не мають, але в будь-якому випадку можна програмно керувати зовнішнім виглядом рамки. Рамка встановлюється методом `setBorder(Border border)`. Як параметр потрібно передати об'єкт класу `Border`, який описує створювану рамку і може мати безліч різних властивостей. `Border` – це абстрактний клас, тому для створення рамок використовуються його спадкоємці:

- EmptyBorder – порожня рамка, що не промальовується, але дозволяє створити відступи по краях компонента. Розміри відступів задаються в конструкторі чотирма цілими числами.
- TitledBorder – рамка з заголовком, для якої необхідно задати текст заголовка і інші необов'язкові параметри, які керують розташуванням заголовка і його стилем.
- EtchedBorder – рамка з ефектом вдавлення або опуклості.

- BevelBorder – об'ємна рамка (опукла або вдавлена). Можна налаштувати кольори, що використовуються для отримання об'ємних ефектів.
- SoftBevelBorder – те ж, що і BevelBorder, але дозволяє додатково округлити кути рамки.
- LineBorder – рамка у вигляді лінії. Можна вибирати колір, товщину і стиль лінії, а також округлити кути.
- MatteBorder – рамка у вигляді повторюваного малюнка.
- CompoundBorder – об'єкт цього класу об'єднує дві рамки, що передаються в якості параметрів конструктора, в одну складну рамку.

Всі перераховані класи описані в пакеті `javax.swing.border`.

Для прикладу створимо шість панелей з різними рамками і розмістимо їх у вигляді таблиці. Щоб не повторювати дії зі створення нової панелі, винесемо їх в окремий метод:

```
private JPanel createPanel(Border border, String text) {
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.add(new JButton(text));
    panel.setBorder(
              new CompoundBorder( border, new EmptyBorder(12, 12, 12, 12) ) );
    return panel;
}
```

Метод `createPanel()` створює панель, весь простір якої займає одна кнопка. В якості параметрів передається напис для кнопки і рамка, яку необхідно використовувати для панелі. Зверніть увагу, що передана рамка комбінується з «порожньою» рамкою `EmptyBorder` – цей прийом дозволяє забезпечити відступи між рамкою і внутрішнім вмістом панелі (в нашому випадку – кнопкою). Тепер створимо кілька схожих панелей з різними типами рамок (рис. 6.8).

```
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(2, 3, 5, 10));
    panel.setBorder(new EmptyBorder(12, 12, 12, 12));
    panel.add( createPanel(
              new TitledBorder("Рамка с заголовком"), "TitledBorder" ));
    panel.add( createPanel(
              new EtchedBorder(), "EtchedBorder" ));
    panel.add( createPanel(
              new BevelBorder(BevelBorder.LOWERED), "BevelBorder" ));
    panel.add( createPanel(
              new SoftBevelBorder(BevelBorder.RAISED), "SoftBevelBorder" ));
    panel.add( createPanel(
              new LineBorder(Color.ORANGE, 4), "LineBorder" ));
    panel.add( createPanel(
              new MatteBorder(new ImageIcon("1.png")), "MatteBorder" ));
```

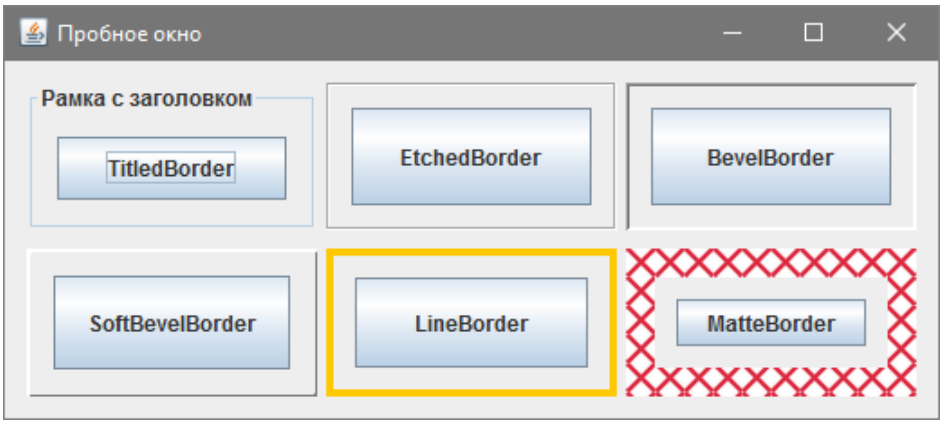

Рис. 6.8. Демонстрація різних видів рамок

У цьому прикладі використані два нових класи: `Color` і `ImageIcon`.

Об'єкт класу **Color** представляє колір. Такий об'єкт можна створити, вказавши параметри кольору в форматі RGB, наприклад, `new Color(128,255,128)` – це світло–зелений колір. У класі `Color` безліч стандартних кольорів визначено у вигляді констант, наприклад: `Color.RED`, `Color.BLACK`, `Color.ORANGE` та ін.

Об'єкт класу **ImageIcon** описує графічне зображення, яке можна використовувати в якості «іконки» в GUI. Такий об'єкт легко створити, вказавши шлях до файлу зображення. У прикладі використовується відносне ім'я файлу "1.png" – такий файл повинен знаходитися в директорії проекту. Крім `ImageIcon` є також споріднені класи `Image` і `BufferedImage`: – всі вони так чи інакше представляють зображення. Відмінність між ними полягає в тому, як саме зображення може створюватися і використовуватися в програмі.

Поняття і принципи usability. Розташування елементів екранної форми

У найширшому сенсі термін ***юзабіліті*** означає ступінь зручності користування тим чи іншим предметом. Стосовно ж до інтерфейсу користувача, під юзабіліті мають на увазі сукупність наступних факторів:

– логічність і простота розташування різних графічних елементів;

– простота і зручність навігації;

– продуманість розташування елементів управління;

– загальна легкість сприйняття інтерфейсу людиною.

Існує безліч правил і рекомендацій юзабіліті, які стосуються як візуального дизайну інтерфейсу, так і логіки роботи з ним. Ми ж торкнемося лише найбільш простих правил, що регулюють взаємне розташування візуальних компонентів. Для дотримання цих правил достатньо правильно налаштувати менеджери розміщення і параметри окремих компонентів.

1. Елементи управління в діалоговому вікні бувають двох видів – введення-виведення (редагування) даних і підтвердження такого введення. Елементи, пов’язані із редагуванням (або внесенням) інформації, повинні бути розташовані зверху, а кнопки підтвердження – знизу в горизонтальний ряд.

2. Розміщення редагуючих елементів управління слід виконувати зверху вниз в порядку важливості редагованої ними інформації. Обов'язково також задати ***порядок табуляції*** (tab order) для переходу між елементами за допомогою клавіатури. В порядку табуляції спочатку повинні йти елементи редагування, потім кнопка підтвердження, потім кнопка скасування дії, а потім – інші кнопки (додаткові дії).

3. Необхідно витримувати відстань між елементами. Тісно пов'язані елементи (такі як текстове поле і підпис до нього) повинні відстояти один від одного на 6 пікселів. Логічно згруповані елементи – на 12 пікселів (наприклад кнопки "Ok" і "Cancel", або поля для введення логіна і пароля). Всі інші елементи повинні перебувати на відстані 17 пікселів один від одного. Не слід забувати і про відступи між елементами управління і рамкою вікна.

Работа с GUI редактором IDEA

IDE IntelliJ IDEA має багатий набір засобів для швидкого і зручного створення графічного інтерфейсу користувача із мінімальною необхідністю написання програмного коду. Всі ці засоби об'єднані в редакторі GUI.

Робота з GUI редактором IDEA докладно описана в книзі:

Давыдов С., Ефимов А. IntelliJ IDEA. Профессиональное программирование на Java. – ***С. 398 – 428.***

Скачати цю книгу можна за посиланням: http://www.twirpx.com/file/142621/.

GUI редактор WindowBuilder для IDE Eclipse

Eclipse не містить вбудованого GUI-редактора. Його функції беруть на себе різні доповнення.

WindowBuilder – це найбільш популярний візуальний дизайнер інтерфейсів на Swing, GWT та SWT для середовища Eclipse. Він виконаний у вигляді плагіна Eclipse. На даний момент WindowsBuilder поширюється вільно.

Сторінка проекту: http://www.eclipse.org/windowbuilder/

Установка WindowsBuilder в Eclipse

1) Завантажуємо WindowBuilder з офіційного сайта:

http://www.eclipse.org/windowbuilder/download.php.

Для своєї версії Eclipse вибираємо "**Zipped Update Site**", вибираємо дзеркало для завантаження і зберігаємо .zip-файл, наприклад "WB_v1.9.1_UpdateSite.zip".

2) Розпаковуємо архів, наприклад в папку "D:\Eclipse\WB_v1.6.0"

3) В IDE Eclipse в меню вибираємо **Help® Install new Software® Add**.

4) У вікні "**Add Repository**" в полі "**Name**" задаємо "WindowsBuilder", тиснемо кнопку "**Local**" і вибираємо папку з установочними файлами – "D:\Eclipse\WB_v1.6.0". Тиснемо "**ОК**".

5) У вікні "Install" відмічаємо всі встановлювані компоненти: GroupLayout, InfraStructure, SwingDesigner, натискаємо "**Next**" і надалі використовуємо інструкції майстра установки.

Створення екранної форми

1) Створюємо Java-проект (File® New® Java Project).

2) Далі натискаємо правою кнопкою миші на створеному проекті, вибираємо **New® Other** (або ж натискаємо Ctrl+N) і у вікні шукаємо папку **WindowBuilder**, а в ній вибираємо "**Swing Designer**"® "**Application Window**".

В результаті буде створений клас, в тілі якого згенерується заголовок коду для створення стандартної екранної форми.

Для того, щоб переключитися в графічний режим редагування форми, необхідно натиснути кнопку "**Design**", розташовану знизу від робочої області Eclipse. У режимі Design можна перетягувати елементи на форму і задавати їх властивості.

Лекція 9

Класи графічних компонентів

Розглянемо перелік основних графічних компонентів Swing. Всі вони також можуть використовуватися і як контейнери, оскільки успадковуються від класу `java.awt.Container`.

- *JLabel* – мітка, напис;
- *JButton* – кнопка;
- *JTextField* – однорядкове текстове поле;
- *JTextArea* – багаторядкове текстове поле з можливістю форматування (як у форматі HTML, так і в форматі RTF);
- *JTextPane* – багаторядковий текстовий надпис з можливістю форматування;
- *JPasswordField* – текстове поле для скритого введення;
- *JCheckBox* – кнопка-прапорець;
- *JRadioButton* – перемикачі, радіо-кнопки, які часто використовуються з компонентом *ButtonGroup*;
- *JComboBox* – випадний список;
- *JProgressBar* – компонент для наглядного відображення числа в заданому діапазоні;
- *JSlider* – компонент, що дозволяє вибрати значення із заданого діапазону;
- *JSpinner* – компонент, що дозволяє вибрати значення із вказаної послідовності;
- *JList* – список;
- *JTable* – таблиця;
- *JTree* – дерево.

Компоненти поділяють на прості і складні. Прості компоненти не вимагають для своєї роботи ніяких додаткових об'єктів і структур даних. Прикладами таких компонентів є кнопки, написи, однорядкові поля редагування.

Складні компоненти, навпаки, використовуються для відображення великих обсягів структурованих даних. Для внутрішнього представлення даних вони використовують спеціальні об'єкти – *моделі даних*. Сюди відносяться списки, дерева, таблиці, текстові області з форматуванням.

У цій лекції будемо знайомитися з простими компонентами, складні ж будуть розглядатися далі в даному курсі.

Прості компоненти Swing

Клас JComponent

Всі візуальні компоненти бібліотеки Swing успадковані від класу `JComponent`. Сам цей клас є абстрактними і безпосередньо не використовується, але всі візуальні компоненти успадковують його методи. Розглянемо найбільш корисні з них.

`setEnabled(boolean enabled)` використовується для управління активністю компонента. В результаті виклику цього методу з параметром false компонент переходить в неактивний стан. Для кожного спадкоємця `JComponent` «неактивний» стан може бути перевизначений по-різному. Наприклад, неактивна кнопка не натискається, не реагує на мишу і відображається сірим кольором. Метод `isEnabled()` повертає true, якщо елемент активний і false в іншому випадку.

`setVisible(boolean visible)` управляє видимістю компонента. Цей метод вже використовувався для відображення вікна `JFrame`. Більшість елементів управління, на відміну від вікна, за замовчуванням є видимими, тому після їх створення в попередніх прикладах метод `setVisible()` не викликався. Метод `isVisible()` повертає false, якщо елемент невидимий і true в іншому випадку.

Метод `setBackground(Color color)` дозволяє змінити колір фону компонента. Однак ефект спостерігається лише для непрозорих компонентів (деякі компоненти, наприклад мітка `JLabel`, за замовчуванням є прозорими). Прозорістю компонента можна керувати за допомогою метода `setOpaque(boolean opaque)`, в якому параметр true означає непрозорість. Методи `getBackground()` і `isOpaque()` відповідно повертають поточний колір фону і непрозорість компонента.

Мітка JLabel

Мітка – це один з найпростіших компонентів. Зазвичай мітка являє собою звичайний текст, який виводиться в заданому місці вікна і використовується для виведення допоміжної текстової інформації: підписів до інших елементів або інструкцій та попереджень для користувача. У Swing мітка може відображати форматований текст, а також може відображати картинку.

Текст і картинка для мітки зазвичай задаються в її конструкторі. Вона має кілька конструкторів з різними параметрами, зокрема:

`JLabel(String text)` – створює мітку з написом text;

`JLabel(Icon image)` – створює мітку із картинкою image;

`JLabel(String text, Icon image, int align)` – створює мітку з написом text і картинкою image. Третій параметр задає вирівнювання тексту разом із картинкою. Для нього може бути використана одна з констант, описаних в інтерфейсі `SwingConstants`: `LEFT`, `RIGHT`, `CENTER`.

Для прикладу створимо вікно з міткою за допомогою третього конструктора (рис. 6.9). Як і раніше, будемо використовувати два класи, один з яких назвемо `SimpleWindow` і успадкуємо його від класу вікна `JFrame`. У його конструкторі будуть створюватися і розміщуватися всі елементи вікна. Другий клас буде створювати це вікно і відображати його на екрані (його код буде таким же, як і в попередніх прикладах).

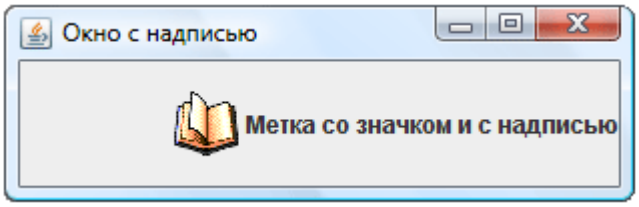

Рис. 6.9. Мітка із картинкою та текстом

Введемо в конструктор класу SimpleWindow наступний код:

```
SimpleWindow() {
    super("Вікно з написом");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JLabel label = new JLabel("Мітка із картинкою і написом",
                              new ImageIcon("book.png"), SwingConstants.RIGHT);
    getContentPane().add(label);
    pack();
}
```

Щоб переконатися, що вирівнювання по правому краю працює, необхідно трохи розтягнути вікно, щоб ширина мітки стала більше оптимальної.

У бібліотеці Swing всі компоненти можуть відображати форматований текст в форматі HTML. Для цього необхідно, щоб рядок, що встановлюється в якості напису, починався з тега `<html>`. Після цього можна використовувати будь-які теги мови HTML версії 3.2, і вони будуть перетворюватися у відповідні атрибути форматування (рис. 6.10). У цьому легко переконатися, змінивши в попередньому прикладі рядок з викликом конструктора на наступний:

```
JLabel label = new JLabel("<html> До цієї мітки застосовно" +
    "HTML-форматування, включаючи: <ul><li> <i>курсив</i>," +
```

```
"<li><b>напівжирний</b> <li><font size = +2> збільшення розміру </font>" +
"<li>маркований список </ul>");
```

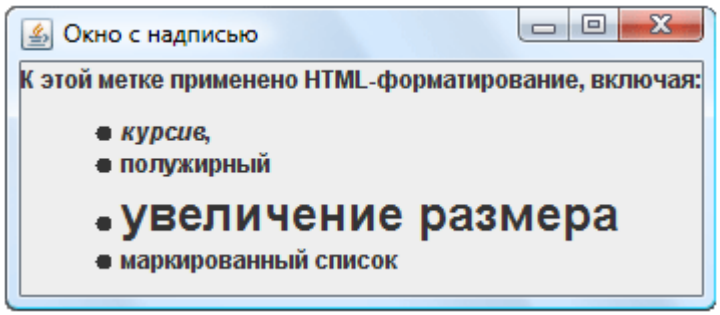

Рис. 6.10. Мітка з форматуванням

Перелічимо найбільш вживані методи класу JLabel:

`getText()` – повертає поточний текст напису мітки;

`setText(String text)` – задає новий текст напису;

`getIcon()` – повертає значок мітки;

`setIcon(Icon image)` – встановлює новий значок. В якості значка зазвичай використовується об'єкт вже знайомого нам простого класу `ImageIcon`.

`getVerticalAlignment()`, `setVerticalAlignment(int align)`, `getHorizontalAlignment()`, `setHorizontalAlignment(int align)` – ці чотири методи дозволяють отримати поточне або встановити нове вирівнювання (по горизонталі і вертикалі) мітки щодо її меж. Можливі положення описані в інтерфейсі `SwingConstants`.

`getVerticalTextPosition()`, `setVerticalTextPosition(int align)`, `getHorizontalTextPosition()`, `setHorizontalTextPosition(int align)` – ці чотири методи дозволяють отримати поточне або встановити нове вирівнювання тексту щодо значка. Можливі положення описані в інтерфейсі `SwingConstants`.

`getIconTextGap()`, `setIconTextGap(int gap)` – дозволяють отримати або задати відстань між текстом і значком мітки в пікселях.

Кнопка JButton

Кнопка являє собою прямокутник з текстом (та/або значком), який можна «натиснути» за допомогою миші або клавіатури, щоб викликати певну дію. Кнопка створюється одним з п'яти конструкторів, зокрема `JButton()`, `JButton(String text)`, `JButton(Icon icon)`, `JButton(String text, Icon icon)`, параметри яких говорять самі за себе.

Крім звичайного значка можна призначити кнопці ще кілька значків для різних станів. Метод `setRolloverIcon(Icon icon)` дозволяє задати значок, який буде з'являтися при наведенні на кнопку миші; `setPressedIcon(Icon icon)` – являє собою значок для кнопки в натиснутому стані і `setDisableIcon(Icon icon)` – це значок для неактивної кнопки. Кожному з цих методів відповідає метод get.

Метод `setMargin(Insets margin)` дозволяє задати величину відступів від тексту напису на кнопці до її полів. Об'єкт класу `Insets`, який передається в цей метод, може бути створений конструктором з чотирма цілочисловими параметрами, які задають величину відступів: `Insets(int top, int left, int bottom, int right)`. Метод `getMargin()` повертає величину поточних відступів у вигляді об'єкта того ж класу.

Всі методи класу `JLabel`, описані раніше, присутні і в класі `JButton`. За допомогою цих методів можна змінювати значок і текст напису на кнопці, а також управляти їх взаємним розташуванням один щодо одного і щодо краю кнопки (з урахуванням відступів).

За допомогою методів

```
setBorderPainted(boolean borderPainted),
```

```
setFocusPainted(boolean focusPainted),
setContentAreaFilled(boolean contentAreaFilled)
```

можна відключати (параметром false) і включати назад (параметром true) промальовування рамки, промальовування фокуса (кнопка, на якій знаходиться фокус, виділяється пунктирним прямокутником) і зафарбовування кнопки в натиснутому стані.

Для прикладу створимо кнопку зі значком і з написом, змінимо її відступи і розташування тексту щодо значка: текст буде вирівняний ліворуч і вгору щодо значка (рис. 6.11).

```
SimpleWindow() {
    super("Вікно з кнопкою");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JButton button = new JButton("Кнопка", new ImageIcon("book.png"));
    button.setMargin(new Insets(0, 10, 20, 30));
    button.setVerticalTextPosition(SwingConstants.TOP);
    button.setHorizontalTextPosition(SwingConstants.LEFT);
    getContentPane().add(button);
    pack();
}
```

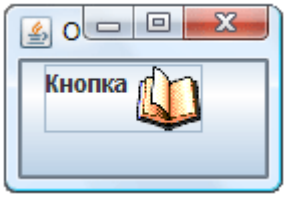

Рис. 6.11. Кнопка зі значком і написом

Компоненти JToggleButton, JCheckBox, JRadioButton

`JToggleButton` представляє собою компонент-кнопку, яка може перебувати в двох станах: натиснутому і відпущеному. Якщо користувач натискає таку кнопку, вона змінює свій стан. Таким чином ведуть себе кнопки форматування на панелі інструментів текстового редактора. Кнопка не тільки встановлює або прибирає курсивне зображення в виділеному тексті, але і сигналізує про його наявність чи відсутність.

Основний конструктор – `JToggleButton(String text, Icon icon, boolean selected)` – створює кнопку із заданими написом, значком і поточним станом. Зауважимо, що кнопку можна перевести в потрібний стан програмним шляхом, викликавши метод `setSelected(boolean selected)`. Метод `isSelected()` повертає true, якщо кнопка вибрана (тобто знаходиться в натиснутому стані) і false у протилежному випадку.

Від класу `JToggleButton` успадкований клас `JCheckBox` – «прапорець». Цей клас має точно такий же набір конструкторів і методів, тобто таку саму поведінку. Єдина відмінність між цими класами проявляється у зовнішньому вигляді.

Схожим чином функціонує клас перемикач або радіо-кнопка `JRadioButton`, що зовні виглядає як кружок. Самі по собі компоненти `JCheckBox` і `JRadioButton` поводяться абсолютно однаково (як і їх спільний предок `JToggleButton`), але ці класи використовують по-різному. Зокрема, `JRadioButton` передбачає вибір єдиної альтернативи з декількох можливих: зазвичай кілька таких об'єктів об'єднуються в одну групу (найчастіше ця група візуально позначається рамкою) і при виборі одного з елементів групи попередній обраний елемент переходить в стан «не вибраний». Для цього використовується спеціальний контейнер `ButtonGroup` – група із взаємовиключенням, яка створюється конструктором без параметрів. Якщо додати в один такий контейнер кілька елементів `JRadioButton`, то обраним завжди буде тільки один з них.

Загалом до `ButtonGroup` можуть бути додані не тільки перемикачі, але і прапорці, і кнопки вибору, і навіть звичайні кнопки `JButton`. Але згідно загально прийнятому підходу у групу із

взаємовиключенням слід об'єднувати об'єкти `JRadioButton`, а в деяких випадках `JToggleButton`, але не інші види кнопок.

Метод `add(AbstractButton button)` додає елемент в групу. Метод `getElements()` повертає всі її елементи у вигляді колекції `Enumeration`. Далі можна перебрати елементи такої колекції ітератором і знайти вибраний користувачем елемент.

Розглянемо приклад, в якому створимо дві кнопки вибору, два прапорці і два перемикачі (рис. 6.12). Кнопки вибору і перемикачі об'єднані в групи `ButtonGroup`. Для того, щоб обвести кожну пару елементів рамкою, необхідно розташувати цю пару елементів на окремій панелі.

```
SimpleWindow() {
    super("Приклад з кнопками вибору, прапорцями і перемикачами");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    ImageIcon icon = new ImageIcon("book.png");
    Box mainBox = Box.createVerticalBox();
    Box box1 = Box.createVerticalBox();
    JToggleButton tButton1 = new JToggleButton("Кнопка вибору 1");
    JToggleButton tButton2 = new JToggleButton("Кнопка вибору 2", icon);
    ButtonGroup bg = new ButtonGroup();   // створюємо групу взаємного виключення
    bg.add(tButton1); bg.add(tButton2);   // додаємо кнопки до групи
    box1.add(tButton1); box1.add(tButton2);   // додаємо їх також до панелі box1
    box1.setBorder(new TitledBorder("Кнопки вибору"));
    Box box2 = Box.createVerticalBox();
    JCheckBox check1 = new JCheckBox("Прапорець 1");
    JCheckBox check2 = new JCheckBox("Прапорець 2", icon); box2.add(check1);
    box2.add(check2);   // додаємо флажки на панель box2
    box2.setBorder(new TitledBorder("Прапорці"));
    Box box3 = Box.createVerticalBox();
    JRadioButton rButton1 = new JRadioButton("Перемикач 1");
    JRadioButton rButton2 = new JRadioButton("Перемикач 2", icon);
    bg = new ButtonGroup();   // створюємо групу взаємного виключення
    bg.add(rButton1); bg.add(rButton2);   // робимо радіо-кнопки взаємновиключаючими
    box3.add(rButton1); box3.add(rButton2);   // додаємо радіокнопки на панель box3
    box3.setBorder(new TitledBorder("Перемикачі"));
    mainBox.add(box1);
    mainBox.add(box2);
    mainBox.add(box3);
    setContentPane(mainBox);
    pack();
}
```

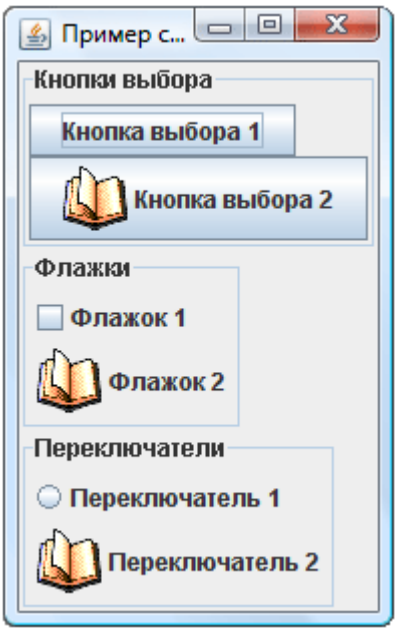

Рис. 6.12. Демонстрація різних видів кнопок

На цьому прикладі ми бачимо, що у прапорців і перемикачів значок замінює індикатор виділення. Однак значок не вказує стан, тому необхідно встановити другий значок для стану "selected" за допомогою метода `setSelectedIcon (Icon icon)`:

```
check2.setSelectedIcon(new ImageIcon("2.png"));
rButton2.setSelectedIcon(new ImageIcon("2.png"));
```

Текстове поле JTextField

Це простий і часто використовуваний компонент, призначений для введення невеликих за обсягом текстових даних в один рядок. Текстове поле створюється одним з конструкторів:

`JTextField(int columns)` – створює порожнє текстове поле, ширина якого достатня для розміщення `columns` символів. Користувач може вводити в текстове поле рядок будь-якої довжини: вона просто буде прокручуватися.

`JTextField(String text)` – текстове поле створюється з початковим текстом text.

`JTextField(String text, int columns)` – дозволяє встановити як ширину, так і початковий текст.

Текст заноситься в поле методом `setText(String text)`. Метод `getText()` повертає вміст текстового поля цілком, а `getText(int offset, int length)` – лише фрагмент тексту.

Підтримується виділення частини тексту як програмним шляхом, так і в результаті дій користувача. При цьому метод `getSelectedText()` дає можливість отримати виділену частину тексту. Заміна виділеного тексту іншим здійснюється за допомогою методу `replaceSelection(String content)`. Методи `getSelectionStart()` і `getSelectionEnd()` повертають межі виділеної ділянки, а методи `setSelectionStart(int start)` і `setSelectionEnd(int end)` змінюють їх.

Метод `getCaretPosition()` повертає позицію курсора (каретки) в текстовому полі, в той час як метод `setCaretPosition(int position)` дозволяє задати її програмно. Метод `setCaretColor(Color color)` дозволяє змінити колір курсора.

Текст в полі за замовчанням вирівнюється за лівим краєм. Змінити це можна методом `setHorizontalAlignment(int align)`, якому в якості параметра передається одна з констант вирівнювання, визначених у цьому ж класі `JTextField`: `LEFT`, `CENTER`, `RIGHT`.

Поле для введення пароля JPasswordField

`JPasswordField` є прямим нащадком `JTextField`, тому для нього справедливо все сказане вище. Відмінність полягає в тому, що весь введений в нього текст прихований: він замінюється зірочками або іншим символом, встановити який дозволяє метод `setEchoChar(char echo)`, а отримати – `getEchoChar()`.

`JPasswordField` застосовується для введення пароля. Метод `getText()` дозволяє отримати введений пароль у вигляді рядка, але користуватися ним не рекомендується, оскільки при бажанні стороння особа може проаналізувати вміст оперативної пам'яті і перехопити пароль. Тому треба в цьому випадку використовувати метод `getPassword()`, який повертає посилання на масив символів `char[]`.

Після того, як введений пароль буде оброблено (наприклад, порівняно з правильним паролем) рекомендується заповнити цей масив нулями, щоб слідів в оперативній пам'яті не залишилося.

Область для введення тексту JTextArea

`JTextArea` також є нащадком `JTextField` і успадковує всі його методи. На відміну від текстового поля, область для введення тексту дозволяє ввести кілька рядків тексту. У зв'язку з цим `JTextArea` пропонує ряд додаткових функцій.

По-перше, можна налаштувати автоматичне перенесення довгих рядків тексту за допомогою методу `setLineWrap(boolean lineWrap)`. Якщо передати параметр true, то довгі рядки будуть переноситися, а не виходити за межі компонента. По-друге, можна налаштувати перенесення слів

за допомогою методу `setWrapStyleWord(boolean wrapStyle)`. Якщо викликати цей метод з параметром true, то слова не будуть розриватися при перенесенні.

Для створення `JTextArea` найчастіше використовують конструктор `JTextArea(int rows, int columns)`, що встановлює видимі висоту (кількість рядків) і ширину (кількість символів) компонента.

Для роботи зі своїм вмістом `JTextArea` додатково пропонує два зручних методи. Це метод `append(String text)`, який додає рядок text в кінець вже наявного тексту та метод `insert(String text, int position)`, що вставляє його в позицію position.

Покажемо роботу розглянутих текстових компонентів на наочному прикладі. Створимо просте вікно, в якому розмістимо три різні текстові поля за допомогою менеджера `BorderLayout` (рис. 6.13).

```
SimpleWindow() {
    super("Приклад текстових компонентів");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JTextField textField = new JTextField("Текстове поле", 20);
    textField.setCaretColor(Color.RED);
    textField.setHorizontalAlignment(JTextField.RIGHT);
    JPasswordField passwordField = new JPasswordField(20);
    passwordField.setEchoChar('$');
    passwordField.setText("пароль");
    JTextArea textArea = new JTextArea(5, 20);
    textArea.setLineWrap(true);
    textArea.setWrapStyleWord(true);
    for (int i = 0; i <= 20; i++) textArea.append("Текст");
    getContentPane().add(textField, BorderLayout.NORTH);
    getContentPane().add(textArea);
    getContentPane().add(passwordField, BorderLayout.SOUTH);
    pack();
}
```

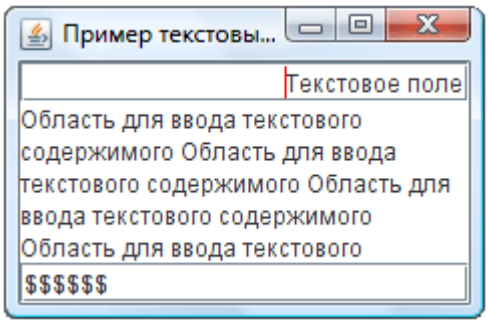

Рис. 6.13. Демонстрація різних компонентів для введення тексту

Замініть по черзі true на false у викликах методів `setLineWrap()` і `setWrapStyleWord()`. При цьому спостерігаються зміни роботи компонента. Також змінюючи розміри вікна можна бачити яким чином текст перебудовується під доступний йому простір.

Панель прокрутки JScrollPane

Спостерігаючи за поведінкою компонента `JTextArea` в попередньому прикладі, легко можна виявити проблеми, які виникають, коли текст не вміщується в поле. Залежно від використовуваного менеджера розміщення текст або обрізається, йдучи за межі компоненту, або розсовує ці межі. В обох випадках текст залишається обмежений розміром вікна. У таких випадках як правило використовують смуги прокрутки.

Додати для компонента смуги прокрутки доволі легко. Для цього служить компонент `JscrollPane` – панель прокрутки. Вона створюється поверх компонента, який має прокручуватися. Наприклад, аби текстова область `textArea` з попереднього прикладу мала полоси прокрутки, необхідно замінити рядок:

```
getContentPane().add(textArea);
```

на наступний:

```
getContentPane().add(new JScrollPane(textArea));
```

Тут створюється панель з смугами прокрутки і в неї поміщається об'єкт `textArea`, а сама панель додається в панель вмісту вікна. Тепер текст вільно прокручується. В разі ж застосування менеджера `FlowLayout` або `BoxLayout` компонент `JTextArea` не буде підлаштовуватися під свій вміст, а матиме бажаний розмір, відповідний параметрам конструктора, і відображати смуги прокрутки за необхідності.

`JScrollPane` дозволяє задавати правила відображення смуг прокрутки за допомогою двох властивостей `horizontalScrollBarPolicy` і `verticalScrollBarPolicy`. Вони мають наступні можливі значення:

- HORIZONTAL_SCROLLBAR_ALWAYS, VERTICAL_SCROLLBAR_ALWAYS – завжди відображати смуги прокрутки, незалежно від розміру вмісту;
- HORIZONTAL_SCROLLBAR_AS_NEEDED, VERTICAL_SCROLLBAR_AS_NEEDED – відображати смуги прокрутки, тільки коли вміст не вміщується всередині панелі;
- HORIZONTAL_SCROLLBAR_NEVER, VERTICAL_SCROLLBAR_NEVER – не відображати смуги прокрутки.

Всі ці константи визначені в інтерфейсі `ScrollPaneConstants`.

Панель інструментів JToolBar

Значна більшість програмних продуктів надають зручні інструментальні панелі, розташовані уздовж меж вікна програми і містять кнопки, списки, що випадають та інші елементи управління, які зазвичай відповідають командам меню. У Swing для інструментальних панелей розроблений багатофункціональний компонент `JToolBar`.

Для прикладу створимо вікно з менеджером розташування `BorderLayout` та розмістимо по центру область для введення тексту `JTextArea`, а до верхнього краю прикріпимо інструментальну панель з трьома кнопками і одним роздільником (рис. 6.14):

```
SimpleWindow() {
    super("Приклад використання JToolBar");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JTextArea textArea = new JTextArea(5, 20);
    getContentPane().add(textArea);
    JToolBar toolBar = new JToolBar("Панель інструментів");
    toolBar.add(new JButton("Кнопка 1"));
    toolBar.add(new JButton("Кнопка 2"));
    toolBar.addSeparator();
    toolBar.add(new JButton("Кнопка 3"));
    getContentPane().add(toolBar, BorderLayout.NORTH);
    pack();
}
```

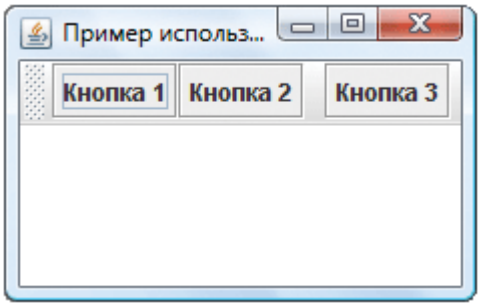

Рис. 6.14. Демонстрація панелі інструментів

Запустимо цей приклад і поекспериментуємо з інструментальною панеллю. Наприклад, спробуємо від'єднати її від верхнього краю вікна і прикріпити до будь-якого іншого краю. Потім від'єднаємо її від меж вікна так, щоб панель стала самостійним вікном. При цьому панель завжди відображається над батьківським вікном, навіть якщо воно, а не панель, є активним. Якщо закрити самостійну панель кнопкою з хрестиком, вона повернеться в своє вікно, тобто в те місце, де вона була закріплена останній раз.

Цей простий приклад продемонстрував базові можливості інструментальної панелі. Тепер наведемо деякі корисні методи `JToolBar`:

Конструктор `JToolBar(String title)` створює панель із заданим заголовком. За замовчанням створюється горизонтальна панель, яка призначена для прикріплення до верхньої або нижньої межі батьківського контейнера та має розміщення BorderLayout. Для створення вертикальної панелі використовується конструктор `JToolBar(String title, int orientation)`, де параметр orientation задається константою SwingConstants.VERTICAL. Інші конструктори JToolBar() і `JToolBar (int orientation)` створюють панель без заголовка.

`setFloatable(boolean floatable)` – дозволяє або забороняє користувачу відкріплювати панель від місця її початкового розташування. Відповідно метод `isFloatable()` повертає true, якщо відкріплювати панель дозволено.

`add(Component component)` – додає на інструментальну панель новий елемент управління. Взаємопов'язані групи елементів управління прийнято розділяти за допомогою лінії-роздільника: `addSeparator()`.

Випадний список JComboBox

Такий список містить кілька варіантів, з яких користувач може вибрати один і тільки один, або ввести свій власний, якщо випадний список це дозволяє.

Випадний список можна створити конструктором за замовчуванням JComboBox(), після чого додавати в нього елементи методом addItem(Object item), що додає новий елемент в кінець списку, або методом insertItemAt(Object item, int index), який вставляє елемент в задану позицію. Зазвичай найпростіше в конструкторі вказати відразу всі елементи списку. Таких конструкторів два: JComboBox(Object[] elements) і JComboBox(Vector elements). Ці конструктори працюють однаково, використовуючи різне подання елементів.

Найчастіше у випадний список додають елементи-рядки, проте, як це випливає з сигнатур описаних вище методів, він *може містити будь-які об'єкти*. Будь-який об'єкт перетворюється в рядок методом toString(), і саме цей рядок представлятиме його у випадному списку.

Метод getItemAt(int index) дозволяє звернутися до елементу списку за номером.

Метод removeAllItems() видаляє з JComboBox всі елементи, а метод removeItem(Object item) видаляє лише один вказаний елемент.

Метод getSelectedIndex() дозволяє отримати індекс обраного користувачем елемента (елементи нумеруються, починаючи з нуля), а метод getSelectedItem() повертає сам обраний об'єкт. Змінити вибір можна і програмно, скориставшись методом setSelectedIndex(int index) або setSelectedItem(Object item).

Для того щоб користувач міг ввести свій варіант, який відсутній в списку, потрібно встановити властивість editable: setEditable(true).

Тепер розглянемо приклад, в якому створюється випадний список з трьох елементів і вибирається другий елемент (рис. 6.15). Рядок, що представляє третій елемент, містить маркування HTML.

```
SimpleWindow() {
    super("Приклад використання JComboBox");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    String[] elements = new String[]
        {"Вася", "Петя", "<html><font size = +1 color = yellow>Іван</font>"};
    JComboBox combo = new JComboBox(elements);
    combo.setSelectedIndex(1);
    JPanel panel = new JPanel();
    panel.add(combo);
```

```
    setContentPane(panel);
    setSize(200, 200);
}
```

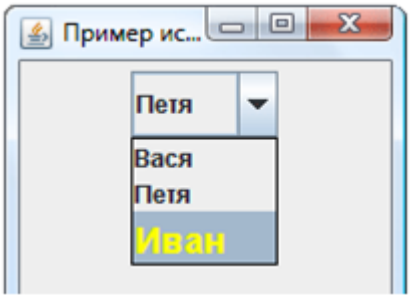

Рис. 6.15. Демонстрація випадного списку

Повзунок JSlider

Повзунок дозволяє користувачеві вибрати певну кількість з діапазону доступних значень, наочно представивши цей діапазон (рис. 6.16). Проте повзунок має один недолік – він займає досить багато місця.

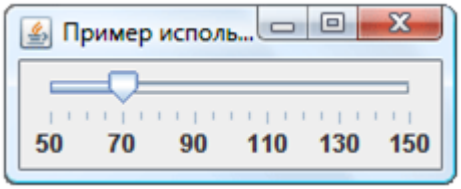

Рис. 6.16. Демонстрація повзунка

Основний конструктор повзунка: JSlider(int orientation, int min, int max, int value), де перший параметр – орієнтація повзунка (HORIZONTAL або VERTICAL), інші параметри вказують відповідно мінімальне, максимальне і поточне значення. Змінити їх дозволяють методи setOrientation(int), setMinimum(int min), setMaximum(int max), setValue(int value), а отримати поточні – відповідні їм методи get. Найчастіше використовується метод getValue() – аби визначити, яке значення вибрав користувач.

Шкала повзунка може мати ділення. Метод setMajorTickSpacing(int spacing) задає ціну великої поділки шкали, а метод setMinorTickSpacing(int spacing) – відповідно ціну малої поділки. В свою чергу, метод setPaintTicks(boolean paint) включає або відключає виведення цих поділок. Метод setSnapToTicks(boolean snap) включає або відключає «прилипання» повзунка до поділок: якщо викликати цей метод з параметром true, користувач зможе вибрати за допомогою повзунка тільки такі значення, що відповідають поділкам. Нарешті, метод setPaintLabels(boolean paint) включає або відключає виведення написів під великими поділками.

Приклад використання перерахованих методів:

```
SimpleWindow() {
    super("Приклад використання JSlider");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JSlider slider = new JSlider(JSlider.HORIZONTAL, 50, 150, 70);
    slider.setMajorTickSpacing(20);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);
    slider.setSnapToTicks(true);
    JPanel panel = new JPanel();
    panel.add(slider);
    setContentPane(panel);
    pack();
}
```

Панель із вкладками JTabbedPane

Створити панель з вкладками можна простим конструктором, в якому визначається тільки місце розташування ярликів (LEFT, RIGHT, TOP або BOTTOM). Проте іноді буває корисний конструктор

```
JTabbedPane(int orientation, int layout);,
```

де другий параметр приймає значення, що відповідають константам:

- SCROLL_TAB_LAYOUT (якщо всі ярлики не вміщуються, то появляється смуга прокрутки);
- WRAP_TAB_LAYOUT (ярлики можуть розташовуватись в кількох рядах).

Після цього можна додавати вкладки методом addTab(), що має кілька варіантів. Зокрема, метод addTab(String title, Component tab) додає закладку із зазначенням тексту ярлика, а метод addTab(String title, Icon icon, Component tab) дозволяє задати також і значок до ярлика. В якості вкладки зазвичай служить панель з розміщеними на ній елементами управління.

Тепер створимо панель з десятьма вкладками, на кожній з яких помістимо кнопку (рис. 6.17). Всі ці вкладки створимо в циклі.

```
SimpleWindow() {
    super("Приклад використання JTabbedPane");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JTabbedPane tabbedPane = new JTabbedPane(JTabbedPane.TOP,
                                             JTabbedPane.WRAP_TAB_LAYOUT);
    for (int i = 1; i <= 10; i++) {
        JPanel panel = new JPanel();
        panel.add(new JButton("Кнопка № " + i));
        tabbedPane.add("Панель " + i, panel);
    }
    getContentPane().add(tabbedPane);
    setSize(300, 200);
}
```

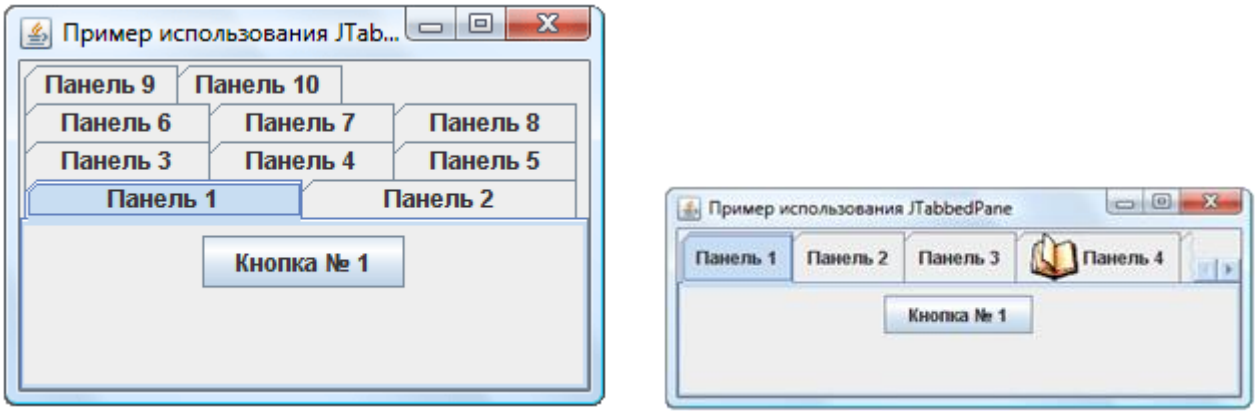

Рис. 6.17. Панель з різними варіантами розміщення вкладок

Список JList

Список JList – це один із важких компонентів, для ефективної роботи з якими необхідно розуміння концепції «Модель-Вид». Компоненти JTree (дерево) і JTable (таблиця) ще складніші, і їх необхідно розглядати окремо. Що стосується списку, то частина його можливостей може бути використана без заглиблення в деталі патерну «Модель-Вид».

Список містить групу елементів, аналогічно випадному списку JComboBox, але, по-перше, на екрані видно одночасно кілька елементів списку і, по-друге, користувач може вибрати в списку не один елемент, а кілька (за умови, якщо встановлений відповідний режим виділення).

Створити список можна на основі масиву Object[] або вектору Vector елементів. Метод setVisibleRowCount(int count) встановлює кількість видимих елементів списку. Інші пункти будуть виходити за його межі (або прокручуватися, якщо помістити список в JScrollPanel).

JList може зберігати не тільки рядки, але будь-які об'єкти. Для відображення цих об'єктів використовується метод toString().

За замовчуванням користувач може вибрати в списку будь-яке число елементів, утримуючи клавішу Ctrl. Режим виділення можна змінити методом setSelectionMode(int mode), де параметр задається однією з констант класу ListSelectionModel:

- SINGLE_SELECTION – можна виділити тільки один елемент,
- SINGLE_INTERVAL_SELECTION – можна виділити кілька елементів, розташованих підряд,
- MULTIPLE_INTERVAL_SELECTION – можна виділити довільну кількість елементів, як суміжних так і не суміжних.

Один виділений елемент списку можна отримати методом getSelectedValue(). Якщо виділено кілька елементів, тоді метод поверне перший з них. Метод getSelectedValues() повертає всі виділені елементи у вигляді масиву Object[]. Аналогічно працюють методи getSelectedIndex() і getSelectedIndices(), але повертають вони не самі виділені елементи, а їх індекси. До всіх цих методів є також парні методи set, так що змінювати виділення елементів списку можна і програмно.

Наступний приклад ілюструє деякі з можливостей JList (рис. 6.18):

```
SimpleWindow() {
    super("Приклад з JList");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Object[] elements = new Object[]{ "Ковбаса",
              "<html><font color = red>Масло", "Згущене молоко"};
    JList list = new JList(elements);
    list.setVisibleRowCount(5);
    list.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
    list.setSelectedIndices(new int[]{1, 2});
    getContentPane().setLayout(new FlowLayout());
    getContentPane().add(new JScrollPane(list));
    setSize(200, 150);
}
```

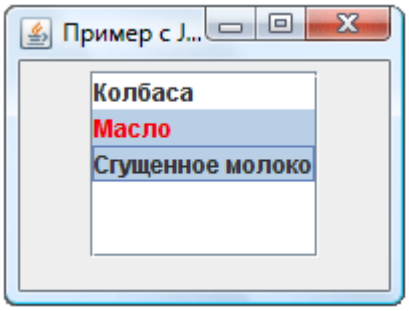

Рис. 6.18. Демонстрація списку

Для того, щоб ефективно додавати і видаляти елементи зі списку, керувати їх відображенням, додавати і видаляти виділені елементи поодинці, необхідно познайомитися з *моделлю даних списку*, що виходить за рамки цієї лекції.

Властивості компонентів

Всі властивості компонентів, що доступні в GUI-редакторі через панель властивостей, можна отримувати і змінювати програмно за допомогою методів set і get. Основні властивості, застосовні до більшості компонентів, такі:

- text – текстовий надпис;

- background – колір заповнення фону;
- foreground – колір тексту;
- font – шрифт тексту, об’єкт класу Font;
- visible – видимість;
- enabled – активність.

Створимо, наприклад, кнопку з жирним червоним написом. Означене можна зробити або за допомогою маркування HTML:

```
JButton myButton = new JButton(
     "<html><b><font color=\"red\">OK</font></b></html>");
```

або використовуючи властивості компонента:

```
JButton b = new JButton("OK”);
b.setForeground( Color.RED );
b.setFont(new Font("Tahoma", Font.BOLD, 14));
```

Лекція 10

Події компонентів Swing

Графічний інтерфейс користувача (GUI) визначає не тільки розташування необхідних елементів управління, але і поведінку додатку у відповідь на дії користувача. Значна частина дій у віконних програмах виконується у відповідь на вибір користувачем команд меню, натискання кнопок, а іноді навіть просто у відповідь на введення нового символу в текстовому полі.

Отже, при створенні додатку необхідно:

1. Виявити події, у відповідь на які потрібна реакція програми.
2. Написати код, який реалізує цю реакцію, тобто обробник подій.
3. Пов’язати обробник події з відповідною подією.

Перший пункт залежить від логіки роботи додатку і перебуває повністю під контролем програміста. ***Обробник*** події – це, як правило, звичайний метод в одному з класів додатку. Обробник будь-якої дії для компонента можна призначити як в GUI-редакторі, так і вручну в коді програми.

Патерн проектування «спостерігач»

У різних мовах програмування і візуальних бібліотеках використовуються різні способи прив'язки процедури (методу), яка повинна викликатися у відповідь на деяку подію, до самої цієї події. У бібліотеці Swing для прив'язки обробника події до компонента, що його викликає, використовується патерн проектування «спостерігач».

Патерни проектування – це стандартні прийоми об'єктно-орієнтованого програмування, що дозволяють оптимальним чином впоратися з нетривіальними, але поширеними задачами в програмуванні. Патерн проектування описує деяке стандартне рішення задачі у вигляді набору класів з чітко визначеними ролями і правилами взаємодії між ними.

Наприклад, патерн «*ітератор*» вирішує задачу перебору елементів довільної колекції, незалежно від її природи. У цьому патерні беруть участь один інтерфейс Iterable і два класи: перший представляє набір елементів і реалізує інтерфейс Iterable, а другий – власне ітератор – реалізує логіку проходу по цьому набору. Даний патерн дозволяє перебрати елементи колекції за допомогою двох простих методів, незалежно від того в якому порядку і за якими правилами відбувається перебір, а також від того, де фізично зберігаються перебирувані елементи.

Патерн проектування «***спостерігач***» застосовується, коли один об'єкт повинен сповіщати інші про зміни, що з ним відбулися. При цьому сам такий об'єкт називається спостережуваним, а об'єкти, які слід оповістити – ***спостерігачами***.

Подібна взаємодія буде можлива, якщо об'єкт-спостерігач має метод (або кілька методів) із заздалегідь визначеною сигнатурою (тобто ім'ям і параметрами), в якому запрограмована реакція на подію. Тоді в момент очікуваної події спостережуваний об'єкт може викликати відповідний метод свого спостерігача, «сповістивши» тим самим його про подію.

Для того, щоб спостережуваний об'єкт міг викликати метод спостерігача, він повинен мати посилання на об'єкт-спостерігач. Це посилання спостережуваного об'єкту потрібно попередньо передати, і це робиться за допомогою спеціального методу. Таким чином, спостережуваний об'єкт повинен зберігати список посилань на всіх своїх спостерігачів, а також метод для реєстрації нових спостерігачів.

Зауважте, що в даній схемі один спостерігач може бути зареєстрований у декількох спостережуваних об'єктів – тоді він буде однаково реагувати на зміни в кожному з них. З іншого боку, у одного спостережуваного об'єкта може бути кілька спостерігачів – тоді всі вони сповіщаються при виникненні події і виконується кілька незалежних методів-обробників. Це неабияк підвищує гнучкість програмування подій.

Механізм обробки подій Swing

У Swing спостережуваними об’єктами є всі без виключення компоненти інтерфейсу: саме вікно програми, кнопки, меню, поля введення і т. д. Вони можуть повідомити своїм спостерігачам

про певні події, як елементарні (наведення миші, натискання клавіші на клавіатурі), так і високого рівня (зміна тексту в текстовому полі, вибір нового елемента у випадному списку і т. д.).

Спостерігачами виступають об'єкти класів, що реалізують спеціальні інтерфейси Listener. Такі класи в термінології Swing називаються ***слухачами (listeners)***.

Інтерфейс MouseListener

Розглянемо технологію написання слухачів на прикладі слухачів подій миші.

Події миші – це один з найбільш затребуваних видів подій. Будь-який компонент інтерфейсу здатний повідомити про те, що на нього навели мишу чи натиснули по ньому і т. д. Про це будуть сповіщені всі зареєстровані слухачі подій миші.

Слухач подій миші повинен реалізувати інтерфейс **MouseListener**. У цьому інтерфейсі перераховані наступні методи:

- public void mouseClicked(MouseEvent event) – натискання мишею на об’єкті;
- public void mouseEntered(MouseEvent event) – курсор миші зайшов у область об’єкта;
- public void mouseExited(MouseEvent event) – курсор миші вийшов із області об’єкта;
- public void mousePressed(MouseEvent event) – кнопка миші натиснута, коли курсор знаходиться над об’єктом;
- public void mouseReleased(MouseEvent event) – кнопка миші відпущена, коли курсор знаходиться над об’єктом.

Наприклад, нам необхідно описати реакцію на подію натискання мишею кнопки «Увійти до системи». Спостережуваним об’єктом в цьому разі є кнопка стандартного класу JButton, а от клас «слухача» нам необхідно описати самим, реалізувавши інтерфейс MouseListener і помістивши код для обробки події в метод mouseClicked().

Опишемо клас слухача в межах класу вікна SimpleWindow. Обробник події перевірятиме, чи ввів користувач логін «Іван», а потім виведе повідомлення про успішний чи безуспішний вхід в систему:

```
class SimpleWindow extends JFrame {

     private JTextField loginField;
     private JTextField passwordField;

     class ButttonMouseListener implements MouseListener {

         public void mouseClicked(MouseEvent event) {
             if (loginField.getText().equals("Іван")) {
                 JOptionPane.showMessageDialog(null, "Вхід виконано");
             }
             else {
                 JOptionPane.showMessageDialog(null, "Вхід НЕ виконано!");
             }
         }

         public void mouseEntered(MouseEvent event) { }
         public void mouseExited(MouseEvent event) { }
         public void mousePressed(MouseEvent event) { }
         public void mouseReleased(MouseEvent event) { }
     }
     // ...
}
```

В нашій програмі слухач являється *вкладеним* класом всередині класу вікна SimpleWindow щоб він міг легко отримати доступ до змінних loginField і passwordField. Зверніть увагу, що нам довелося перевизначити всі п'ять абстрактних методів MouseListener, хоча тільки один із них реально використовується, а чотири мають порожню реалізацію. Якщо в класі реалізовані не всі абстрактні методи, то неможливо створити об'єкт цього класу.

Далі необхідно створити об'єкт-слухач і пов’язати його із кнопкою методом **addMouseListener(MouseListener listener)**. Це доречно зробити в конструкторі вікна SimpleWindow(), одразу після створення кнопки:

```
JButton ok = new JButton("OK");
ok.addMouseListener(new ButttonMouseListener());
```

Створення слухачів за допомогою анонімних класів

У попередньому прикладі для створення слухача для кнопки, ми створили новий вкладений клас, який виглядає досить громіздко. Якщо, як в нашому випадку, клас описується тільки для одноразового використання, то код можна скоротити, зробивши клас ***анонімним***.

Анонімний клас не має імені, а його оголошення поєднане зі створенням об'єкта цього класу. Всього в програмі може бути створений тільки один об'єкт анонімного класу.

У нашому прикладі необхідний тільки один об'єкт класу ButttonMouseListener. І в інших програмах дуже часто слухач пишеться для обробки подій одного єдиного об'єкта, а значить, використовується в програмі лише один раз: під час прив'язки до цього об'єкта.

Замінимо вкладений клас ButttonMouseListener анонімним. Для цього його старий опис потрібно видалити, а рядок:

```
ok.addMouseListener(new ButttonMouseListener());
```

замінити на наступний код:

```
ok.addMouseListener(
    new MouseListener() {
         public void mouseClicked(MouseEvent event) {
             if (loginField.getText().equals("Іван")) {
                 JOptionPane.showMessageDialog(null, "Вхід виконано");
             }
             else {
                 JOptionPane.showMessageDialog(null, " Вхід НЕ виконано!");
             }
         }
         public void mouseEntered(MouseEvent event) {}
         public void mouseExited(MouseEvent event) {}
         public void mousePressed(MouseEvent event) {}
         public void mouseReleased(MouseEvent event) {}
});
```

Новий варіант коротше, але в ньому настройка графічних компонентів (кнопки) і логіка обробки подій суміщені в одному методі, що неправильно з точки зору ООП: ці дії повинні бути розділені. Тому зловживання анонімними класами може зробити програму непрозорою і нечитабельною. На практиці їх застосовують в разі простих (в кілька рядків) обробників подій. Якщо ж в обробниках знаходиться багато коду, то їх необхідно виносити в окремі класи, які називаються *контролерами*.

Відзначимо, що можна реалізувати інтерфейс слухача в будь-якому класі, додавши до його опису `implements MouseListener` і визначивши в ньому необхідні методи. Зокрема, це можна зробити в самому класі вікна:

```
public class SimpleWindow implements MouseListener {...}
```

Але такий варіант використовується нечасто. Зазвичай в межах одного вікна кілька компонентів мають однотипні події (наприклад, кілька кнопок які можна натискати мишею), а значить необхідно мати кілька різних класів-слухачів, успадкованих від MouseListener.

Клас MouseAdapter

Додавання слухача в нашій програмі виглядає громіздко переважно через те, що крім корисного для нас методу mouseClicked() нам довелося визначати порожні реалізації всіх інших методів MouseListener. Цього можна уникнути.

Клас **MouseAdapter** реалізує інтерфейс MouseListener, визначаючи порожні реалізації для кожного з його методів. Можна успадкувати свій клас слухача від нього і перевизначити тільки ті методи, які нам потрібні.

```
ok.addMouseListener(new MouseAdapter() {
    public void mouseClicked (MouseEvent event){
        if (loginField.getText().equals("Іван")) {
              JOptionPane.showMessageDialog(null, "Вхід виконано");
        }
        else { JOptionPane.showMessageDialog(null, "Вхід НЕ виконано!"); }
    }
});
```

Подібні класи «адаптерів» є для всіх інтерфейсів слухачів Swing, які містять більше одного методу.

Загальна структура слухачів

Візуальні компоненти Swing підтримують цілий ряд різних видів слухачів. При цьому різні компоненти можуть мати різний набір застосовних до них слухачів.

Кожен слухач повинен реалізовувати інтерфейс XxxListener, де Xxx – тип слухача. Практично кожному з цих інтерфейсів (за винятком тих, в яких всього один метод) відповідає порожній клас-заглушка XxxAdapter. Кожен метод інтерфейсу слухача приймає один параметр типу XxxEvent, в якому зібрана вся інформація, що відноситься до події. Ми не будемо детально розглядати методи отримання цієї інформації. Як правилом, їх небагато і їх призначення легко зрозуміти, а перелік методів дасть контекстна підказка IDE.

Щоб прив'язати слухача до компоненту у всіх випадках використовується метод:

```
addXxxListener(XxxListener listener);
```

Наприклад, слухач подій, пов'язаних з коліщатком миші, повинен реалізувати інтерфейс MouseWheelListener, і є клас-заглушка MouseAdapter. В даному випадку один клас-заглушка відповідає відразу декільком інтерфейсам слухачів, пов'язаних з мишею. Методи інтерфейсу MouseWheelListener мають параметр типу MouseWheelEvent, а реєструється слухач методом addMouseWheelListener (MouseWheelListener listener). Ми не будемо повторювати цю інформацію для інших типів слухачів, вони працюють аналогічно. Замість цього ми розглянемо прийоми їх використання.

Універсальний слухач ActionListener

Багато елементів управління (але не всі) мають одну основну подію, яка обробляється значно частіше, ніж інші. Наприклад, для кнопки це натискання, а для списку – зміна обраного елемента. Для відстеження та обробки такої події використовують слухач **ActionListener**, який має один метод:

```
public void actionPerformed (ActionEvent event);
```

ActionListener – це найбільш часто використовуваний слухач. Зазвичай більшість компонентів форми не мають інших, більш спеціалізованих слухачів. Розглянемо кілька типових прикладів.

Приклад 1. Перепишемо обробник для кнопки «Ok» з попереднього прикладу за допомогою ActionListener:

```
ok.addActionListener(
    new ActionListener() {
         public void actionPerformed(ActionEvent event) {
              if (loginField.getText().equals("Іван")) {
                   JOptionPane.showMessageDialog(null, "Вхід виконано");
              } else {
                   JOptionPane.showMessageDialog(null, "Вхід НЕ виконано!");
              }
         }
});
```

Зверніть увагу, що цей обробник не еквівалентний MouseAdapter, оскільки кнопка натискається не тільки мишею, але також і за допомогою клавіші Enter.

Приклад 2. Для текстового поля додамо перевірку введеного значення на допустимість. Нехай необхідно вводити дійсне число, а при неправильному введенні – виділити введений текст червоним кольором:

```
     txt = new JTextField();
     txt.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent arg0) {
               if( !txt.getText().matches("\\d*\\.?\\d+") ) {
                    txt.setForeground( new Color( 255, 100, 100 ) );
               } else {
                    textField.setForeground( Color.BLACK );
               }
          }
     });
     form.getContentPane().add(textField);
}
```

Зверніть увагу, що для текстового поля ActionListener спрацює тільки в разі завершення введення клавішею Enter. Для ситуації, коли користувач перейшов на наступне поле за допомогою миші або клавіші Tab, потрібно обробляти інші події, які ми будемо обговорювати далі.

Приклад 3. Створення вікна з кнопками OK і Cancel

```
import java.awt.*;
import javax.Swing.*;

// Клас додатку
public class App {
     private JFrame form;
     public static void main(String[] args) {
          // Створення вікна
          App app = new App();
          app.form.setVisible(true);
}
     // Ініціалізація вікна
     public App() {
          form = new JFrame();                    // Створити головне вікно
          form.setBounds(100, 100, 450, 300);   // Виставити позицію і розмір
          form.setTitle("MyWnd");
          form.setResizable(false);
          frmLogin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          // Отримати панель вмісту і виміряти її параметри
          Container pane = form.getContentPane();
          pane.setBackground(Color.WHITE);
          pane.setFont(new Font("Tahoma", Font.BOLD, 14));

          // Виставити розташування елементів панелі
          pane.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 10));
```

```
        // Створити кнопки
        JButton btnOK = new JButton("OK");
        // Добавити обробник
        btnOK.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent arg0) {
                  JOptionPane.showMessageDialog(null, "Message");
             }
        });
        pane.add(btnOK);

        JButton btnCancel = new JButton("Cancel");
        btnCancel.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent arg0) {
               System.exit(0);
             }
        });
        pane.add(btnCancel);
    }
}
```

Слухач фокуса FocusListener

Слухач FocusListener активується, коли об'єкт отримує фокус введення (тобто стає активним) або втрачає його. Концепція фокуса дуже важлива для віконних додатків. У кожен момент часу в вікні тільки один елемент управління має фокус введення – він є активним і саме він отримує інформацію про натиснуті на клавіатурі клавіші, про прокрутку коліщатка миші і т.д., і може реагувати на ці події. Користувач активує елемент управління натисканням миші або перемикаючись між елементами за допомогою клавіші Tab.

Інтерфейс FocusListener має два методи:

- public void focusGained(FocusEvent event) – викликається, коли елемент отримує фокус введення;
- public void focusLost(FocusEvent event) – викликається, коли елемент втрачає фокус введення (тобто фокус переходить до іншого елемента, або до іншого вікна).

Слухач клавіатури KeyListener

Слухач KeyListener спрацьовує для елемента управління, що має фокус введення, в момент натискання будь-якої клавіші на клавіатурі. Цей інтерфейс призначений для реагування на події низького рівня, пов'язані з натисканням окремих клавіш і їх комбінацій. Він погано підходить, наприклад, для відстеження введення окремих символів в текстове поле – для цього використовується слухач DocumentListener. Для інших типових задач, пов'язаних з клавіатурою, Swing має спеціалізовані засоби, такі як Key Bindings, Mnemonics і Accelerators, що сильно звужує область використання KeyListener. В основному він застосовується у випадках, коли необхідно управляти нестандартними (користувацькими) компонентами за допомогою клавіш.

Інтерфейс KeyListener має такі методи:

- public void mouseKeyTyped (KeyEvent event) – викликається, коли з клавіатури введено символ;
- public void mouseKeyPressed (KeyEvent event) – викликається в момент натискання клавіші (в т.ч. керуючої, не обов'язково символьної);
- public void mouseKeyReleased (KeyEvent event) – викликається в момент відпускання клавіші.

Всі ці методи отримують параметр event, який містить повну інформацію про подію. Зокрема, вираз event.getKeyChar() повертає символ (char), пов'язаний з натиснутою клавішею. Якщо з натиснутою клавішею не пов'язаний ніякий символ, повертається константа CHAR_UNDEFINED. Вираз event.getKeyCode() видає код клавіші (ціле число типу int). Цей код унікальний для всіх клавіш, включаючи керуючі, а список всіх можливих кодів визначено у вигляді констант в класі KeyEvent: VK_F1, VK_SHIFT, VK_D, VK_MINUS і т. д. Додатково

методи isAltDown (), isControlDown (), isShiftDown () дозволяють дізнатися, чи були одночасно натиснуті клавіші-модифікатори Alt, Ctrl або Shift.

Слухач зміни стану ChangeListener

ChangeListener реагує на зміну стану об'єкта. Кожен елемент управління по своєму визначає поняття «зміна стану». Наприклад, для панелі з вкладками JTabbedPane це перехід на іншу вкладку, для повзунка JSlider – зміна його положення. Для кнопки JButton зміною стану вважається кожне її натискання. Отже, хоча подія зміни стану є досить загальною, необхідно уточнювати її специфіку для кожного конкретного компонента.

В інтерфейсі визначено всього один метод:

```
public void stateChanged (ChangeEvent event);
```

Слухач подій вікна WindowListener

Слухач WindowListener може бути пов'язаний тільки з вікном (тобто контейнером верхнього рівня) і спрацьовує при різних подіях, що сталися з вікном:

- public void windowOpened(WindowEvent event) – вікно відкрилося;
- public void windowClosing(WindowEvent event) – спроба закриття вікна (наприклад, користувач натиснув на хрестик). Тут слово «спроба» означає, що метод викликається до того, як вікно буде закрито і може відмінити закриття (наприклад, вивести попередження «Ви впевнені?» і скасувати закриття вікна, якщо користувач вибере «Ні»);
- public void windowClosed(WindowEvent event) – вікно вже закрилося;
- public void windowIconified(WindowEvent event) – вікно згорнуто в панель задач;
- public void windowDeiconified(WindowEvent event) – вікно розгорнуто на екран;
- public void windowActivated(WindowEvent event) – вікно стало активним (тобто отримало фокус введення);
- public void windowDeactivated(WindowEvent event) – вікно стало неактивним.

Слухач подій компонента ComponentListener

Слухач ComponentListener оповіщається, коли спостережуваний візуальний компонент змінює своє положення, розміри або видимість. Ці зміни можуть відбуватися з різних причин, в тому числі в результаті виклику методу setLocation() або роботи менеджера розміщення. В інтерфейсі ComponentListener чотири методи:

- public void componentMoved(ComponentEvent event) – викликається, коли спостережуваний компонент переміщається;
- public void componentResized(ComponentEvent event) – викликається, коли змінюються розміри спостережуваного компонента;
- public void componentHidden(ComponentEvent event) – викликається, коли компонент стає невидимим;
- public void componentShown(ComponentEvent event) – викликається, коли компонент стає видимим.

Слухач вибору елемента ItemListener

Слухач ItemListener реагує на зміну стану спостережуваного компонента. Він застосовується не для до всіх компонентів і кожен клас компонентів по-своєму визначає поняття «зміна стану». Наприклад, для списку JComboBox цей слухач реагує на зміну обраного елемента. Для JCheckBox і JRadioButton він оповіщається при установці або знятті прапорця, для JToggleButton – при зміні стану кнопки. Слухач має один метод:

```
public void itemStateChanged(ItemEvent event).
```

Приклад – простий текстовий редактор

Створимо простий додаток для редагування довільних текстових файлів. Цей додаток дозволить:

- вводити ім'я файлу безпосередньо в текстове поле (JTextField);
- вибирати файл за допомогою стандартного діалогу відкриття файлу (JFileChooser) за кнопкою "Open";
- редагувати вміст файлу в багаторядковому текстовому полі (JTextArea);
- зберігати зміни у файлі за кнопкою "OK";
- вийти з програми за кнопкою "Cancel", відображаючи попередження перед виходом.

Вікно додатка має вигляд, показаний на рис. 6.19:

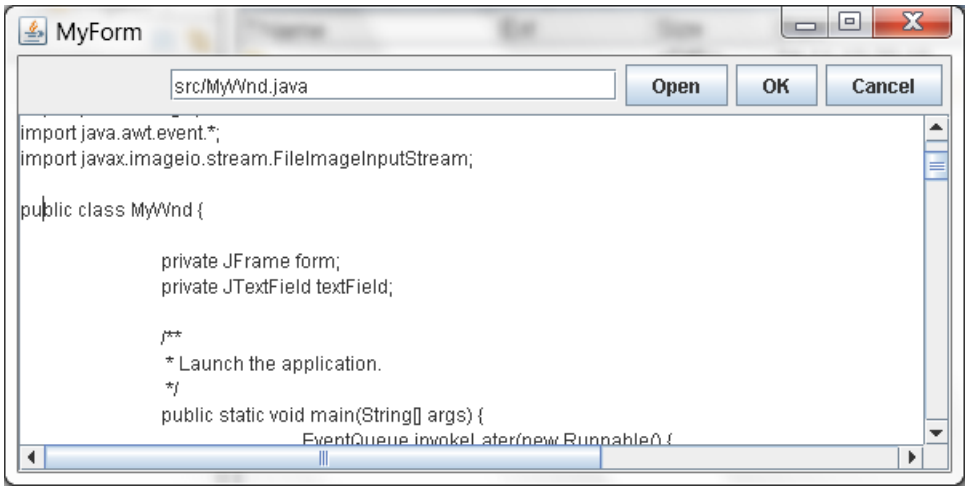

Рис. 6.19. Вікно додатку редактора

Код додатку:

```
import java.io.*;
import java.util.Scanner;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MyApp {                    // Клас додатку

     private JFrame form;               // Головне вікно
     private JTextField txtName;        // Поле для імені файлу
     private JTextArea txtFile;         // Вміст файлу

     public static void main(String[] args) {
          MyApp app = new MyApp();      // Створити об'єкт-додаток
          app.form.setVisible(true);    // Показати його вікно
     }

     public MyApp() {
          // Створюємо вікно
          form = new JFrame();
          form.setTitle("MyForm");
          form.setBounds(100, 100, 600, 300);
          form.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          // Розташування для елементів головної панелі
          Container pane = form.getContentPane();
          pane.setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));

          // Створюємо верхню панель
          JPanel panel = new JPanel();
```

```
panel.setLayout(new FlowLayout( FlowLayout.RIGHT ));
panel.setMaximumSize(new Dimension(600,30));
pane.add(panel);

// Добавляємо елементи верхньої панелі
txtName = new JTextField();
txtName.setColumns(25);
panel.add(txtName);
// Перевірка правильності введення
txtName.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent arg0) {
          if( !txtName.getText().matches(".+\\.html?") )
               txtName.setForeground( new Color( 255, 100, 100 ) );
          else txtName.setForeground( Color.BLACK );
     }
});
JButton btnOpen = new JButton("Open");
panel.add(btnOpen);
JButton btnOK = new JButton("OK");
panel.add(btnOK);
JButton btnCancel = new JButton("Cancel");
panel.add(btnCancel);

// Створюємо поле з прокруткою для виведення тексту
txtFile = new JTextArea("No file selected");
JScrollPane scroll = new JScrollPane (txtFile);
form.getContentPane().add(scroll);

// Додаємо обробники:
// Кнопка Open
btnOpen.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent arg0) {
          // Створюємо діалог вибору файла
          JFileChooser fc = new JFileChooser();
          // Якщо файл вибраний
          if( JFileChooser.APPROVE_OPTION ==
                    fc.showOpenDialog(form) )
               // То його ім'я записуємо  в txtName
               txtName.setText( fc.getSelectedFile().getAbsolutePath() );
     }
});
// Кнопка OK
btnOK.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent arg0) {
          try {
               // Створюємо об'єкт  File
               File f = new File(txtName.getText());
               // Перевіряємо чи файл існує
               if( !f.exists() ) txtFile.setText("No such file");
               else {
                    // Відкриваємо файл для читання з допомогою Scanner
                    Scanner sc  = new Scanner(f); String s ="";
                    // Тоді записуємо весь його зміст в поле txtFile
                    while( sc.hasNextLine() ) s = s + sc.nextLine() + "\n";
                    txtFile.setText(s);
               }
          } catch (Exception e) {
               // Виводимо повідомлення про помилку
               JOptionPane.showMessageDialog(null, e.getMessage());
          }
     }
});
// Кнопка Cancel
btnCancel.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent arg0) {
```

```
                    // Створюємо діалог підтвердження з кнопками OK і CANCEL
                    if( JOptionPane.OK_OPTION ==
                         JOptionPane.showConfirmDialog( form,
                                   "Exit without saving?", "Exit",
                                   JOptionPane.OK_CANCEL_OPTION ) )
                         // Якщо натиснуто OK, то завершуємо програму
                         System.exit(0);
               }
          });
     }
}
```

Робота з меню

Меню є невід'ємною частиною сучасних віконних додатків і представляє собою зручно згрупований набір команд. Меню поділяється на два типи: головне і контекстне. *Головне меню* розташовується уздовж верхньої межі вікна і містить команди, що відносяться до додатку в цілому (точніше, всі команди, доступні в даному вікні додатку). *Контекстне меню* викликається натисненням правої кнопки миші на конкретному графічному елементі і містить команди, які можуть бути застосовані саме до цього елемента.

Головне меню

В Swing *головне меню* вікна представлено класом ***JMenuBar***. Меню являє собою панель з менеджером розташування BoxLayout (по горизонталі), в яку можна додавати як пункти меню, так і будь-які інші графічні компоненти: випадні списки, кнопки, навіть панелі з закладками. Однак для зручності користувача рекомендується використовувати тільки «традиційні» пункти меню.

Головне меню приєднується до вікна методом `setJMenuBar(JMenuBar menuBar)`.

Елементами головного меню є *звичайні меню*, які являють собою випадні прямокутні блоки команд і є об'єктами класу ***JMenu***. Конструктор JMenu(String title) приймає один параметр: назву меню, яка буде відображатися в рядку головного меню.

Кожне меню у свою чергу складається з *пунктів меню*, представлених класом ***JMenuItem***. За логікою роботи пункти меню аналогічні кнопці JButton, тобто при натисканні на кожному пункті виконується певна дія.

Елемент меню створюється порожнім конструктором JMenuItem() або одним з конструкторів, в які передається текст і/або значок елемента меню:

- JMenuItem(String text),
- JMenuItem(Icon icon),
- JMenuItem(String text, Icon icon).

В будь-який момент текст і значок можна змінити методами setText(String text) і setIcon(Icon icon) відповідно.

Запис додається до меню JMenu методом add(JMenuItem item). Щоб відокремити групи взаємопов'язаних елементів меню можна додати між ними роздільник методом addSeparator().

Також в меню можна додати вкладене меню. Вкладені меню дуже широко використовуються в сучасних додатках, але зазвичай глибина вкладеності більше трьох рівнів незручна для користувачів.

Для прикладу додамо у вікно програми головне меню, що складається з двох підменю: «Файл» і «Правка», а в меню «Правка» помістимо випадне підменю (рис. 6.20). Скористаємося знаннями про менеджер розташування головного меню щоб додати з правого краю значок (який не є пунктом меню).

```
SimpleWindow() {
    super("Вікно з меню");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JMenuBar menuBar = new JMenuBar();
    JMenu fileMenu = new JMenu("Файл");
    fileMenu.add(new JMenuItem("Новий"));
    fileMenu.add(new JMenuItem("Відкрити", new ImageIcon("1.png")));
```

```
    fileMenu.add(new JMenuItem("Зберегти"));
    fileMenu.addSeparator();
    fileMenu.add(new JMenuItem("Вийти"));
    JMenu editMenu = new JMenu("Правка");
    editMenu.add(new JMenuItem("Копіювати"));
    JMenu pasteMenu = new JMenu("Вставити");
    pasteMenu.add(new JMenuItem("Із буфера"));
    pasteMenu.add(new JMenuItem("Із файлу"));
    editMenu.add(pasteMenu);
    menuBar.add(fileMenu);
    menuBar.add(editMenu);
    menuBar.add(Box.createHorizontalGlue());
    menuBar.add(new JLabel(new ImageIcon("2.png")));
    setJMenuBar(menuBar);
    setSize(250, 150);
}
```

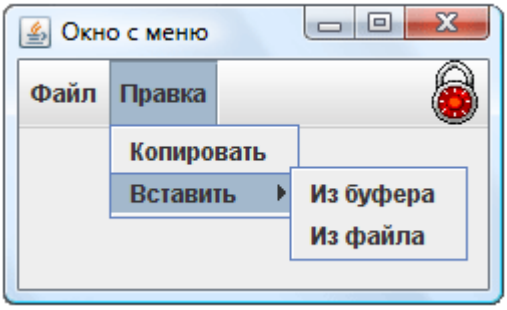

Рис. 6.20. Вікно з головним меню

Обробка команд меню

Кожен елемент меню по суті являє собою кнопку, тому реагує на такі ж події і дозволяє приєднувати до себе таких же слухачів, що і кнопка. Наприклад, щоб при виборі в меню елемента «Вийти» програма припиняла свою роботу, слід замінити в прикладі рядок

```
fileMenu.add(new JMenuItem("Вийти"));
```

на наступні рядки:

```
JMenuItem exit = new JMenuItem("Вийти");
exit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) { System.exit(0); }
});
fileMenu.add(exit);
```

Зауважимо, що метод `System.exit(0)` закінчує роботу додатку.

Як створити контекстне меню

Контекстне (інакше спливаюче) меню реалізовано в класі **JPopupMenu**, дуже схожому на клас JMenu. Його відмінністю є метод

```
show(Component comp, int x, int y),
```

який відображає меню в точці з заданими координатами щодо меж батьківського компонента.

Контекстне меню, як правило, відображається при натисканні правою кнопкою миші над батьківським компонентом. Отже, щоб відобразити меню, потрібно додати до цього компоненту слухача миші.

Розглянемо приклад, в якому до вже створеного вікна додається напис із контекстним меню (рис. 6.21). В конструкторі вікна необхідно додати наступний код:

```
label=new JLabel("КНИЖКА", new ImageIcon("book.png"), JLabel.RIGHT);
JPanel panel = new JPanel();
panel.add(label);
popup=new JPopupMenu();

popup.add(new JMenuItem("Прочитати"));
popup.add(new JMenuItem("Спалити") );
label.addMouseListener(new MouseAdapter() {
    public void mouseClicked (MouseEvent event){
        if (SwingUtilities.isRightMouseButton(event)) {
            popup.show(label, event.getX(), event.getY());
        }
    }
});
setContentPane(panel);
```

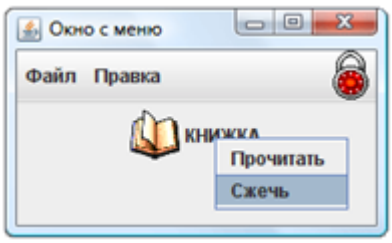

Рис. 6.21. Контекстне меню для напису

При цьому в класі вікна потрібно додати два поля:

```
JPopupMenu popup; JLabel label;
```

Це необхідно для того, щоб анонімний клас-слухач міг звернутися і до мітки і до контекстного меню.

Наведений приклад також ілюструє застосування методу `isRightMouseButton (MouseEvent event)` у слухачі подій миші. Цей метод визначений в класі допоміжних утиліт SwingUtilities і дозволяє за подією MouseEvent визначити, чи була натиснута саме права кнопка миші. При цьому методи event.getX() і event.getY() повертають координати курсора миші відносно спостережуваного компонента.

Клас AbstractAction та інтерфейс Action

Дуже часто одна і та ж команда дублюється в графічному інтерфейсі і може бути викликана кількома шляхами. Скажімо, команда «Копіювати» може бути присутня в головному меню вікна, в контекстному меню певного об'єкту, а також у вигляді кнопки на панелі інструментів. Це необхідно для зручності користування програмою. Проте, прив’язувати до всіх згаданих компонентів (кнопок і команд меню) однакових слухачів досить незручно. Більш того, всі ці команди однаково повинні реагувати на зміни стану програми: якщо копіювання з якихось причин стає неможливим, то всі елементи управління, що представляють дану команду, повинні стати неактивними.

Для спрощення роботи в таких ситуаціях передбачений інтерфейс Action. Цей інтерфейс збирає в одному місці все, що відноситься до деякої дії: обробник, показник активності, підказку, значок і т. д.

Інтерфейс Action успадкований від інтерфейсу ActionListener, тому його головним методом є `actionPerformed(ActionEvent event)`. Саме тут програмується необхідна дія. Крім того, за допомогою методу `putValue(String key, Object value)` можна задати додаткові властивості дій. Перший параметр – це строковий ідентифікатор властивості. Цей параметр може приймати одне із значень, описаних в константах інтерфейсу Action. Другий параметр – об'єкт, який представляє собою значення властивості.

Основні властивості інтерфейсу Action та відповідні їм константи:

- `NAME` – ім'я дії,
- `SMALL_ICON` – значок, що відповідає даній дії,
- `SHORT_DESCRIPTION` – короткий опис дії.

Зробити дію активною чи неактивною дозволяє метод `setEnabled(boolean enabled)`.

На основі об'єкта-дії можна створювати різні елементи управління, передаючи цю дію в якості єдиного параметра конструктора. Елементи меню і кнопки, зокрема, якраз і відносяться до таких елементів управління.

Недоліком інтерфейсу Action є занадто велика кількість допоміжних абстрактних методів, які досить складно програмувати (наприклад, методи setEnabled(), putValue()). Тому зазвичай замість нього використовується клас AbstractAction, який містить реалізації за замовчанням всіх методів інтерфейсу Action, крім actionPerformed(). Тоді нам залишається написати код тільки для одного метода.

Розглянемо приклад.

```
public class SimpleWindow extends JFrame {
    private ExitAction exitAction;

    SimpleWindow() {
        super("Вікно з меню");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        exitAction = new ExitAction();
        DeactivateAction deactivateAction = new DeactivateAction();
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("Файл");
        fileMenu.add(new JMenuItem("Новий"));
        fileMenu.addSeparator();
        fileMenu.add(deactivateAction);
        fileMenu.add(exitAction);
        menuBar.add(fileMenu);
        setJMenuBar(menuBar);
        JToolBar toolBar = new JToolBar("Панель інструментів");
        toolBar.add(exitAction);
        toolBar.add(deactivateAction);
        getContentPane().add(toolBar, BorderLayout.NORTH);
        JPanel panel = new JPanel();
        panel.add(new JButton(exitAction));
        panel.add(new JButton(deactivateAction));
        getContentPane().add(panel);
        setSize(250, 250);
    }

    class ExitAction extends AbstractAction {
        ExitAction() {
            putValue(Action.NAME, "Вийти");
            putValue(Action.SHORT_DESCRIPTION, "Завершення програми.");
            putValue(Action.SMALL_ICON, new ImageIcon("2.png"));
        }

        public void actionPerformed(ActionEvent event) {
            System.exit(0);
        }
    }

    class DeactivateAction extends AbstractAction {
        DeactivateAction() {
            putValue(Action.NAME, "Заборонити вихід");
            putValue(Action.SMALL_ICON, new ImageIcon("1.png"));
        }

        public void actionPerformed(ActionEvent event) {
            if (exitAction.isEnabled()) {
```

```
                exitAction.setEnabled(false);
                putValue(Action.NAME, "Дозволити вихід");
            } else {
                exitAction.setEnabled(true);
                putValue(Action.NAME, "Заборонити вихід");
            }
        }
    }
}
```

Тут в класі вікна описані два вкладених класи-дії. Перша дія – це вихід з програми. Об'єкт цього класу exitAction є полем в класі вікна і на його основі створюються пункт меню та дві кнопки. В результаті пункт меню і обидві кнопки будуть запускати одну і ту саму дію, мати однакові назви, значки та спливаючі підказки. Другий клас-дія активує і деактивує об'єкт exitAction, роблячи першу дію доступною або навпаки недоступною. При цьому також змінюється назва всіх кнопок, що запускають другу дію. Запустіть приклад і поспостерігайте за тим, як виглядають і поводяться всі кнопки і пункти меню в обох випадках: до вибору дії «Заборонити вихід» і після неї.

Стандартні діалогові вікна

Клас JOptionPane

Невід'ємною частиною більшості програм є невеликі діалогові вікна, які з'являються на екрані за потребою. Зазвичай вони виводять користувачеві повідомлення (наприклад, повідомлення про помилку), задають питання (наприклад, вікно підтвердження або скасування дії), або вимагають певну інформацію (наприклад, вікно відкриття файлу). Подібні діалогові вікна можна запрограмувати вручну на основі класу JDialog, який багато в чому схожий на знайомий нам JFrame. Однак з огляду на те, що ці вікна є типовими для багатьох додатків, Swing надає в розпорядження програміста кілька готових класів для роботи з ними.

Найчастіше використовується клас **JOptionPane**, що містить кілька статичних методів для створення стандартних діалогових вікон.

Метод `showMessageDialog()` виводить вікно-повідомлення, яке містить напис, значок і кнопку OK. Існує кілька різновидів цього методу з різними наборами параметрів. Найпростіший з них showMessageDialog(Component component, Object content) вимагає вказати компонент, над яким з'явиться діалогове вікно і вміст вікна. Найчастіше вмістом вікна є рядок повідомлення. Замість першого параметра можна передати null – тоді вікно з'являється по центру екрана. Розширений варіант showMessageDialog(Component component, Object content, String title, int type) дозволяє задати також заголовок вікна і тип повідомлення (від якого залежить іконка у вікні): повідомлення про помилку (ERROR_MESSAGE), попередження (WARNING_MESSAGE), інформація (INFORMATION_MESSAGE).

Діалогове вікно є ***модальним***. Це означає, що поки користувач не натисне в цьому вікні кнопку OK, додаток буде заблоковано – користувач не зможе працювати з іншими вікнами.

Наведемо приклад створення вікна-попередження (рис. 6.22).

```
public class SimpleWindow extends JFrame {
    private JButton button;

    SimpleWindow() {
       super("Попереджувальний діалог");
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       button = new JButton("Інформація");
       button.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent event) {
               JOptionPane.showMessageDialog(button,
                   "Не треба було натискати на цю кнопку", "Інформація",
                   JOptionPane.WARNING_MESSAGE);
           }
       });
```

```
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(button);
        setSize(200, 150);
    }
}
```

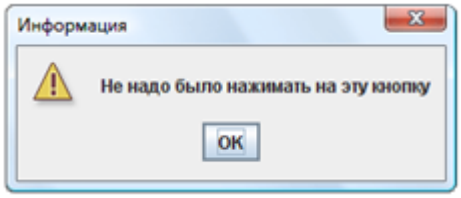

Рис. 6.22. Вікно попередження

Часто від користувача вимагається певна відповідь, в найпростішому випадку – відповідь «так» чи «ні» – тоді використовується діалогове вікно “Confirm Dialog” (вікно підтвердження). Воно має кілька кнопок, одну з яких користувач повинен натиснути. Програма тоді повинна отримати інформацію про вибір користувача, на основі якої і будується подальша логіка її роботи.

Вікно підтвердження відображається методом `showConfirmDialog(Component component, Object content)`. Параметри цього методу є ідентичними за змістом параметрам showMessageDialog(), але тепер вікно має не одну кнопку, а три: «Yes», «No» і «Cancel». Метод повертає значення, вибране користувачем, яке можна порівняти з константами YES_OPTION, NO_OPTION і CANCEL_OPTION.

Покажемо логіку роботи з цим методом на прикладі (рис. 6.23)

```
public class SimpleWindow extends JFrame {
    JButton button;

    SimpleWindow() {
        super("Діалог підтвердження");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        button = new JButton("Вихід");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                if (JOptionPane.showConfirmDialog(button,
                        "Ви впевнені,що хочете вийти?") == JOptionPane.YES_OPTION) {
                    System.exit(0);
                }
            }
        });
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(button);
        setSize(200, 150);
    }
}
```

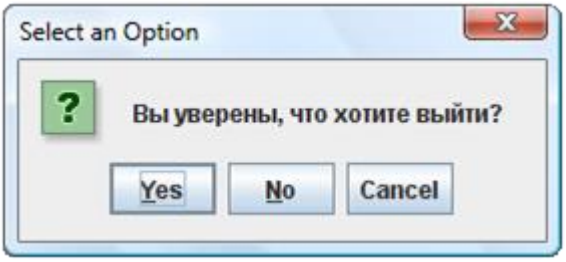

Рис. 6.23. Вікно підтвердження

Метод showConfirmDialog() має ще ще чотири різновиди з різним набором параметрів, які дозволяють змінювати заголовок і значок вікна, а також набір кнопок.

Клас JFileChooser

Swing має готове вікно для вибору файлу, яке корисне, наприклад, для програмування пункту меню «Файл» ® «Відкрити». Об'єкт класу **JFileChooser** створюється простим конструктором без параметрів, після чого може виводитися на екран методом `showOpenDialog()`. Цей метод повертає результат, яким завершився вибір файлу і який має наступні можливі значення (описані як константи в класі JFileChooser):

`APPROVE_OPTION` – вибір файлу пройшов успішно; тепер можна методом getSelectedFile() отримати обраний файл;

`CANCEL_OPTION` – користувач скасував вибір файлу, натиснувши кнопку Cancel;

`ERROR_OPTION` – при виборі файлу сталася помилка або користувач закрив діалогове вікно хрестиком.

Метод `showSaveDialog()` відображає схоже вікно, призначене для збереження файла. Користувач може не обирати існуючий файл, а вибрати директорію для збереження файлу і вказати ім'я нового файлу. Метод повертає результат того ж типу, що й showOpenDialog(). Якщо вибір шляху для збереження пройшов успішно, виклик методу getSelectedFile() поверне об’єкт File, що містить повне ім’я файлу (зі шляхом), куди користувач бажає зберегти дані.

Слід мати на увазі, що сам клас JFileChooser не відкриває і не зберігає файлів. Він тільки повертає шлях до обраного користувачем файлу. А відкрити файл або зберегти його в заданому місці повинна вже сама програма.

Тема 7. Java та теорія графів

Алгоритм Прима

Даний алгоритм є дуже ефективним для знаходження мінімального кістякового дерева. В свою чергу, знання такого дерева дозволяє вирішити надзвичайно актуальні логістичні задачі. Але спочатку дамо визначення дерева, кістякового дерева та мінімального кістякового дерева.

Дерева – найпоширеніший і специфічний клас графів. Даному класу притаманні ряд особливих властивостей, що відрізняє їх від інших видів графів. Специфіка дерев ще і в тому, що вони найчастіше використовуються в програмуванні. Дерево – це зв'язний ациклічний граф, тобто граф, в якому немає циклів. На рис. 7.1 показано саме такий граф.

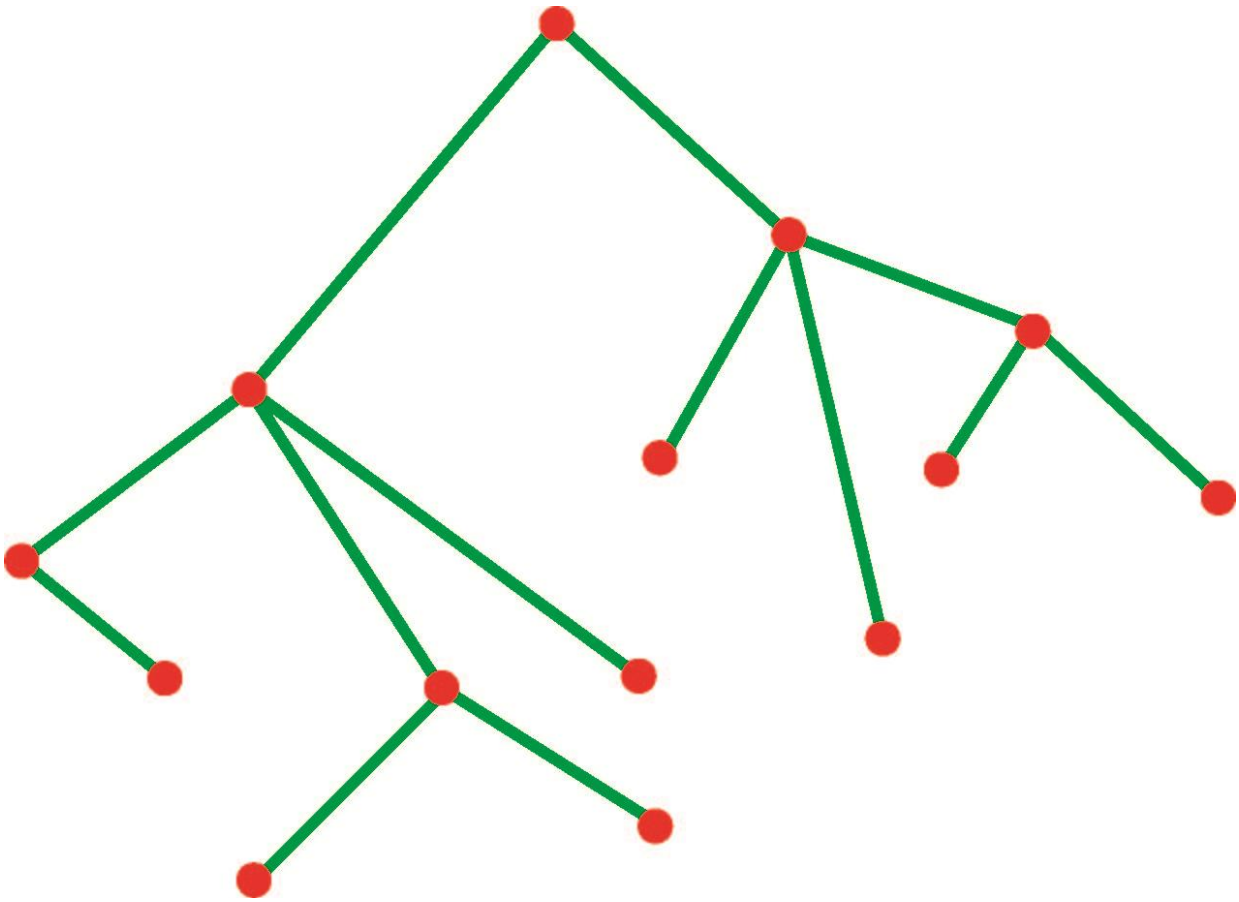

Рис. 7.1. Дерево – специфічний ациклічний граф.

Кістякове дерево зв'язного графа, що містить 9 вершин, показане на рис. 7.2. Ребра дерева виділені товстими лініями. Тонкі лінії разом з товстими утворюють сам граф, у якому виділено кістякове дерево. Зрозуміло, що кістякових дерев для будь-якого графа може бути декілька. **Отже, кістякове дерево складається з деякої підмножини ребер графа, таких, що рухаючись вздовж цих ребер можна обійти всі вершини графа**. Будь-яке кістякове дерево у графі з N вершинами має $N-1$ ребро (наприклад дерево на рис.7.2 для графа з 9 вершинами має 8 ребер). Кількість кістякових дерев у довільному графі обчислюється за допомогою матричної теореми про дерева. Для повного графа, тобто такого графа, у якого кожна вершина зв'язана з усіма іншими, кількість кістякових дерев визначається за формулою Келі: N^{N-2} (де N – число вершин графа).

Незв'язний граф,компонентами якого є дерева, називається лісом. **Кістякове дерево (spanning tree) зв'язного неорієнтованого графа – це ациклічний зв'язний підграф цього графа, що містить всі його вершини.**

.

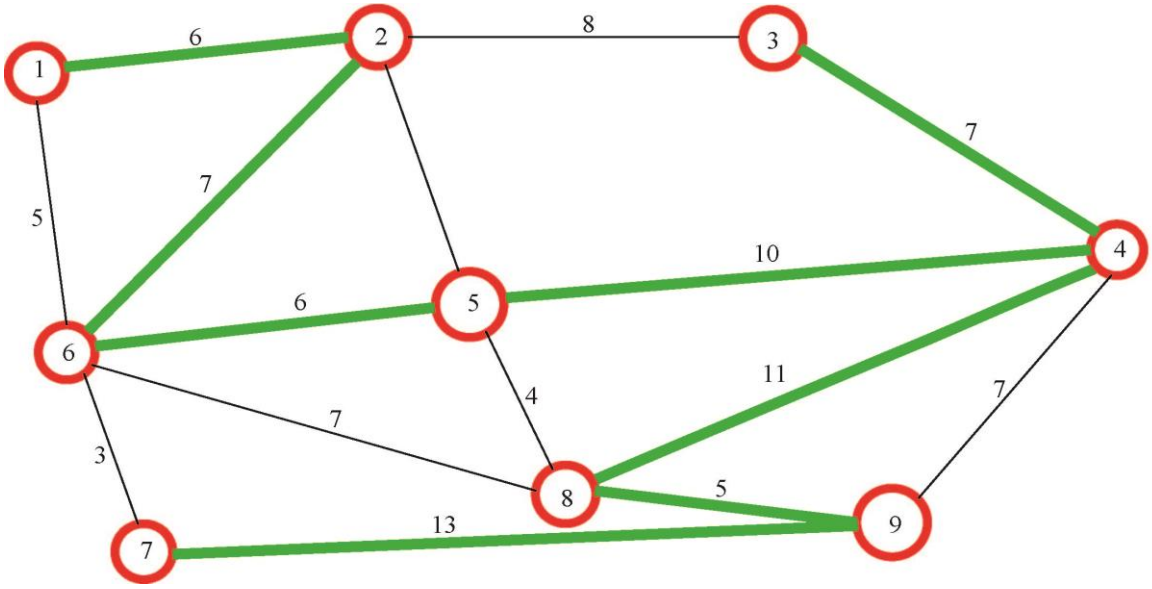

Рис. 7.2. Одне із кістякових дерев зваженого графа із 9 вершинами.

Нехай для певного зваженого графа (наприклад графа на рис. 7.2) утворений ліс кістякових дерев. Виявляється, в цьому лісі є одне особливе дерево, що називається **мінімальним кістяковим деревом**. Сума довжин ребер такого дерева є мінімальною з усіх можливих.

Отже, для всякого зваженого графа можна побудувати певну кількість кістякових дерев. Сукупність таких дерев називають лісом. Але в цьому лісі є лише одне мінімальне кістякове дерево.

Зрозуміло, що принципово важливим питанням є знаходження для даного графа мінімального кістякового дерева. Одним із самих відомих алгоритмів для знаходження такого дерева є алгоритм Прима.

Алгоритм Прима – це алгоритм побудови мінімального кістякового дерева зваженого зв'язного неорієнтованого графа. Для прикладу розглянемо граф, представлений на рис.7.3.

Вибирається ребро з найменшою вагою у графі. В нашому випадку ребро 3-6 буде саме таким ребром. Додаємо це ребро у дерево. Розглядаємо інцидентні ребра до вершин 3 та 6 та знаходимо серед них послідовно ребра з мінімальними вагами. Виберемо вершину 3. Після цього знаходимо ребро мінімальної ваги, що підходить до цієї вершини. Це буде ребро 3-5. Зі сторони вершини 6 підключаємо ребро 6-9 як ребро з найменшою вагою. Отже, на цей момент ми додали у дерево ребра 3-5, 3-6 та 6-9. Далі треба вибрати ребро мінімальної ваги, що підходить до однієї з вершин 3, 5 або 6: це ребро 5-4. Від вершини 4 відходить ребро мінімальної ваги 4-8. Далі підключаємо ребро 9-7, а потім 6-2, 2-0 і, нарешті, 0-1. Важливо зауважити, що при виборі мінімального кістякового дерева слід уникати підключення тих ребер, які приводять до утворення циклу в графі. Наприклад, не можна підключити ребро 3-2, тому що це приведе до утворення циклу *(3- 2)® (2- 6)® (6- 3)*. Нагадаємо, що дерево по визначенню – це ациклічний граф, тобто граф в якому немає циклів.

А взагалі навіщо знаходити дерева у графі? Справа в тому, що граф представляє собою символічну конструкцію, яку можна співставити, наприклад, з транспортною мережею, комп'ютерною мережею, населеним пунктом тощо. Тоді, якщо розглядати для прикладу транспортну мережу, то дороги між перехрестями можна ідентифікувати як ребра графа, а самі перехрестя доріг – як вершини. Таке саме співставлення можна зробити і для інших видів мереж. Після процедури співставлення можна на моделі графа вирішувати різного роду практичні задачі – наприклад, знаходження оптимальних варіантів для обходу всіх вершин графа (на практиці це означає, скажімо, знаходження оптимального варіанту постачання природнього газу до всіх населених пунктів району).

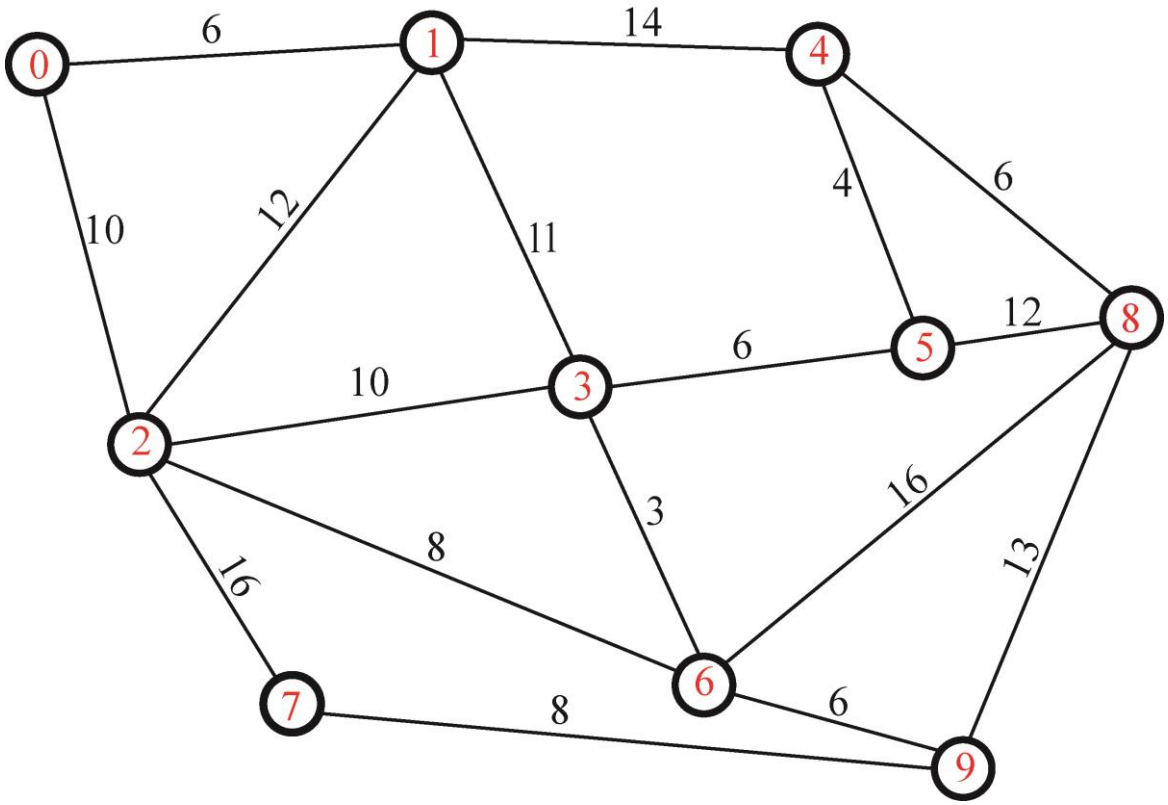

Рис.7.3. Зважений планарний простий неорієнтований граф.

Отже, потрібен алгоритм знаходження мінімального кістякового дерева у будь-якому зв'язному графі. Зокрема, для графа на рис. 7.3 алгоритм Прима на основі описаної процедури дозволяє отримати кістякове дерево, представлене на рис.7.4 виділеними (товстішими) ребрами.

На рис. 7.3 та 7.4 представлені зважені графи, тобто кожне їх ребро має певну вагу. Що означає вага ребра? Це може бути, наприклад, відстань між містами (якщо вершини графа символізують собою міста, з'єднані дорогами певної довжини). Або це може бути час проїзду між вказаними пунктами чи ступінь складності дороги.

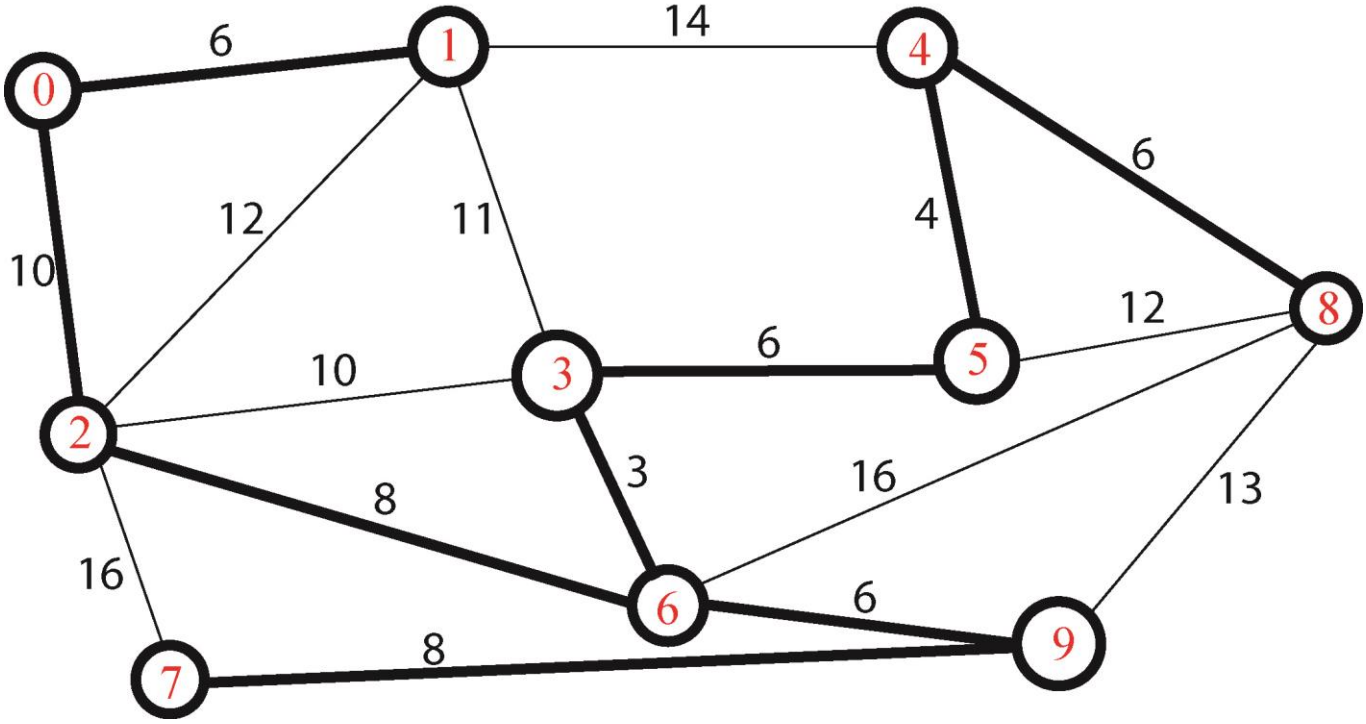

Рис.7.4.Мінімальне кістякове дерево, що фомується лише із товстіших ребер та включає в себе всі вершини графа.

Можливо при складанні логістичної карти в якості ваги ребра треба вибрати час проїзду. В таких ситуаціях доцільно ввести поняття коефіцієнта складності дороги k_R: чим складніша дорога, тим більшим буде такий коефіцієнт. Скажімо, для грунтової дороги з ямами та вибоїнами k_R виставляємо рівним 10, а для траси з асфальтобетонним покриттям – широкої, з кількома смугами, без будь-яких дефектів дорожнього покриття – $k_R = 1$. В цьому відношенні самий оптимальний варіант вибору в якості ваги ребра – це мультипліката виду $k_R \mathbf{Ч}_R$, де l_R– геометрична відстань між об'єктами (вершинами).

Для програмної реалізації алгоритму Прима необхідно задати вагову матрицю, що відповідає приведеному графу. Взагалі вагова матриця задається наступним чином

$$a_{ij} = \begin{cases} 0, \textit{ якщо } v_i = v_j \\ \infty, \textit{ якщо } v_i \textit{ та } v_j \textit{ не з'єднані дугою} \\ d, \textit{ якщо дуга } v_i v_j \textit{ має вагу } d \end{cases} \qquad (7.1)$$

Тут v_i та v_j – позначення вершин графа. З метою програмної реалізації треба перевизначити вагову матрицю так

$$a_{ij} = \begin{cases} 0, \textit{ якщо } v_i = v_j \\ 999, \textit{ якщо } v_i \textit{ та } v_j \textit{ не з'єднані дугою} \\ d, \textit{ якщо дуга } v_i v_j \textit{ має вагу } d \end{cases} \qquad (7.2)$$

Тепер для графа на рис. 7.3 отримаємо симетричну вагову матрицю виду:

$$a_{ij} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} 0 & 6 & 10 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 6 & 0 & 12 & 11 & 14 & \infty & \infty & \infty & \infty & \infty \\ 10 & 12 & 0 & 12 & \infty & \infty & 8 & 16 & \infty & \infty \\ \infty & 11 & 12 & 0 & \infty & 6 & 3 & \infty & \infty & \infty \\ \infty & 14 & \infty & \infty & 0 & 4 & \infty & \infty & 6 & \infty \\ \infty & \infty & \infty & 6 & 4 & 0 & \infty & \infty & 12 & \infty \\ \infty & \infty & 8 & 3 & \infty & \infty & 0 & \infty & 16 & 6 \\ \infty & \infty & 16 & \infty & \infty & \infty & \infty & 0 & \infty & 8 \\ \infty & \infty & \infty & \infty & 6 & 12 & 16 & \infty & 0 & 13 \\ \infty & \infty & \infty & \infty & \infty & \infty & 6 & 8 & 13 & 0 \end{pmatrix} \end{array} \qquad (7.3)$$

Використання приведеного знизу алгоритму (**package** IC-1) дозволяє знайти мінімальне кістякове дерево для графа будь-якої складності (скажімо із 150 вершинами та 450 ребрами). Ясно, що описовий алгоритм в такій ситуації використати проблематично. А от програма,

написана на Java, вирішує поставлену задачу за проміжок часу, не більший 2 с. Запустимо приведену далі програму та введемо у консоль число вершин (number of vertices) графа на рис. 7.3 та зважену матрицю (7.3).

```
package IC-1;
import java.util.InputMismatchException;
import java.util.Scanner;
public class Prims
{
        private boolean unsettled[];
        private boolean settled[];
        private int numberofvertices;
        private int adjacencyMatrix[][];
        private int key[];
        public static final int INFINITE = 999;
        private int parent[];
        public Prims(int numberofvertices)
        {
        this.numberofvertices = numberofvertices;
        unsettled = new boolean[numberofvertices + 1];
        settled = new boolean[numberofvertices + 1];
        adjacencyMatrix = new int[numberofvertices + 1][numberofvertices + 1];
        key = new int[numberofvertices + 1];
        parent = new int[numberofvertices + 1];
           }
        public int getUnsettledCount(boolean unsettled[])
        {
        int count = 0;
        for (int index = 0; index < unsettled.length; index++)
        {
        if (unsettled[index])
        {
        count++;
        } }
        return count;
        }
        public void primsAlgorithm(int adjacencyMatrix[][])
        {
        int evaluationVertex;
        for (int source = 1; source <= numberofvertices; source++)
        {
        for (int destination = 1; destination <= numberofvertices;
             destination++)
        {
        this.adjacencyMatrix[source]
        [destination] =  adjacencyMatrix[source][destination];
        }  }
        for (int index = 1; index <= numberofvertices; index++)
        {
        key[index] = INFINITE;
        }
        key[1] = 0;
        unsettled[1] = true;
```

```
parent[1] = 1;
while (getUnsettledCount(unsettled) != 0)
{
evaluationVertex = getMimumKeyVertexFromUnsettled(unsettled);
unsettled[evaluationVertex] = false;
settled[evaluationVertex] = true;
evaluateNeighbours(evaluationVertex);
} }
private int getMimumKeyVertexFromUnsettled(boolean[] unsettled2)
{
int min = Integer.MAX_VALUE;
int node = 0;
for (int vertex = 1; vertex <= numberofvertices; vertex++)
{
if (unsettled[vertex] == true && key[vertex] < min)
{
node = vertex;
min = key[vertex];
}}
return node;
}
public void evaluateNeighbours(int evaluationVertex)
{
for (int destinationvertex = 1;
destinationvertex <= numberofvertices;          destinationvertex++)
{
if (settled[destinationvertex] == false)
{
if (adjacencyMatrix[evaluationVertex][destinationvertex] != INFINITE)
{
if (adjacencyMatrix[evaluationVertex][destinationvertex] <
key[destinationvertex])
{
key[destinationvertex]
=  adjacencyMatrix[evaluationVertex][destinationvertex];
parent[destinationvertex] = evaluationVertex;
}
unsettled[destinationvertex] = true;
} } } }
public void printMST()
{
System.out.println("SOURCE  : DESTINATION = WEIGHT");
for (int vertex = 2; vertex <= numberofvertices; vertex++)
        {
System.out.println(parent[vertex] + "\t:\t" + vertex +"\t=\t"+
adjacencyMatrix[parent[vertex]][vertex]);
        } }
public static void main(String... arg)
 {
int adjacency_matrix[][];
int number_of_vertices;
Scanner scan = new Scanner(System.in);
try
      {
```

```
System.out.println("Enter the number of vertices");
number_of_vertices = scan.nextInt();
adjacency_matrix = new int[number_of_vertices + 1][number_of_vertices + 1];
System.out.println("Enter the Weighted Matrix for the graph");
for (int i = 1; i <= number_of_vertices; i++)
{
for (int j = 1; j <= number_of_vertices; j++)
{
adjacency_matrix[i][j] = scan.nextInt();
if (i == j)
{
adjacency_matrix[i][j] = 0;
continue;
            }
if (adjacency_matrix[i][j] == 0)
          {
adjacency_matrix[i][j] = INFINITE;
 } } }
Prims prims = new Prims(number_of_vertices);
prims.primsAlgorithm(adjacency_matrix);
prims.printMST();
} catch (InputMismatchException inputMismatch)
{
System.out.println("Wrong Input Format");
}
scan.close();
} }
```

Після запуску програми, введення даних та відпрацювання програми консольний інтерфейс виглядатиме наступним чином:

Enter the number of vertices
10
Enter the Weighted Matrix for the graph
0 6 10 999 999 999 999 999 999 999
6 0 12 11 14 999 999 999 999 999
10 12 0 12 999 999 8 16 999 999
999 11 12 0 999 6 3 999 999 999
999 14 999 999 0 4 999 999 6 999
999 999 999 6 4 0 999 999 12 999
999 999 8 3 999 999 0 999 16 6
999 999 16 999 999 999 999 0 999 8
999 999 999 999 6 12 16 999 0 13
999 999 999 999 999 999 6 8 13 0
SOURCE : DESTINATION = WEIGHT
0 : 1 = 6
0 : 2 = 10
6 : 3 = 3
5 : 4 = 4
3 : 5 = 6
2 : 6 = 8

```
9       :       7       =       8
4       :       8       =       6
6       :       9       =       6
Process finished with exit code 0
```

Проаналізуємо результати роботи програми. Спочатку було введено число вершин – 10. Саме стільки вершин має граф, зображений на рис.7.3. Далі в консолі було введено зважену матрицю (7.3), відповідаючу розглядуваному графу. Програма фактично виводить ребра та відповідні їм ваги. На рис.7.4 представлено мінімальне кістякове дерево графа рис.7.3. Це ж дерево було отримане і з допомогою описової алгоритмічної реалізації. Програмна реалізація дає таке ж дерево.Товстішими лініями показані ребра, що власне і складають мінімальне кістякове дерево. Як бачимо, всі вершини вихідного графа являються включеними в такий кістяковий граф. В чому полягає цінність побудови останнього? Уявімо собі, що графу на рис.7.3 відповідає населений пункт, вершинами якого є будинки, а ребрами – відстані між будинками. Якщо, наприклад, потрібно підключити всі будинки до мережі газопостачання, то з метою економії необхідно щоб сумарна довжина прокладених підземних газопостачальних труб була мінімальною. Отже, під'єднавшись до зовнішньої мережі газопостачання, наприклад, у вершині 9 (або, іншими словами, біля будинку №9) досягнемо оптимізації в плані затрат матеріальних ресурсів на прокладання газорозподільчої мережі у вибраному населеному пункті.Така ж ситуація і у випадку з іншими видами сервісу. Таким чином, алгоритм Прима має винятково важливе значення в плані побудови оптимальних логістичних мереж.

Використання приведеної вище програми спрямоване на вирішення також і інших практичних задач, таких як прокладання електричних мереж, телефонних ліній, водогонів, ліній зв'язку, каналізаційних мереж, кабельного інтернету тощо.

Алгоритм Краскала

Даний алгоритм, як і алгоритм Прима, використовується для знаходження мінімального кістякового дерева. Представимо спочатку словесний опис алгоритму, який полягає у виконанні наступної послідовності кроків:

1.У вибраному зваженому планарному графі вибираємо ребро з найменшою вагою;
2.Вибране ребро буде складати перший елемент кістякового дерева;
3.Додаємо ребро, вага якого найменша із невибраних ребер;
4. Продовжуємо вибирати ребра поки не включимо всі вершини графа, але так щоб не було жодного циклу.

Програма **package** IC-2 реалізує алгоритм Краскала. Запустимо програму та введемо матрицю відстаней графа, представленого на рис.7.3. Результат роботи програми дозволяє отримати мінімальне кістякове дерево (spanning tree), де будуть представлені ребра цього дерева в матричному вигляді.

```
package IC-2;
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;
public class KruskalAlgorithm
{
private List<Edge> edges;
private int numberOfVertices;
public static final int MAX_VALUE = 999;
private int visited[];
```

```
private int spanning_tree[][];
public KruskalAlgorithm(int numberOfVertices)
{
this.numberOfVertices = numberOfVertices;
edges = new LinkedList<Edge>();
visited = new int[this.numberOfVertices + 1];
spanning_tree = new int[numberOfVertices + 1][numberOfVertices + 1];
}
public void kruskalAlgorithm(int adjacencyMatrix[][])
{
boolean finished = false;
for (int source = 1; source <= numberOfVertices; source++)
{
for (int destination = 1; destination <= numberOfVertices; destination++)
{
if (adjacencyMatrix[source][destination] != MAX_VALUE && source != destination)
{
Edge edge = new Edge();
edge.sourcevertex = source;
edge.destinationvertex = destination;
edge.weight = adjacencyMatrix[source][destination];
adjacencyMatrix[destination][source] = MAX_VALUE;
edges.add(edge);
}}}
Collections.sort(edges, new EdgeComparator());
CheckCycle checkCycle = new CheckCycle();
for (Edge edge : edges)
{
spanning_tree[edge.sourcevertex][edge.destinationvertex] = edge.weight;
spanning_tree[edge.destinationvertex][edge.sourcevertex] = edge.weight;
if (checkCycle.checkCycle(spanning_tree, edge.sourcevertex))
{
spanning_tree[edge.sourcevertex][edge.destinationvertex] = 0;
spanning_tree[edge.destinationvertex][edge.sourcevertex] = 0;
edge.weight = -1;
continue;
}
visited[edge.sourcevertex] = 1;
visited[edge.destinationvertex] = 1;
for (int i = 0; i < visited.length; i++)
{
if (visited[i] == 0)
{
finished = false;
break;
} else
{
finished = true;
}

  }
if (finished)
break;
      }
```

```
System.out.println("The spanning tree is ");
for (int i = 1; i <= numberOfVertices; i++)
System.out.print("\t" + i);
System.out.println();
for (int source = 1; source <= numberOfVertices; source++)
{
System.out.print(source + "\t");
for (int destination = 1; destination <= numberOfVertices; destination++)
{
System.out.print(spanning_tree[source][destination] + "\t");
                    }
System.out.println();
}      }
public static void main(String... arg)
 {
int adjacency_matrix[][];
int number_of_vertices;
Scanner scan = new Scanner(System.in);
System.out.println("Enter the number of vertices");
number_of_vertices = scan.nextInt();
adjacency_matrix = new int[number_of_vertices + 1][number_of_vertices + 1];
System.out.println("Enter the Weighted Matrix for the graph");
for (int i = 1; i <= number_of_vertices; i++)
                     {
for (int j = 1; j <= number_of_vertices; j++)
          {
adjacency_matrix[i][j] = scan.nextInt();
if (i == j)
                    {
adjacency_matrix[i][j] = 0;
continue;
                }
if (adjacency_matrix[i][j] == 0)
              {
adjacency_matrix[i][j] = MAX_VALUE;
      }   }   }
KruskalAlgorithm kruskalAlgorithm = new KruskalAlgorithm(number_of_vertices);
kruskalAlgorithm.kruskalAlgorithm(adjacency_matrix);
scan.close();
  }  }
class Edge
{
int sourcevertex;
int destinationvertex;
int weight;
     }
class EdgeComparator implements Comparator<Edge>
{
@Override
public int compare(Edge edge1, Edge edge2)
{
if (edge1.weight < edge2.weight)
return -1;
if (edge1.weight > edge2.weight)
```

```
return 1;
return 0;
}   }
class CheckCycle
{
private Stack<Integer> stack;
private int adjacencyMatrix[][];
public CheckCycle()
        {
stack = new Stack<Integer>();
    }
public boolean checkCycle(int adjacency_matrix[][], int source)
        {
boolean cyclepresent = false;
int number_of_nodes = adjacency_matrix[source].length - 1;
adjacencyMatrix = new int[number_of_nodes + 1][number_of_nodes + 1];
for (int sourcevertex = 1; sourcevertex <= number_of_nodes; sourcevertex++)
  {
for (int destinationvertex = 1; destinationvertex <= number_of_nodes; destinationvertex++)
            {
adjacencyMatrix[sourcevertex][destinationvertex] = adjacency_matrix[sourcevertex][destinationvertex];
          }       }
int visited[] = new int[number_of_nodes + 1];
int element = source;
int i = source;
visited[source] = 1;
stack.push(source);
while (!stack.isEmpty())
          {
element = stack.peek();
i = element;
while (i <= number_of_nodes)
       {
if (adjacencyMatrix[element][i] >= 1 && visited[i] == 1)
       {
if (stack.contains(i))
             {
cyclepresent = true;
return cyclepresent;
     }   }
if (adjacencyMatrix[element][i] >= 1 && visited[i] == 0)
              {
stack.push(i);
visited[i] = 1;
adjacencyMatrix[element][i] = 0;// mark as labelled;
adjacencyMatrix[i][element] = 0;
element = i;
i = 1;
continue;
}
i++;
        }
stack.pop();
      }
```

```
return cyclepresent;
    }    }
```

Знизу приведено консоль програми:

```
Enter the number of vertices
10
Enter the Weighted Matrix for the graph
0   6   10  999 999 999 999 999 999 999
6   0   12  11  14  999 999 999 999 999
10  12  0   12  999 999 8   16  999 999
999 11  12  0   999 6   3   999 999 999
999 14  999 999 0   4   999 999 6   999
999 999 999 6   4   0   999 999 12  999
999 999 8   3   999 999 0   999 16  6
999 999 16  999 999 999 999 0   999 8
999 999 999 999 6   12  16  999 0   13
999 999 999 999 999 999 6   8   13  0

The spanning tree is
      0   1   2   3   4   5   6   7   8   9
0     0   6   10  0   0   0   0   0   0   0
1     6   0   0   0   0   0   0   0   0   0
2     10  0   0   0   0   0   8   0   0   0
3     0   0   0   0   0   6   3   0   0   0
4     0   0   0   0   0   4   0   0   6   0
5     0   0   0   6   4   0   0   0   0   0
6     0   0   8   3   0   0   0   0   0   6
7     0   0   0   0   0   0   0   0   0   8
8     0   0   0   0   6   0   0   0   0   0
9     0   0   0   0   0   0   6   8   0   0

Process finished with exit code 0.
```

Виділений напівжирним курсивом масив – це дані, які вводяться в консолі. Далі – результат роботи програми, що фактично являє собою матрицю відстаней. Якщо по цій матриці побудувати граф, то в точності отримаємо мінімальне кістякове дерево, зображене на рис.7.4. Як бачимо, результати роботи обох описаних алгоритмів – Прима і Краскала – аналогічні. Результати апробації цих двох алгоритмів на потужних графах (з великою кількістю вершин та ребер: понад 150 та 450 відповідно) показують, що алгоритм Прима працює бездоганно, а алгоритм Краскала в деяких випадках може не вказати однієї вершини.

Алгоритм Флойда

По своїй суті це досить простий алгоритм. Він дозволяє знайти спектр найкоротших відстаней між всіма вершинами графа. Принцип роботи алгоритма Флойда полягає у визначенні мінімальної відстані шляхом порівняння двох величин, а саме відстані між сусідніми вершинами графа, взятої напряму, та відстані між цими ж вершинами, але пройденої через інші два ребра.

$$A_k(i,j) = \min\left(A_{k-1}(i,j), A_{k-1}(i,k) + A_{k-1}(k,j)\right) \qquad (7.4)$$

Графічно ситуація виглядає як показано на рис. 7.5. Аналізуючи цей рисунок, виникає питання:який обрати шлях – від i до j напряму чи застосувати обхідний маневр – пройти через вершину k ? Відповідь на поставлене питання дає формула (7.4) – із двох порівнюваних варіантів

шляху треба вибрати коротший. Алгоритм перебирає всі можливі варіанти, тобто проходить по всім вершинам k та знаходить найкоротший шлях між i та j. Далі описана процедура повторюється аж поки не буде знайдений оптимальний маршрут між всіна вершинами графа. Зрозумілою тепер стає практична значимість графа, адже він дає можливість прокладати оптимальні маршрути, припустимо між населеними пунктами області.

.

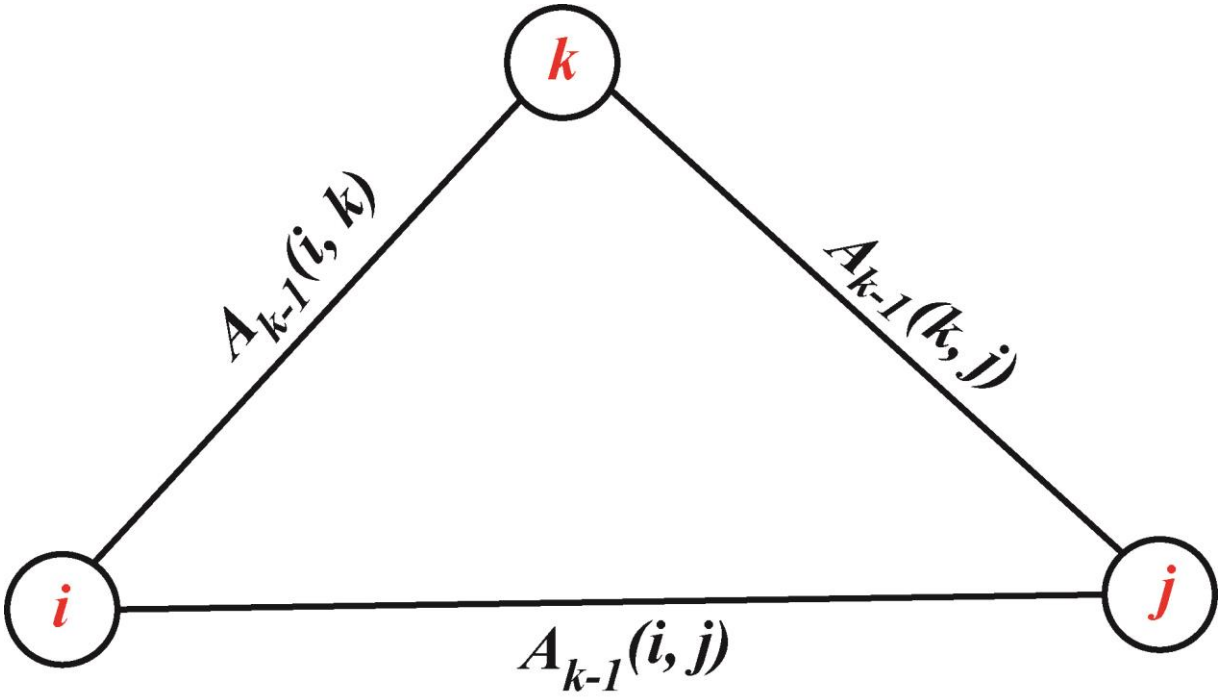

Рис.7.5. Елементарний фрагмент у алгоритмі пошуку оптимального шляху в графі.

Нехай маємо орграф

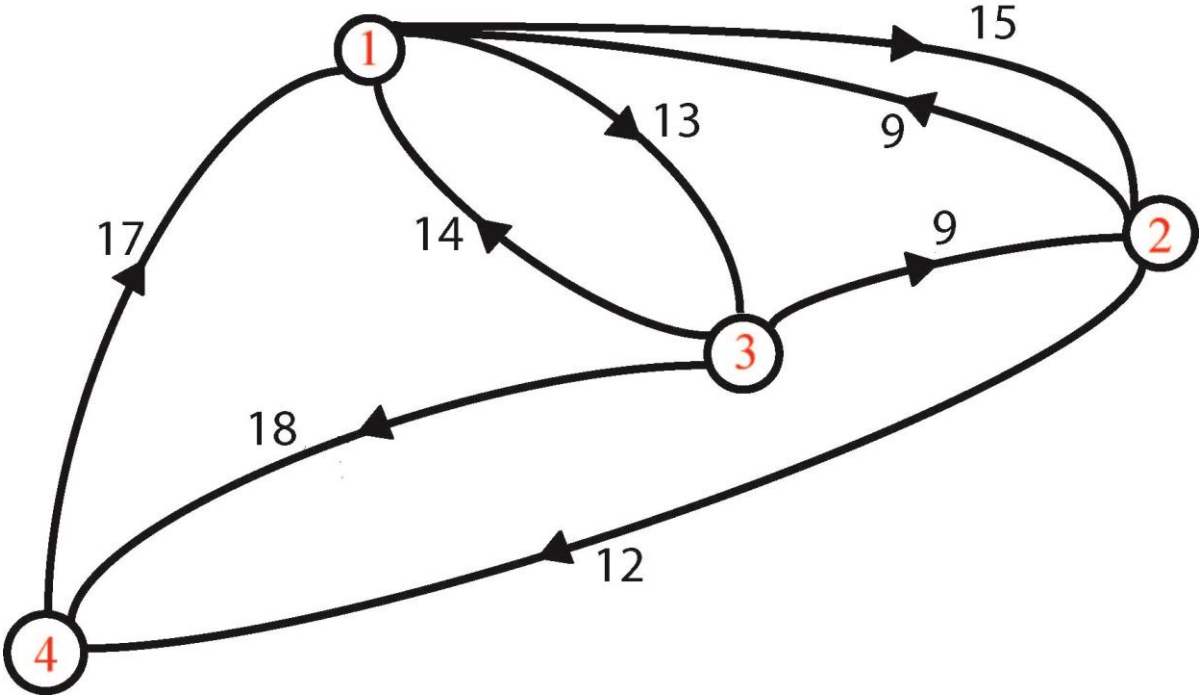

Рис.7.6. Орієнтований навантажений планарний мультиграф.

Для даного графа необхідно знайти мінімальні відстані між всіма вершинами. Для цієї мети використаємо алгоритм Флойда. Сформуємо для такого графа вагову матрицю A_{ij} у відповідності

із визначенням (7.1). На нульовому кроці алгоритму це буде просто вагова матриця графа на рис.7.6.

$$A_0 = \begin{pmatrix} 0 & 15 & 13 & \infty \\ 9 & 0 & \infty & 12 \\ 14 & 9 & 0 & 18 \\ 17 & \infty & \infty & 0 \end{pmatrix}$$

На першому кроці ітераційного процесу із співвідношення (7.4) отримаємо:

$$A_1(i, j) = \min\left(A_0(i, j), A_0(i,1) + A_0(1, j)\right) . \qquad (7.5)$$

Як бачимо, в цьому виразі змінна k приймає значення 1. Відповідна матриця матиме вигляд

$$A_1 = \begin{pmatrix} 0 & 15 & 13 & \infty \\ 9 & 0 & 22 & 12 \\ 14 & 9 & 0 & 18 \\ 17 & 32 & 30 & 0 \end{pmatrix}.$$

Порівнюючи матрицю A_1 з матрицею A_0 бачимо, що деякі елементи матриці A_1 змінились: в результаті проходження по всім елементам матриці з допомогою виразу (7.5) змінюємо ті елементи матриці, які виявляються меншими за попередні: наприклад, елемент $(A_1)_{2,3}$ став рівним 22, а в матриці A_0 цей елемент був рівним ∞ . Аналогічним чином проходимо другий, третій та четвертий ітераційні кроки, для яких k змінюється відповідно в напрямку $2 \to 4$ (у формулі (7.5) замість 1 виставляємо послідовно 2,3 і 4). Остаточно матимемо матрицю

$$A_4 = \begin{pmatrix} 0 & 15 & 13 & 27 \\ 9 & 0 & 22 & 12 \\ 14 & 9 & 0 & 18 \\ 17 & 32 & 30 & 0 \end{pmatrix}.$$

Власне це і є матриця найкоротших відстаней. Наприклад, від вершини 1 до вершини 4 мінімальний шлях буде рівним 27 – це буде елемент матриці $(A_4)_{14}$. До речі, дістатися від вершини 1 до вершини 4 можна двома маршрутами: $1 \to 3 \to 4$ та $1 \to 2 \to 4$. Проте останній маршрут – коротший і тому алгоритм вибирає саме цей маршрут.

Отже, якщо A_0 – матриця ваг графа, то згідно алгоритму Флойда пробігаємо по всім вершинам цього графа і шукаємо більш короткий шлях через вершину k, реалізуючи всі варіанти, що відповідають співвідношенню (7.4). Фактично, для графа з V вершинами пройдені вище кроки зводяться до виконання потрійного циклу *for* :

```
{for (int k = 0; k <V; k++)//пробігаємо по всіх вершинах графа
for (int i = 0; i < V; i++)
for (int j = 0; j < V; j++)
if (dist[i][k] + dist[k][j] < dist[i][j]) {//знаходимо найкоротший шлях
dist[i][j] = dist[i][k] + dist[k][j];
next[i][j] = next[i][k];}.
```

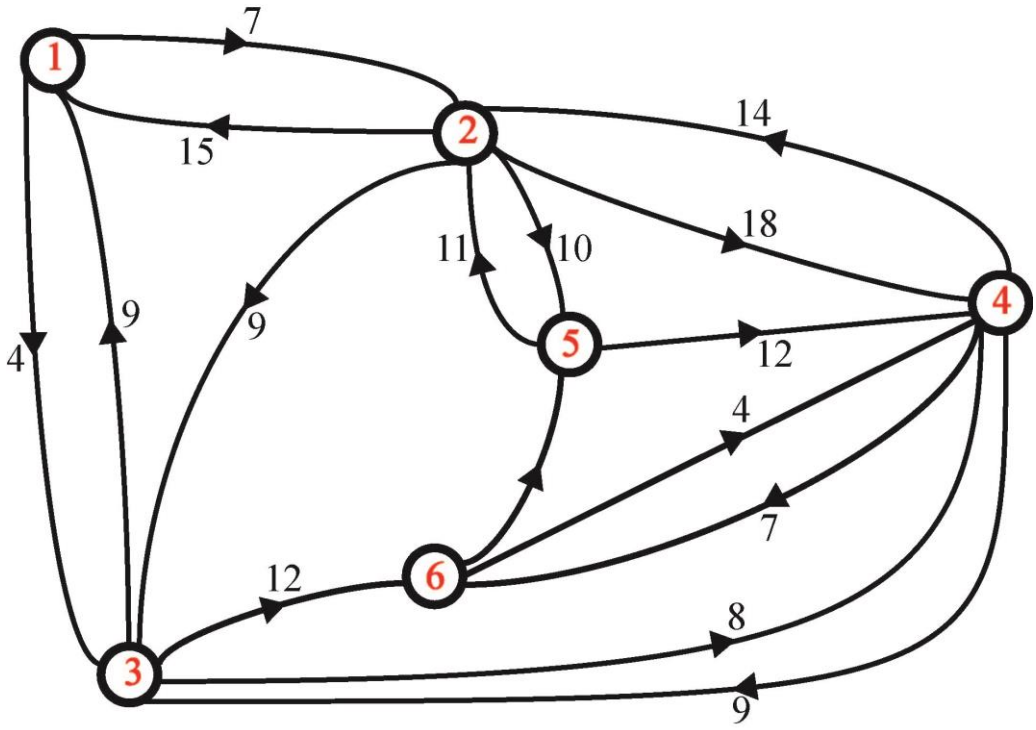

Рис.7.7. Навантажений орієнтований граф, відстані між вершинами якого розраховує програма **package** IC-3.

Звичайно, використовувати приведений вище формульно-словесний алгоритм Флойда для об'ємних графів нереально. Наприклад, для графа із 16 ребрами та 6 вершинами, зображеного на рис.7.7, приведена знизу програма **package** IC-3 дозволяє знаходити відстані за долі секунди (в той час як використання описового алгоритму в даному випадку вимагатиме набагато більше часу).

Ваги ребер в **package** IC-3 задаються в основному коді цієї програми. Зверніть також увагу, що ваги ребер у приведеному графі не корелюють зовсім з геометричними відстанями між вершинами ребер.

У програмі **package** IC-3 двовимірний масив

```
int[][] weights = {{1,2,7}, {2,1,15}, {1,3,4}, {3,1,9}, {2,3,9}, {2,5,10},{5,2,11}, {2,4,18}, {4,2,14},
{5,4,12}, {6,5,4}, {3,4,8}, {4,3,9}, {3,6,12}, {4,6,7}, {6,4,4}};
```

слід розуміти наступним чином: перше число – це номер вершини, з якої виходить ребро, друге – це номер вершини, у яку входить ребро, і третє – це вага такого ребра. І таким чином задається будь-який граф. Запустимо програму та проаналізуємо результат її роботи.

```
package IC-3;
import static java.lang.String.format;
import java.util.Arrays;
public class FloydWarshall {
public static void main(String[] args) {
int[][] weights = {{1, 2, 7},{2,1,15}, {1, 3, 4},{3,1,9}, {2, 3, 9},{2,5,10},
 {5, 2, 11},{2,4,18}, {4, 2, 14},{5,4,12}, {6,5,4},{3,4,8},{4,3,9},{3,6,12},
{4,6,7},{6,4,4}};
int numVertices = 6;
floydWarshall(weights, numVertices);
```

```
    }
static void floydWarshall(int[][] weights, int numVertices) {
double[][] dist = new double[numVertices][numVertices];
for (double[] row : dist)
Arrays.fill(row, Double.POSITIVE_INFINITY);
for (int[] w : weights)
dist[w[0] - 1][w[1] - 1] = w[2];
int[][] next = new int[numVertices][numVertices];
for (int i = 0; i < next.length; i++) {
for (int j = 0; j < next.length; j++)
if (i != j)
next[i][j] = j + 1;
      }
for (int k = 0; k < numVertices; k++)//пробігаємо по всіх вершинах графа
for (int i = 0; i < numVertices; i++)
for (int j = 0; j < numVertices; j++)
if (dist[i][k] + dist[k][j] < dist[i][j]) {//знаходимо найкоротший шлях
dist[i][j] = dist[i][k] + dist[k][j];
next[i][j] = next[i][k];
                }
printResult(dist, next);
    }
static void printResult(double[][] dist, int[][] next) {
System.out.println("pair     dist    path");
for (int i = 0; i < next.length; i++) {
for (int j = 0; j < next.length; j++) {
if (i != j) {
int u = i + 1;
int v = j + 1;
String path = format("%d -> %d    %2d     %s", u, v,
(int) dist[i][j], u);
do {
u = next[u - 1][v - 1];
path += " -> " + u;
} while (u != v);
System.out.println(path);
  } } } }  }
```

Результат виконання програми:

```
pair     dist    path
1 ® 2     7     1®  2
1®  3     4     1®  3
1®  4    12      1 ® 3 ® 4
1 ® 5    17      1 ® 2 ® 5
1 ® 6    16      1®  3 ® 6
2 ® 1    15      2®  1
2 ® 3     9     2 ®  3
2 ® 4    17      2 ® 3 ® 4
2 ® 5    10      2 ® 5
2 ® 6    21      2 ® 3® 6
3 ® 1     9      3 ® 1
```

```
3 ® 2   16   3 ® 1® 2
3 ® 4    8   3®  4
3 ® 5   16   3®  6® 5
3 ® 6   12   3 ® 6
4 ® 1   18   4 ® 3® 1
4 ® 2   14   4 ® 2
4 ® 3    9   4®  3
4 ® 5   11   4 ® 6 ® 5
4 ® 6    7   4®  6
5 ® 1   26   5 ® 2 ® 1
5 ® 2   11   5 ® 2
5 ® 3   20   5 ® 2 ® 3
5 ® 4   12   5 ® 4
5 ® 6   19   5 ® 4® 6
6 ® 1   22   6 ® 4® 3 ® 1
6 ® 2   15   6 ® 5® 2
6 ® 3   13   6 ® 4® 3
6 ® 4    4   6® 4
6 ® 5    4   6® 5
Process finished with exit code 0
```

Тут представлені найкоротші маршрути між всіма вершинами графа, а також довжини цих маршрутів. Скажімо, найкоротший шлях в напрямку від вершини 6 до вершини 1, тобто маршрут $6 \to 1$, дорівнює 22 та пролягає вздовж простого ланцюга $6 \to 4 \to 3 \to 1$.

З практичної точки зору така програма може бути ефективно використана для прокладання оптимальних маршрутів у будь-яких графах. В свою чергу графи є, фактично, відображенням мереж будь-якої природи. Це можуть бути авіатраси, автомобільні дороги (міста, країни, континенту), інтернет-мережі, газогони, електричні мережі, водні артерії. І навіть артерії та вени людського організму.

Алгоритм Дейкстри

Існує багато способів знаходження мінімального шляху в графі. Точніше сказати – оптимального маршруту. Під оптимальністю можна розуміти, наприклад, мінімальний час проходження маршруту, чи мінімальні витрати палива, затраченого на проходження маршруту, або мінімальну відстань. В цьому сенсі алгоритм Дейкстри досить ефективний та близький по суті і по принципу роботи до алгоритму Флойда. Але на відміну від згаданого, дозволяє **знайти спектр найкоротших відстаней між вибраною вершиною графа та всіма іншими його вершинами**. Суть алгоритму Дейкстри полягає у порівнянні відстаней до сусідніх (інцидентних) вершин та знаходження оптимального вибору на кожному кроці. Розглянемо рис.7.8. Тут представлений навантажений планарний простий неорієнтований граф. Вершини цього графа позначені літерами латинського алфавіту. Біля ребер графа вставлені їх ваги. Звернемо увагу, що ваги ребер зовсім не корелюють з їх геометричною довжиною, тому що їх визначення залежить від постановки конкретної задачі. На цю обставину ми звернемо особливу увагу далі. А зараз лише зазначимо, що вага ребра по суті може представляти собою будь-яку сутність. Важливо зауважити, що вага одного і того ж ребра часто буває змінною з часом величиною. Розглянемо для прикладу, транспортні магістралі між містами. У погожий літній день такі транспортні артерії володіють великою пропускною здатністю і автомобілі по них рухаються з великими швидкостями. Але в зимній період під час снігопадів та заметілей ситуація корінним чином змінюється – транспортні засоби застрягають у снігових заметах і тому прохідність такої дороги (іншими словами, ребра графа) стає рівною нулеві. Із сказаного випливає, що потрібно створити базу даних, в якій

зберігаються ваги ребер та постійно оновлювати і базу даних, і ваги ребер у відповідності із реаліями, що мають місце. Особливо наочно динамічність ваг ребер проявляється у випадку із транспортною мережею у місті, де ситуація змінюється щосекундно та з'являються затори на перехрестях.

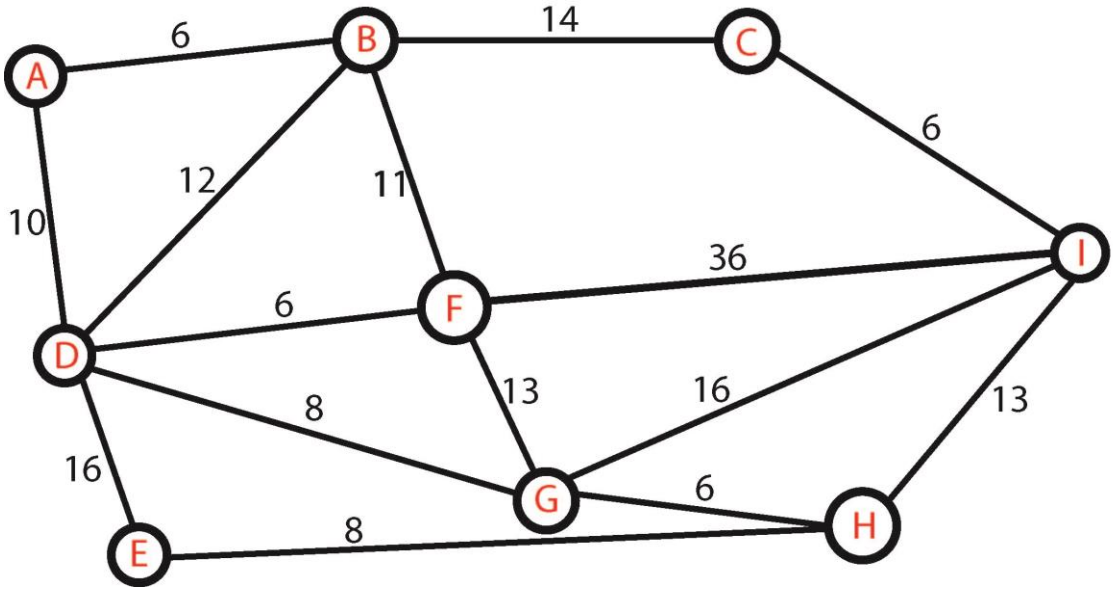

Рис. 7.8. Навантажений неорієнтований граф, в якому з допомогою алгоритму Дейкстри необхідно знайти найкоротші відстані між вершиною A та всіма іншими вершинами. Вивно, що ваги ребер не корелюють з геометричними відстанями між вершинами графа. Наприклад, найдовше ребро E - H має порівняно невелику вагу. Це означає, що вага ребра визначається виключно умовами поставленої задачі та може представляти собою будь-яку величину, що стосується особливостей вирішуваної проблеми.

Для цього графа вагова матриця з урахуванням виразу (7.1) записується у наступному вигляді

$$a_{ij} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \end{array} \begin{array}{c} \begin{array}{ccccccccc} A & B & C & D & E & F & G & H & I \end{array} \\ \begin{pmatrix} 0 & 6 & \infty & 10 & \infty & \infty & \infty & \infty & \infty \\ 6 & 0 & 14 & 12 & \infty & 11 & \infty & \infty & \infty \\ \infty & 14 & 0 & \infty & \infty & \infty & \infty & \infty & 6 \\ 10 & 12 & \infty & 0 & 16 & 6 & 8 & \infty & \infty \\ \infty & \infty & \infty & 16 & 0 & \infty & \infty & 8 & \infty \\ \infty & 11 & \infty & 6 & \infty & 0 & 13 & \infty & 36 \\ \infty & \infty & \infty & 8 & \infty & 13 & 0 & 6 & 16 \\ \infty & \infty & \infty & \infty & 8 & \infty & 6 & 0 & 13 \\ \infty & \infty & 6 & \infty & \infty & 36 & 16 & 13 & 0 \end{pmatrix} \end{array}. \quad (7.5)$$

По головній діагоналі даної матриці стоять нулі. Символ І свідчить про те, що вибрані вершини – не інцидентні. Звернемо також увагу на те, що дана матриця симетрична, тобто $a_{ij} = a_{ji}$. Наприклад, $a_{24} = a_{42} = 12$.

Табл. 7.1. Словесний алгоритм Дейкстри

Крок	Відмічені вершини	Відстань до вершини									Невідмічені вершини
		A	B	C	D	E	F	G	H	I	
1	A	0	6	∞	10	∞	∞	∞	∞	∞	B,C,D,E,F,G,H,I
2	B	0	6	20	10	∞	17	∞	∞	∞	C,D,E,F,G,H,I
3	D	0	6	20	10	26	16	18	∞	∞	C,E,F,G,H,I
4	F	0	6	20	10	26	16	18	∞	52	C,E,G,H,I
5	G	0	6	20	10	26	16	18	24	34	C,E,H,I
6	C	0	6	20	10	26	16	18	24	26	E,H,I
7	H	0	6	20	10	26	16	18	24	26	E,I
8	E	0	6	20	10	26	16	18	24	26	I
9	I	0	6	20	10	26	16	18	24	26	

Виконання алгоритму Дейкстри будемо здійснювати покроково:

Крок 1: Оскільки в задачі стоїть завдання знайти відстані від вершини A до всіх інших вершин (рис.7.8), то це значить, що вершина A пройдена і тому на першому кроці відмічаємо цю вершину як пройдену, а всі інші вершини графа записуємо у рядок невідмічених вершин (табл.7.1). Крім того, у першому рядку таблиці виставляємо відстані від вершини A до всіх інших вершин. Зрозуміло, що цей рядок буде співпадати з першим рядком матриці ваг (7.5).

Крок 2: Відмічаємо вершину B, оскільки вона є найближчою до A. Далі обчислюємо довжини шляхів, що ведуть від вершини A до невідмічених вершин через вершину B. Якщо нові значення виявляються менше старих, то змінюємо останні на нові. Через вершину B є три варіанти проходження: ABD, ABF та ABC. Спектр відповідних відстаней буде: 18, 17 та 20. Отже, попередні значення до вершин F та C замінюються на нові, а відстань до вершини D не змінюється. Значить, тепер другий рядок таблиці у порівнянні з першим зміниться у двох клітинках – (2-C) та (2-F).

Крок 3: Звернемо увагу тепер на рядок №2 табл.7.1. Із вершин, що залишились невідміченими, тобто C,D,E,F,G,H,I, найближче до A знаходиться вершина D. Із A через вершину D у невідмічені вершини ідуть три ланцюги – ADF, ADG та ADE з відповідними довжинами – 16, 18 та 26. Вносимо відповідні зміни у рядок №3, врахувавши, що вершина D вже відмічена і прокладати до неї маршрут не можна. Знову ж таки знаходимо найменше значення серед невідмічених на даний момент вершин: це буде вершина F.

Крок №4. Через вершини D і F до непройдених вершин пролягає два шляхи – ADFI та ADFG з відповідними вагами 52 та 29. Відповідно у рядку №4 змінюємо лише відстань до вершини I та знаходимо мінімальне значення для непройдених вершин – воно рівне 18 і стосується вершини G. У цю вершину шлях пролягав через вершину D – відповідно маємо два ланцюги: ADGI, ADGH з довжинами 34 та 24. Вносимо тепер зміни у рядок №5. Аналізуючи цей рядок, приходимо до висновку, що вершині C відповідає мінімальна величина – 20.

Крок №5. Через вершину C проходить лише один ланцюг – ABCI довжини 26. Тому вносимо відповідні зміни у рядок №6. В цьому ж таки рядку знаходимо мінімальне значення для не пройдених вершин E,H,I. Воно виявляється рівним 24 та стосується вершини H. Проглядаємо відповідні ланцюги – ADGHI та ADGHE. Їм відповідають довжини 37 та 32 відповідно. Бачимо, що в даному випадку ніяких змін вносити не треба.

Крок №6. Вибираємо вершину E. Через цю вершину проходить лише один маршрут – ADEH з довжиною 34. Тому змін теж немає ніяких. Далі вибираємо вершину I. І в цьому випадку ситуація незмінна. Взагалі для даного графа останні чотири рядки повторюються. Всі вершини тепер пройдені. Результат записаний в останньому рядку таблиці.

Зрозуміло, що приведений вище словесний алгоритм Дейкстри важливий з точки зору розуміння його роботи. На практиці застосовують програмні варіанти. Один з таких задається лістингом **package IC-4.** Але оскільки названа програма працює з числовими символами, то спочатку необхідно перепозначити вершини графа на рис. 7.8, замінивши $A \to 1,\ B \to 2$ і т.д.

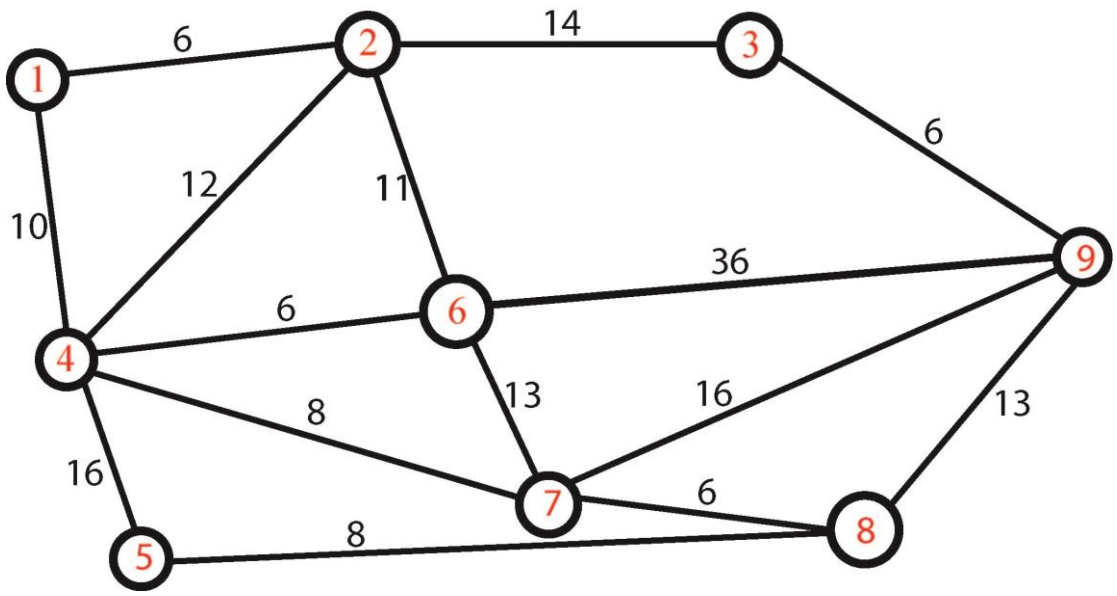

Рис. 7.9. Зважений неорієнтований планарний простий граф, в якому за допомогою алгоритму Дейкстри знаходяться відстані між вершиною 1 та всіма іншими вершинами.

Отже, матимемо граф, представлений на рис. 7.9, повністю ідентичний графу рис. 7.8 (в теорії графів такі графи називають ізоморфними).
Обчислювальне ядро алгоритму перебирає всі можливі варіанти вибору маршруту в графі та відшукує оптимальний. Далі приводимо всю програму повністю. Після виконання цієї програми достатньо лише ввести список суміжності для ваг ребер заданого графа і програма обрахує оптимальний маршрут між вибраною вершиною (вершина 1) та всіма іншими вершинами графа.

```
package IC-4;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.StringTokenizer;
public class MiniWay {
private static int INF = Integer.MAX_VALUE/2;
double weightU;
int u;
private int n; //кількість вершин у графі
private int m; //кількість дуг у графі
private ArrayList adj[]; //список суміжності
private ArrayList weight[]; //вага ребра в орграфі
private boolean used[]; //масив для зберігання інформації про пройдені
 // та не пройдені вершини
private double dist[]; //массив для зберігання відстані від стартової вершини
```

```
private int[] pred;//масив предків, необхідних  для відновлення
 // найкоротшого шляху від стартової вершини
int start; //стартова вершина, від якої знаходимо відстань до всіх інших
private BufferedReader cin;
private PrintWriter cout;
private StringTokenizer tokenizer;
private void dejkstra(int s) {//процедура запуска алгоритма Дейкстри із
 // стартової вершини
dist[s] = 0; //найкоротша відстань до стартової вершини рівна 0
for (int iter = 0; iter < n; ++iter) {
int v = -1;
double distV = INF;
for (int i = 0; i < n; ++i) {
if (used[i]) {
continue;
        }
if (distV < dist[i]) {
continue;
        }
v = i;
distV = dist[i];
       }
for (int i = 0; i < adj[v].size(); ++i) {
int u = (int) adj[v].get(i);
double weightU = (double) weight[v].get(i);
if (dist[v] + weightU < dist[u]) {
dist[u] = dist[v] + weightU;
pred[u] = (int) v;  }}
used[v] = true;           }}
private void readData() throws IOException {
       cin = new BufferedReader(new InputStreamReader(System.in));
       cout = new PrintWriter(System.out);
       tokenizer = new StringTokenizer(cin.readLine());
       n = Integer.parseInt(tokenizer.nextToken());
       m = Integer.parseInt(tokenizer.nextToken());
       start = Integer.parseInt(tokenizer.nextToken()) - 1;
       adj = new ArrayList[n];
       for (int i = 0; i < n; ++i) {
       adj[i] = new ArrayList();
          }
       //ініціалізація списку, в якому зберігаються ваги ребер
       weight = new ArrayList[n];
       for (int i = 0; i < n; ++i) {
       weight[i] = new ArrayList();
          }
       //зчитуємо граф, заданий списком ребер
       for (int i = 0; i < m; ++i) {
       tokenizer = new StringTokenizer(cin.readLine());
       int u = Integer.parseInt(tokenizer.nextToken());
       int v = Integer.parseInt(tokenizer.nextToken());
       double w = Double.parseDouble(tokenizer.nextToken());
       u--;
       v--;
       adj[u].add(v);
```

```
    weight[u].add(w);
       }
    used = new boolean[n];
    Arrays.fill(used, false);
    pred = new int[n];
    Arrays.fill(pred, -1);
    dist = new double[n];
    Arrays.fill(dist, INF);
    }
    void printWay(int v) {
    if (v == -1) {
    return;
       }
    printWay(pred[v]);
    cout.print((v + 1) + " ");
    }
    private void printData() throws IOException {
    for (int v = 0; v < n; ++v) {
    if (dist[v] != INF) {
    cout.print(dist[v] + " ");
    } else {
    cout.print("-1 ");
          }
       }
    cout.println();
    for (int v = 0; v < n; ++v) {
    cout.print((v + 1) + ": ");
    if (dist[v] != INF) {
    printWay(v);
          }
    cout.println();
       }
    cin.close();
    cout.close();
    }
    private void run() throws IOException {
    readData();
    dejkstra(start);
    printData();
    cin.close();
    cout.close();
    }
    public static void main(String[] args) throws IOException {
    MiniWay solution = new MiniWay();
    solution.run();
    }
}
```

Результат роботи програми:

```
9  13  1
1  2  6
1  4  10
2  4  12
4  5  16
```

```
4  6  6
4  7  8
5  8  8
7  8  6
7  9  16
6  9  36
2  3  14
3  9  6
8  9  13
0.0  6.0  20.0  10.0  26.0  16.0  18.0  24.0  26.0
1: 1
2: 1 2
3: 1 2 3
4: 1 4
5: 1 4 5
6: 1 4 6
7: 1 4 7
8: 1 4 7 8
9: 1 2 3 9
Process finished with exit code 0
```

Проаналізуємо програму **package IC-4** та результат її роботи**.** Перша тріада чисел задає відповідно число вершин графа, число ребер графа та номер вершини, від якої розраховуються відстані до всіх інших вершин (в нашому випадку це вершина 1). Тепер з другого рядка у консолі вводимо тріади чисел у послідовності: номер однієї вершини, далі номер інцидентної вершини і третє число – вага відповідного ребра. І так вводимо всі 13 ребер графа (рис. 7.9) та натискаємо клавішу Enter. Після відпрацювання програма видає результат: і). Послідовність із 9 чисел, що власне і являють собою найкоротші відстані від вершини 1 до всіх інших вершин; іі). Дев'ять рядків, що символізують собою оптимальний маршрут. Скажімо, запис *9: 1 2 3 9* слід розуміти так – маршрут до вершини 9 пролягає вздовж ланцюга $1 \to 2 \to 3 \to 9$. Відповідно довжина цього оптимального ланцюга складає 26.0 одиниць.

Доцільно порівняти результати роботи словесного алгоритму Дейкстри та варіанту **package IC-4** його програмної реалізації: маємо повний збіг.

A-алгоритм*

Суть такого алгоритму полягає у знаходження найкоротшого шляху у графі між двома вибраними вершинами. Це так званий евристичний алгоритм. Його принципова особливість полягає у введенні поняття евристичної відстані. Позначимо цю відстань h(n), де n – номер вибраної вершини графа. Це така відстань, яка прокладається від кожної проміжної вершини графа до кінцевої. Далі вводиться функція f(n), що являє собою суму евристичної відстані від поточної вершини та відстані від цієї вершини до кінцевої, тобто f(n) = g(n) + h(n), де g(n) – це відстань від стартової вершини до вершини n, але пройдена вздовж ребер графа. Завдяки введенню евристики, зменшується число можливих варіантів вибору шляху і тому алгоритмічна складність A^*-алгоритму невелика. На відміну від алгоритмів Дейкстри та Флойда, цей алгоритм має лінійну алгоритмічну складність (алгоритм Дейкстри – квадратичну, а алгоритм Флойда – кубічну). Далі буде приведений словесний варіант а також програмна реалізація A^*-алгоритму та показані шляхи його використаня при прокладанні оптимальних маршрутів у різного роду мережах.

Розглянемо граф, представлений на рис.7.10. Припустимо, треба знайти оптимальний маршрут $A \to P$.

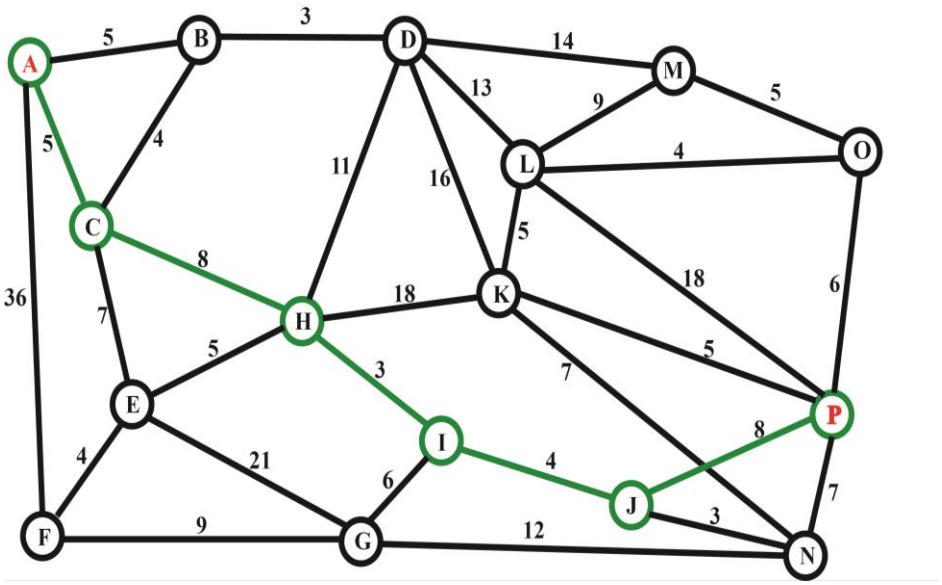

Рис.7.10. Зважений граф, в якому з допомогою A*-алгоритму знаходиться найкоротша відстань між вершиною A та вершиною P.

Сформуємо таблицю, в яку з самого початку занесемо згадані вже евристичні відстані h(n). Фактично це особливі відстані, що вимірюються від поточної вершини n до кінцевої вершини маршруту по прямій. Отже, такі величини є, звичайно, найкоротшими з усіх можливих. Для чого вводиться евристична відстань h(x)? Така процедура необхідна для знаходження мінімальних величин f(x) і на цій основі прокладання оптимального маршруту. Іншими словами, вибирається найкоротший ланцюг, що пролягає від стартової вершини через поточну вершину n до кінцевої. Технічно процес прокладання найкращого шляху в павутинні графа виглядає наступним чином:
1). Нехай стартовою вершиною в нашій задачі є вершина A, а фінішною – вершина P.

Табл.7.2. A*-алгоритм – перший крок

Вершина	Відстань від A, g(x)	Евристична відстань, g(x)	f(x)=g(x)+h(x)	Попередня вершина
A	0	16	16	
B	5	17	22	A
C	5	13	**18**	A
D		16		
E		16		
F	36	20	56	A
G		17		
H		11		
I		10		
J		8		
K		5		
L		7		
M		10		
N		7		
O		5		
P		0		

Інцидентними до вершини A є вершини B, C та F. Ваги ребер AB, AC та AF виставляємо у другій колонці табл.7.2. У четвертій колонці таблиці проставляємо суми f(x) = g(x) + h(x). В колонці

Табл.7.3. A*-алгоритм – другий крок

Вершина	Відстань від A, g(n)	Еврbстична відстань, h(n)	f(n)=g(n)+h(n)	Попередня вершина
A	0	16	16	
B	5, ***9***	17	22, ***26***	A, ***C***
C	5	13	18	A
D	***12***	16	***28***	***C***
E	***12***	16	***28***	***C***
F	36	20	56	A
G		17		
H	**13**	11	***24***	***C***
I		10		
J		8		
K		5		
L		7		
M		10		
N		7		
O		5		
P		0		

«Попередня вершина» проставляємо назву вершини, із якої відбувся перехід – у вершини B, C та F ми потрапили із A. Порівнюємо три величини f(x) – 22, 18 та 56. Вибираємо найменшу із них, рівну 18. Вершину A тепер розглядаємо як закриту та виділяємо відповідний рядок сірим кольором.

2). Оскільки величина функції f(n) для вершини C є найменшою (цифра 18 в таблиці 7.2 виділена), то тепер ця вершина стає поточною. Вершина C має 4 інцидентних вершини – B, D, H та E (рис.7.10). Обчислимо відстані до цих вершин через вершину C та виставимо їх у табл. 7.3;

Табл.7.4. A*-алгоритм – третій крок

Вершина	Відстань від A, g(n)	Евристична відстань, h(n)	f(т)=g(n)+h(n)	Попередня вершина
A	0	16	16	
B	5	17	*22*	A
C	5	13	18	A
D	12, ***8***	16	28, ***24***	C, ***B***
E	12, ***40***	16	28, ***56***	C,***F***
F	36	20	56	A
G	***45***	17	***52***	***F***
H	13	11	24	C
I		10		
J		8		
K		5		
L		7		
M		10		
N		7		
O		5		
P		0		

іншими словами – обчислимо величини функцій g(n). В стовпчику для функції f(n) виставимо відповідні суми, а вершину C вважатимемо пройденою. Щоб продемонструвати це, виділимо рядок C сірим кольором. Отримані величини та назви вершин на даному етапі виділимо напівжирним курсивом;

3). В табл. 7.3 звернемо увагу на рядок B – в ньому слід залишити меншу пару величин (так власне потрібно робити і далі), що ми і зробимо у табл. 7.4.
4). Тепер наш шлях пролягає у вершину B. Інцидентна їй не пройдена вершина D. В табл. 7.4 внесемо відповідні проміжні дані та приєднаємо вершину B до пройдених, виділивши у табл. 7.4 рядок з вершиною B сірим кольором. У рядку D залишаємо до розгляду меншу пару величин – 8 та 24.Тепер наш шлях пролягає у вершину F. Проводимо аналогічні процедури із цією вершиною та виставляємо результати у табл.7.4. Наша задача – це задача оптимізації, тому у табл. 7.5 виділяємо курсивом пари з мінімальними значеннями, тобто D та H;
5). Вибираємо H в якості поточної вершини. Серед непройдених вершин цій вершині

Табл.7.5. A*-алгоритм – четвертий крок

Вершина	Відстань від A, g(n)	Евристична відстань, h(n)	f(x)=g(n)+h(n)	Попередня вершина
A	0	16	16	
B	5	17	*22*	A
C	5	13	18	A
D	8,***24***	16	*24*,***40***	B,***H***
E	12,***18***	16	28, ***34***	C,***H***
F	36	20	56	A
G	45	17	52	F
H	13	11	*24*	C
I	***16***	10	***26***	***H***
J		8		
K	***31***	5	***36***	***H***
L		7		
M		10		
N		7		
O		5		
P		0		

Табл.7.6. A*-алгоритм – п’ятий крок

Вершина	Відстань від A, g(n)	Евристична відстань, h(n)	f(n)=g(n)+h(n)	Попередня вершина
A	0	16	16	
B	5	17	*22*	A
C	5	13	18	A
D	8	16	24	B
E	12	16	28	C
F	36	20	56	A
G	45	17	52	F
H	13	11	24	C
I	16	10	*26*	H
J	20	8	28	I
K	31,***24***	5	36,***29***	H,***D***
L	***21***	7	***28***	***D***
M	***22***	10	***32***	***D***
N		7		
O		5		
P		0		

інцидентними є D, K, I та E. В табл. 7.5 виставляємо значення величин g(n) та f(n), маючи на увазі, що маршрути до цих вершин пролягають через H. У колонці «Попередня вершина» виставляємо H. Як і раніше, залишаємо пари величин з меншими значеннями та виділяємо вершину H як пройдену. Переходимо, таким чином, до табл.7.6;

5). Аналізуючи у колонці для величин f(n) спектр даних для непройдених вершин, знаходимо мінімальне значення 24 і тому в якості поточної вершини вибираємо D. Виділяємо інцидентні непройдені вершини – M,L,K – та проставляємо у табл. 7.6 довжини ланцюгів до цих вершин, якщо маршрут пролягатиме через D. Закриваємо вершину D, виділивши відповідний рядок сірим кольором. Аналізуючи спектр величин у колонці для f(n) приходимо до висновку, що тепер поточною має стати вершина I, оскільки величина 26 є мінімальною. Вводимо у розгляд вершину J та закриваємо I, виділивши сірим кольором відповідний рядок та перейшовши до табл.7.7.

6). У колонці для f(n) присутні три однакових мінімальних величини із значенням 28. Доцільно в якості поточної вибрати J-вершину, оскільки в цьому випадку інцидентною стає фінішна вершина P, тобто фінішна вершина. Аналізуючи отримані результати, представлені в колонці для f(n), приходимо до висновку, що коротшого шляху як через цю вершину просто не існує;

Табл.7.7. A*-алгоритм – шостий крок

Вершина	Відстань від A, g(x)	Еврістична відстань, g(x)	f(x)=g(x)+h(x)	Попередня вершина
A	0	16	16	
B	5	17	22	A
C	5	13	18	A
D	8	16	24	B
E	12	16	28	C
F	36	20	56	A
G	45	17	52	F
H	13	11	24	C
I	16	10	26	H
J	20	8	28	I
K	24	5	29	D
L	21	7	28	D
M	22	10	32	D
N	***23***	7	***30***	***J***
O		5		
P	***28***	0	***28***	***I***

7). Наше завдання тепер – відновити оптимальний маршрут. Це можна зробити за допомогою табл. 7.7: у вершину P ми прийшли із J, а у J прибуття відбулось із I; далі H,C та A. Отже, пройдений шлях виглядає так: $A \to C \to H \to I \to J \to P$. Його довжина дорівнює 28 та є найменшою із усіх можливих.

Для порівняння результатів розглянемо програмний варіант A*-алгоритму, застосувавши його до графа на рис.7.10. Код програми представлений лістингом **package IC-4.** Запустимо програму та прослідкуємо за результатом.

```
package IC-4;
import java.util.PriorityQueue;
import java.util.HashSet;
import java.util.Set;
    import java.util.List;
    import java.util.Comparator;
    import java.util.ArrayList;
    import java.util.Collections;
    public class AstarSearchAlgo {
```

```
public static void main(String[] args) {
//вводимо евристичні відстані від кожної з вершин графа до кінцевої//вершини P
Node n1 = new Node("A", 16);
Node n2 = new Node("B", 17);
Node n3 = new Node("C", 13);
Node n4 = new Node("D", 16);
Node n5 = new Node("E", 16);
Node n6 = new Node("F", 20);
Node n7 = new Node("G", 17);
Node n8 = new Node("H", 11);
Node n9 = new Node("I", 10);
Node n10 = new Node("J", 8);
Node n11 = new Node("K", 5);
Node n12 = new Node("L", 7);
Node n13 = new Node("M", 10);
Node n14 = new Node("N", 7);
Node n15 = new Node("O", 5);
Node n16 = new Node("P", 0);
//ініціалізуємо ребра, ввівши ваги всіх інцидентних вершин
n1.adjacencies = new Edge[]{
      new Edge(n2, 5),
      new Edge(n3, 5),
      new Edge(n6, 36),};
n2.adjacencies = new Edge[]{
      new Edge(n1, 5),
      new Edge(n4, 3),
      new Edge(n3, 4) };
n3.adjacencies = new Edge[]{
      new Edge(n1, 5),
      new Edge(n2, 4),
      new Edge(n4, 7),
      new Edge(n8, 8),
      new Edge(n5, 7)};
n4.adjacencies = new Edge[]{
      new Edge(n2, 3),
      new Edge(n13, 14),
      new Edge(n12, 13),
      new Edge(n11, 16),
      new Edge(n8, 11),
      new Edge(n3, 7),};
n5.adjacencies = new Edge[]{
      new Edge(n3, 7),
      new Edge(n8, 5),
      new Edge(n6, 4),
      };
n6.adjacencies = new Edge[]{
      new Edge(n1, 36),
      new Edge(n5, 4),
      new Edge(n7, 9),};
n7.adjacencies = new Edge[]{
      new Edge(n6, 9),
      new Edge(n9, 6),
      new Edge(n14, 12),
      };
```

```
    n8.adjacencies = new Edge[]{
            new Edge(n5, 5),
            new Edge(n3, 8),
            new Edge(n4, 11),
            new Edge(n11, 18),
            new Edge(n9, 3) };
    n9.adjacencies = new Edge[]{
            new Edge(n8, 3),
            new Edge(n10, 4),
            new Edge(n7, 6), };
    n10.adjacencies = new Edge[]{
            new Edge(n9, 4),
            new Edge(n16, 8),
            new Edge(n14, 3) };
    n11.adjacencies = new Edge[]{
            new Edge(n8, 18),
            new Edge(n4, 16),
            new Edge(n12, 5),
            new Edge(n16, 5),
            new Edge(n14, 7) };
    n12.adjacencies = new Edge[]{
            new Edge(n4, 13),
            new Edge(n13, 9),
            new Edge(n15, 4),
            new Edge(n11, 5),
            };
    n13.adjacencies = new Edge[]{
            new Edge(n4, 14),
            new Edge(n15, 5),
            new Edge(n12, 9),
              };
    n14.adjacencies = new Edge[]{
            new Edge(n7, 12),
            new Edge(n10, 3),
            new Edge(n11, 7),
            new Edge(n16, 7),
             };
    n15.adjacencies = new Edge[]{
            new Edge(n13, 5),
            new Edge(n12, 4),
            new Edge(n16, 6), };
    n16.adjacencies = new Edge[]{
            new Edge(n14, 7),
            new Edge(n10, 8),
            new Edge(n11, 5),
            new Edge(n15, 6),
            };// введемо початкову та кінцеву вершини
    AstarSearch(n1, n16);
    List<Node> path = printPath(n16);
    System.out.println("Path: " + path);}
public static List<Node> printPath(Node target) {
    List<Node> path = new ArrayList<Node>();
    for (Node node = target; node != null; node = node.parent) {
       path.add(node);}
```

```
        Collections.reverse(path);
        return path;}
    public static void AstarSearch(Node source, Node goal) {
        Set<Node> explored = new HashSet<Node>();
        PriorityQueue<Node> queue = new PriorityQueue<Node>(30,new Comparator<Node>() {
            //перевизначимо метод порівняння
            public int compare(Node i, Node j) {
                if (i.f_scores > j.f_scores) {
                    return 1;
                } else if (i.f_scores < j.f_scores) {
                    return -1;
                } else {
                    return 0;
                }}});
        //вага від стартової вершини A
        source.g_scores = 0;
        queue.add(source);
        boolean found = false;
        while ((!queue.isEmpty()) && (!found)) {
            //вузол, що має найменше значення f_score
            Node current = queue.poll();
            explored.add(current);
            //ціль знайдена
            if (current.value.equals(goal.value)) {
                found = true;}
            //перевірка кожного нащадка поточного вузла
            for (Edge e : current.adjacencies) {
                Node child = e.target;
                double cost = e.cost;
                double temp_g_scores = current.g_scores + cost;
                double temp_f_scores = temp_g_scores + child.h_scores;
  / * якщо вузол-нащадок оцінений і
     нова величини  f_score більша, тоді залишити попередній варіант * /
                if ((explored.contains(child)) &&
                        (temp_f_scores >= child.f_scores)) {
                    continue;}
                else if ((!queue.contains(child)) ||
                        (temp_f_scores < child.f_scores)) {
                    child.parent = current;
                    child.g_scores = temp_g_scores;
                    child.f_scores = temp_f_scores;
                    if (queue.contains(child)) {
                        queue.remove(child);}
                    queue.add(child);
                }}}}}
class Node {
    public final String value;
    public double g_scores;
    public final double h_scores;
    public double f_scores = 0;
    public Edge[] adjacencies;
    public Node parent;
    public Node(String val, double hVal) {
        value = val;
```

```
        h_scores = hVal;}
    public String toString() {
        return value;
    }}
class Edge {
    public final double cost;
    public final Node target;
    public Edge(Node targetNode, double costVal) {
        target = targetNode;
        cost = costVal;
    }}
```

Результат роботи програми виглядає наступним чином:

Path: [A, C, H, I, J, P]. Process finished with exit code 0.

Отриманий шлях співпадає із знайденим раніше з допомогою «ручного» алгоритму (див. текст після табл.7.7). A*-алгоритм можна назвати інтелектуальним (або евристичним), оскільки він відсікає неперспективні варіанти маршрутів завдяки аналізу значень функції f(n).

Комп'ютерні ігри, графи та A-алгоритм*

Ігрове поле в комп'ютерній грі часто використовується для переміщення гравця (ігрового агента) з одного місця на інше, скажімо, з точки A в точку B. Тому важливо поставити питання «Як зробити таке переміщення оптимальним з точки зору геометричної протяжності маршруту переміщення?». Іншими словами, необхідно прокласти найкоротший маршрут в ігровому полі. Загалом переміщення об'єкта із однієї ігрової позиції в іншу можливе з допомогою кількох варіантів, але в любому випадку необхідно вирішити наступні проблеми:

1. Як дістатись із позиції A в B?
2. Як обійти перешкоди під час руху?
3. Як знайти найкоротший (оптимальний) варіант шляху із всіх можливих варіантів?
4. Як знайти оптимальний маршрут якнайшвидше?

Найефективнішим способом досягнення поставлених цілей є використання A*-алгоритму, який застосовується для прокладання оптимальних маршрутів у графах. Нехай ігрове поле представляється складеним із квадратних комірок (плиток). Плитки пронумеровані: їх номера проставлені всередині плиток зверху зліва (рис. 7.11).

Завдання полягає в тому щоб оптимальним чином провести ігрового агента із стартової ігрової позиції у фінішну. Для вирішення цієї проблеми на першому етапі скористаємось описовим варіантом A*-алгоритму. Для цього знову розглянемо функцію f(n) = g(n) + h(n), де g(n) – функція, що визначає вагу (відстань) маршруту від стартової позиції до прохідної плитки з номером n, а h(n) – функція евристичної відстані. Зауважимо, що визначення цієї функції неоднозначне та залежить від постановки задачі і характеристик ігрового поля. Стандартна евристична функція для квадратної ігрової решітки являє собою так звану Манхеттенську відстань. Введення такого типу евристичної функції обумовлене тим, що в багатьох випадках ігровий агент може рухатись тільки вздовж горизонталей та вертикалей – саме так розташовані квартали Манхеттена – центральної частини міста Нью-Йорк. Якраз таку евристику ми будемо використовувати в нашій задачі.

Проте досить часто евристична відстань вимірюється по прямій, оскільки пряма є найкоротшою відстанню між двома точками. Зокрема, використовуючи A*-алгоритм (рис. 7.10), евристика являла собою пряму між кожною поточною вершиною n та кінцевою вершиною маршруту P. Зауважимо також, що у деяких випадках виміряти евристичну відстань проблематично або взагалі неможливо (скажімо, коли мова йде про інтернет-мережу, то розташування серверів та окремих комп'ютерів може бути невідомим!). В цьому випадку необхідно використовувати алгоритми без застосування евристики.

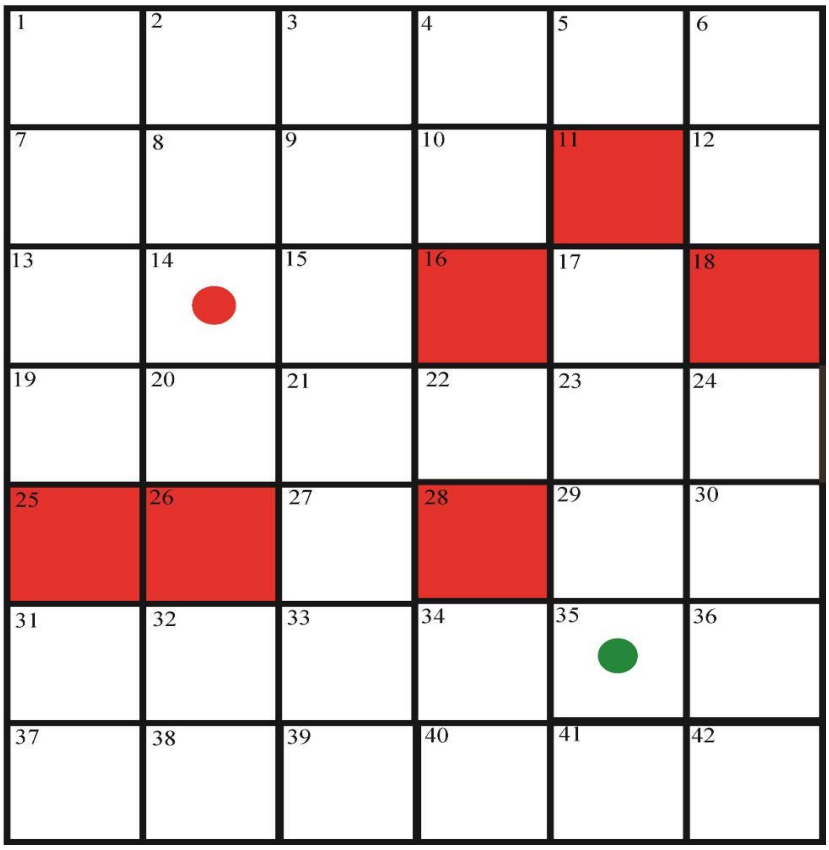

Рис.7.11. Ігрове поле з перешкодами – закрашеними комірками. Зліва вверху у кожній комірці проставлені їх номери. В комірці 14 знаходиться стартова позиція ігрового агента, а в комірці 35 – фінішна.

На рис. 7.12 показано два можливих манхеттенських маршрути. Маршрут 1 проходить виключно по відкритих плитках. Але часто ігрове поле має високу густину закритих плиток і тому треба прокладати маршрут через закриті ігрові плитки, як це показано на на рис.7.12 з допомогою маршруту під номером 2.

Як підкреслювалось вище часто, на противагу Манхеттенській евристиці, доцільно розраховувати відстань між поточною вершиною та фінішною вершиною по прямій. Іншими словами, потрібно використовувати так звану евклідову евристику, вимірюючи саме найкоротшу відстань між двома точками. При цьому необхідно щораз обчислювати квадратний корінь, але при відповідній швидкодії процесора така перешкода не є суттєвою. Фактично треба розрахувати спектр відстаней виду $h(n)= Math.sqrt((x_n - x_f)^2 + (y_n - y_f)^2)$, де x_n – координата поточної вершини, взята по вісі абсцис; x_f – аналогічна координата, але для фінішної вершини; y_n – координата поточної вершини, взята по вісі ординат; y_f – y-координата фінішної вершини.

Інколи евристика може представляти собою суму двох відрізків. Це може трапитись у випадку, коли граф розділений на дві частини, які з'єднані між собою лише одним ребром (в теорії графів таке ребро називається мостом). Припустимо, необхідно прокласти оптимальний маршрут між вершинами графа, одна з яких знаходиться у одній частині графа, а інша – у частині графа, відділеній від першої частини мостом. В такому випадку треба застосувати евристику у вигляді двох прямих відрізків: один із них пролягає від стартової вершини до моста, а інший – від моста до фінішної вершини.

Отже, вибір евристичної функції залежить від виду конкретного графа (іншими словами, від геометрії розтушування об'єктів, що моделюються графом).

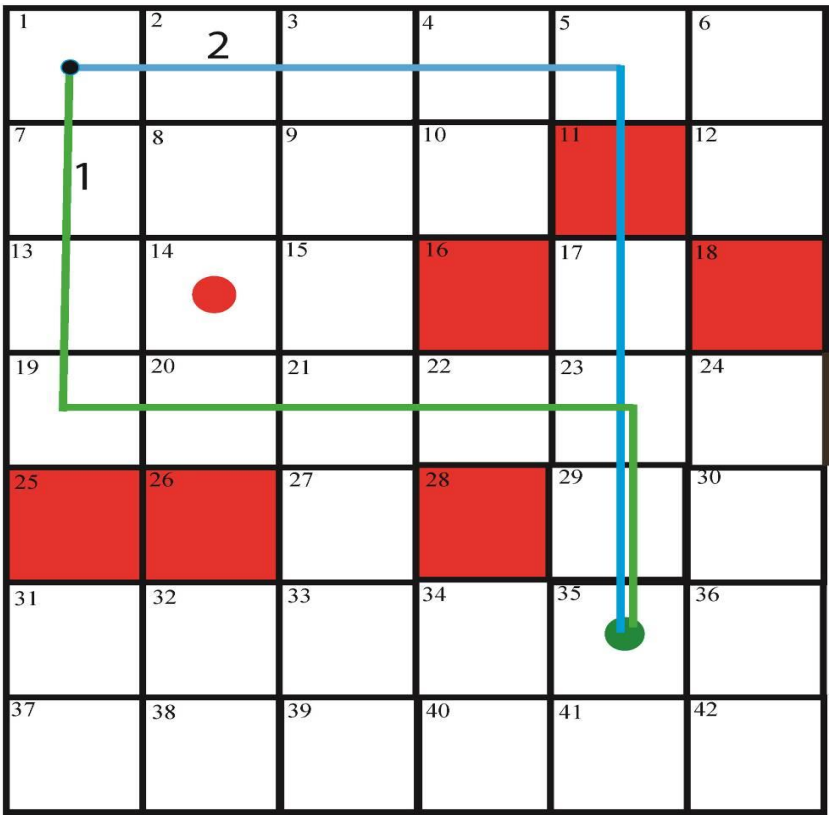

Рис.7.12. Маршрути, що з'єднують комірки 1 та 35, демонструють метод визначення евристичної відстані.

Нехай кожна плитка (tile) має розміри 10×10. В ігровому полі зобразимо перешкоди – це закрашені плитки. В комірках такого типу ігровий агент перебувати не може, на відміну від відкритих комірок, через які можливе пересування ігрових об'єктів. Стартова позиція ігрового агента виділена та являє собою кружок, що знаходиться в комірці під номером 14. Фінішна позиція – це кружок, що знаходиться в комірці 35. Необхідно провести ігрового агента із стартової позиції у фінішну найкоротшим шляхом. Вважається, що ігровий агент може рухатись по горизонтальних відкритих плитках, по вертикальних відкритих плитках а також в діагональному напрямку плиток такого типу. Оскільки в іграх, як правило, ставиться завдання пройти із стартової позиції у фінішну оптимальним маршрутом з використанням A*-алгоритму, то необхідно визначити принцип обрахування евристичної відстані. В даній задачі використаємо обрахунок евристичних відстаней як показано на рис.7.12. Для конкретного випадку евристичну функцію можна записати

$$h(n) = abs(x_n - x_f) + abs(y_n - y_f) \qquad (7.6)$$

де x_n – х-координата однієї із прохідних плиток, а x_f – х-координата фінішної плитки, abs – позначення абсолютної величини, y_n – у-координата прохідної плитки, y_f – у-координата фінішної плитки.

Якщо рух ігрового агента дозволений по діагоналях плиток, тоді евристика має значно більш складну структуру. В такому випадку, очевидно, більш доцільним варіантом буде використання в якості евристичної відстані або пряму лінію, що з'єднує початкову та кінцеву точку ігрового агента, або кілька прямолінійних відрізків, що проходять між забороненими для проходження плитками.

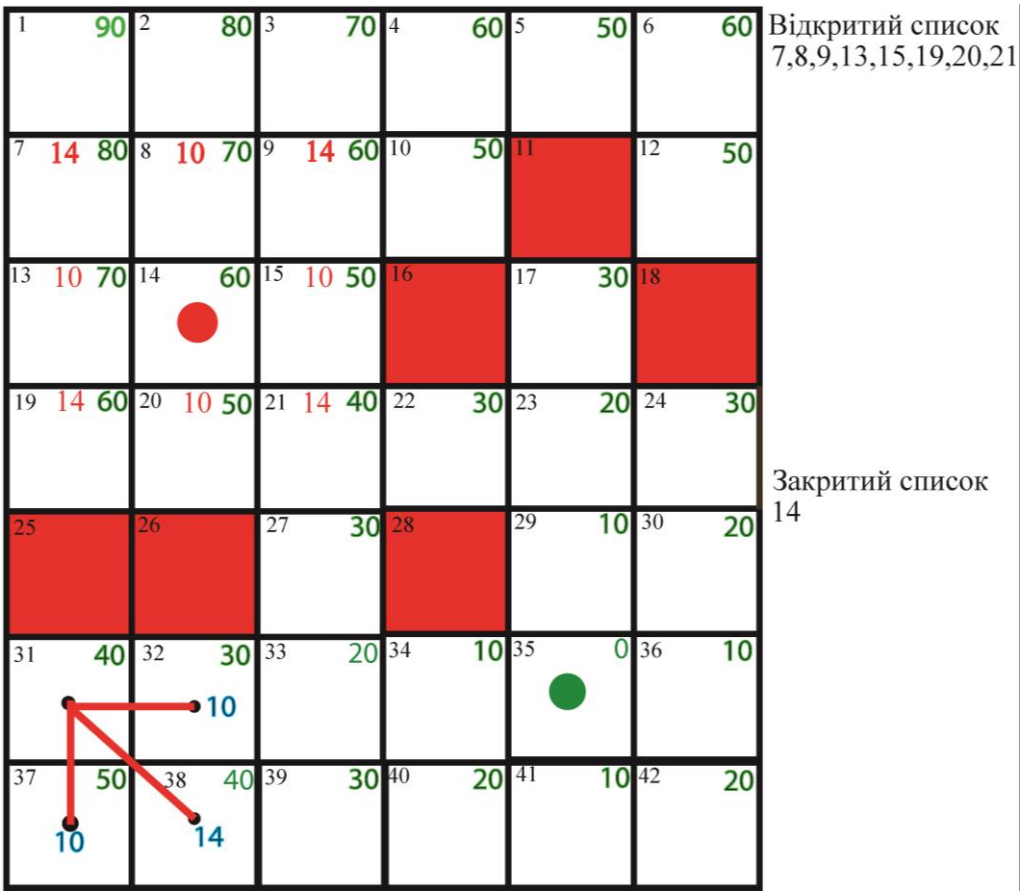

Рис.7.13. Навколо стартової позиції агента (плитка 14) розташовані вісім плиток, які формують так званий відкритий список. У відкритих плитках вверху справа виставлені величини евристичних відстаней.

Вважаючи, що відстань між центрами сусідніх плиток по горизонталі та вертикалі складає 10 умовних одиниць, можна обрахувати всі евристичні відстані між будь-якою коміркою та коміркою 35, де знаходиться кінцева позиція руху ігрового агента. Обрахунок здійснюється за формулою $h(n) = 10 \times k$, де k – число проміжків між центрами початкової та кінцевої плиток (наприклад, вздовж ламаних ліній на рис.7.12 число таких проміжків дорівнює 9).

Сформуємо відкритий список – список плиток, що оточують стартову позицію (рис. 7.13). Цей список спочатку буде складатись із 8 плиток з номерами 7,8,9,13,15,19,20 та 21. Закритий список буде містити всього одну плитку за номером 14, тобто стартову плитку. Наступний крок – визначення величин f(n) = g(n) + h(n). Тут n – номер проміжної плитки, тобто плитки, що лежить між стартовою та фінішною плитками. Функція g(n) представляє собою реальну відстань між стартовою та проміжною плитками, а h(n) – це так звана евристична відстань, яка в нашій задачі вимірюється як показано на рис.7.12 – або вздовж маршруту 1, або вздовж маршруту 2.

Зупинимось більш детально на поняттях прохідних та непрохідних плиток. Зрозуміло, що це в певній мірі абстракція. Реально маршрут в ігровому полі може складатися із плиток різної прохідності або словами теорії графів – із ребер з різними вагами. Така ситуація є більш природньою, оскільки моделює реальний земний ландшафт з його рівнинами, горами, лісами та річками. В цьому випадку доцільно ввести поняття коефіцієнта складності маршруту k_{ij}. Для кожної ігрової плитки (або для кожного відрізка маршруту) матричні елементи k_{ij} приймають значення, що відповідають реальній складності проходження конкретного елементу шляху.

Іншими словами, якщо ігрове поле (рис. 7.11) складається із плиток з різною прохідністю, тоді таку ситуацію можна графічно представити у вигляді рис. 7.14.

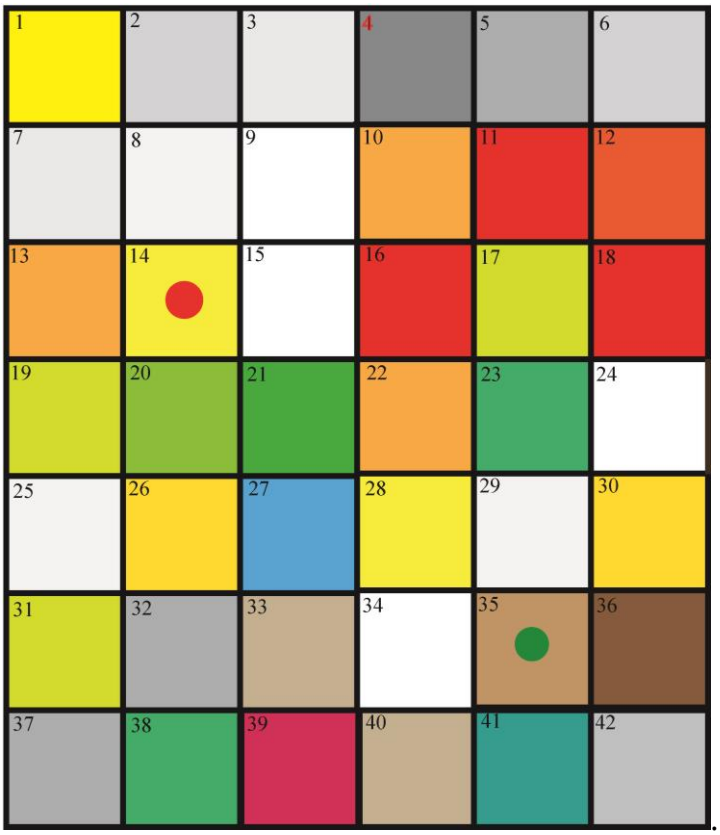

Рис.7.14. Ігрове поле з різною прохідністю плиток.

Якщо умовно встановити відповідність між кольором плитки та її прохідністю (наприклад, білий колір – $k = 1$, червоний – $k = 2$, оранжевий – $k = 3$, жовтий – $k = 4$, зелений – $k = 5$, голубий – $k = 6$, синій – $k = 7$, фіолетовий – $k = 8$, сірий $k = 9$, чорний – $k = 10$), тоді будь-якому ігровому полю можна у відповідність з прохідністю його плиток співставити матрицю

$$k_{ij} = \begin{pmatrix} k_{11} & k_{12} & k_{13} \dots k_{1i} & \dots k_{1n} \\ k_{21} & k_{22} & k_{23} \dots k_{2i} & \dots k_{2n} \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ k_{j1} & k_{j2} & k_{j3} \dots k_{ji} & \dots k_{jn} \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ k_{n1} & k_{n2} & k_{n3} \dots k_{ni} & \dots k_{nn} \end{pmatrix} \qquad (7.7)$$

Після такого співставлення довжина маршруту ігрового агента в горизонтальному або вертикальному напрямках може бути представлена у вигляді

$$W = \sum_{ij} k_{ij} l_{ij} \ , \qquad (7.8)$$

де l_{ij} – матриця довжин, що відповідає геометричній довжині кожної плитки; якщо всі плитки ігрового поля еквівалентні, то тоді із (7.8) отримуємо

$$W = l \sum_{ij} k_{ij} . \qquad (7.9)$$

.

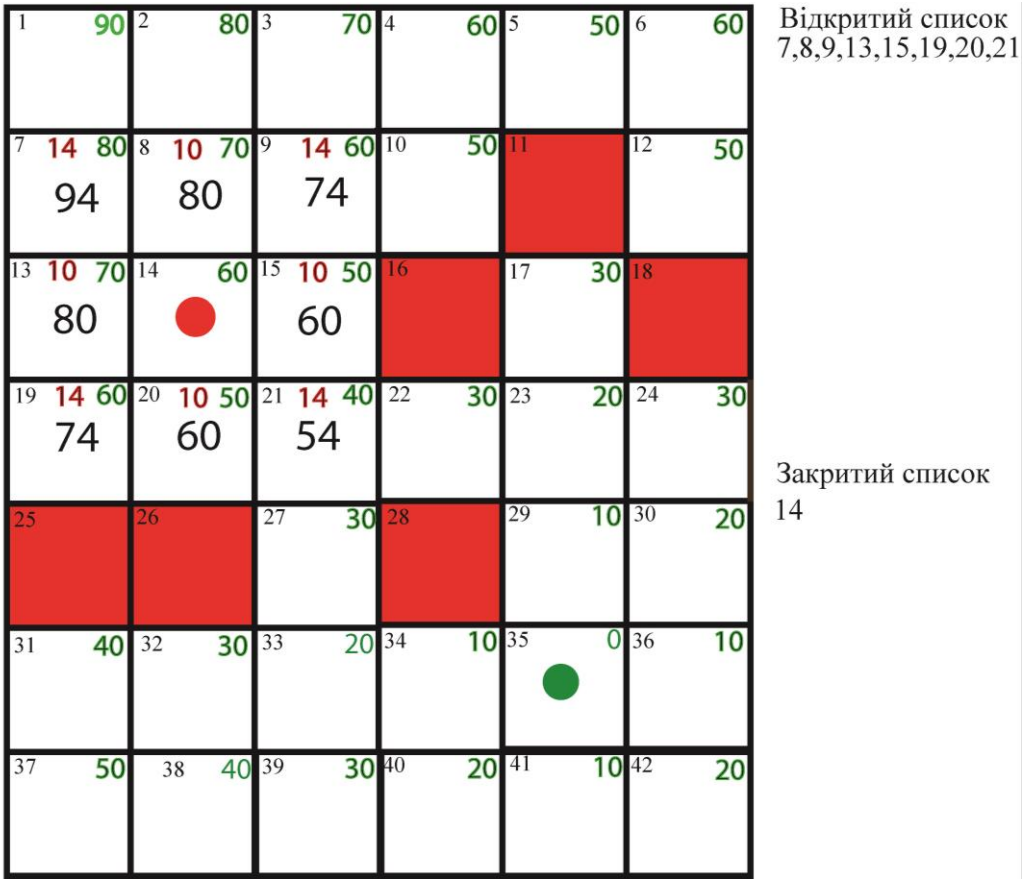

Рис.7.15. В центах плиток відкритого списку, що оточують стартову позицію ігрового агента, проставлені значення функції f(n).

Надалі розглянемо спрощену задачу прокладання оптимального маршруту в ігровому полі, вважаючи, що прохідність плиток приймає лише два значення: повна прохідність ($k = 1$) – це незабарвлені плитки та нульова прохідність ($k =$ Ґ) – це забарвлені червоним кольором плитки. Для обчислення значення функції f(n) в нашому спрощеному варіанті ігрового поля скористаємось сталістю геометричних розмірів плиток (10r 10) та незмінною величиною коефіцієнта прохідності ($k = 1$). Після цього значення функції f(n) виставимо у центрі кожної плитки з відкритого списку, як показано на рис.7.15. Вважатимемо, що ігровий агент має змогу рухатись по діагоналі між прохідними плитками а також по діагоналі між непрогідними плитками. Наприклад, ігровий агент може пройти між плитками 11 та 16 (рис. 7.15). Припускаємо також, що евристичний маршрут може проходити як через прохідні плитки, так і через непрохідні. На рис. 7.12 показано два принципово відмінних маршрути, що визначають евристичну відстань: маршрут 1 проходить виключно через прохідні плитки, в той час як маршрут під номером 2 перетинає непрохідну плитку під номером 11.

Аналіз значень функції f(n) дозволяє встановити її мінімальну величину, рівну 54 (рис.7.15). Наступний крок полягає у розширенні закритого списку плиток, вибираючи плитки із відкритого списку, але із мінімальними величинами f(n). Це дозволяє приєднати до списку закритих плиток комірку з номером 21, що характеризується величиною функції f(n), рівною 54 – мінімальному значенню серед представленого спектра величин. Сусіди вказаної плитки мають номери 22 та 27. Визначимо значення функції f(n) для цих плиток та приєднаємо їх до списку відкритих плиток.

Тепер знову серед відкритих плиток встановимо плитки з мінімальними значеннями функції f(n). Це будуть саме щойно відкриті плитки, тобто плитки за номерами 22 та 27 – саме значення 54 є мінімальним серед всіх відкритих плиток (рис.7.16).

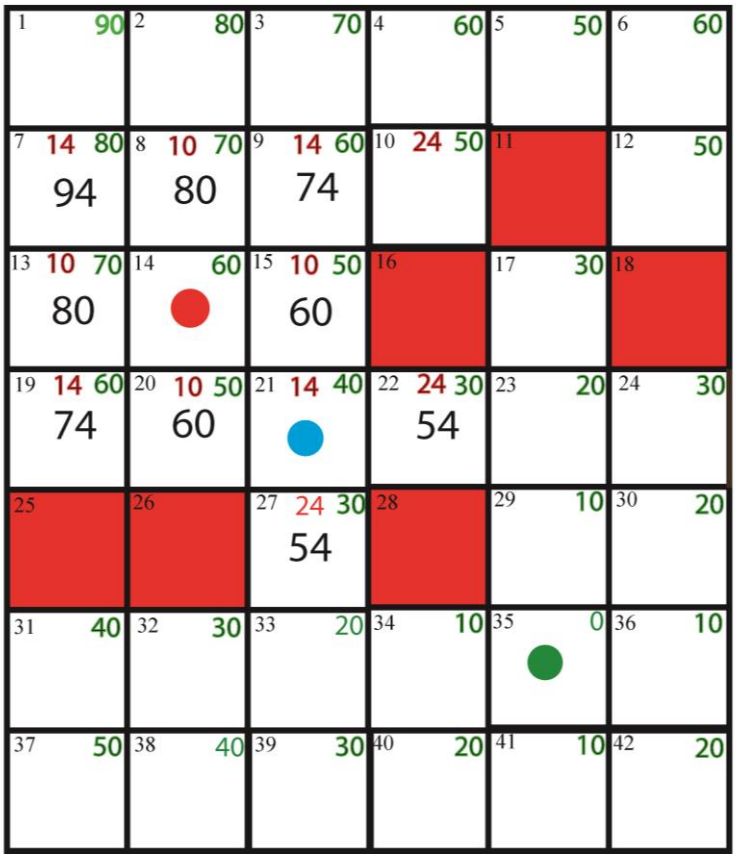

Рис. 7.16. Навколо закритих плиток виставлені значення функції f(n) = g(n) + h(n) та величини її складових – g(n) та h(n). Значення функції f(n) приведені в центрі плитки, а значення величин g(n) та h(n) – відповідно вверху в центрі та вверху справа.

Розширимо спектр закритих плиток, так само як і відкритих (рис. 7.16). Відповідні дані занесемо у списки, представлені справа від ігрового поля. В результаті отримаємо 4 закриті плитки та до списку відкритих плиток додамо 6 плиток. Як і у попередніх варіантах, проведемо аналіз значень функції f(n) для відкритих плиток. Коли маємо справу з невеликим масивом відкритих плиток, то таку сепарацію плиток по значенню функції f(n) провести просто. Але у випадку великого масиву краще названу операцію доручити комп'ютерній програмі, яка виставляє величини у порядку їх зростання. Приведемо код цієї програми.

```
package sorting;
import java.util.*;
/**
 * Created by pk on 12.07.2018.
 */
public  class SortDemo {
public static void main(String[] args) {
int[] a = {94,80,74,80,60,68, 74,60,54,48,68,54,48};
Arrays.sort(a);
for (int n : a) {
System.out.print( "   " +n);
      }
```

```
    }
}
```

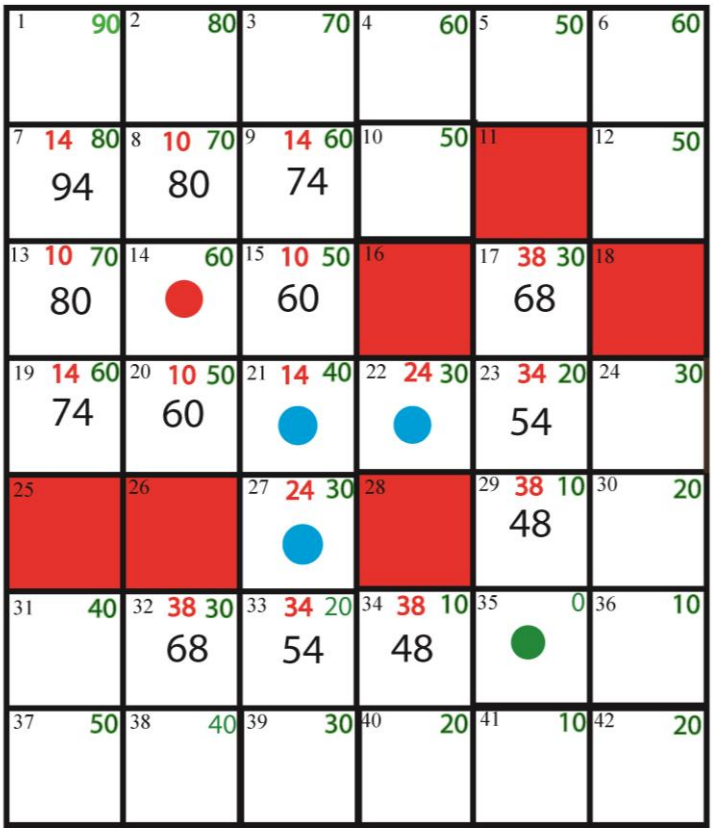

Рис.7.17. Закритий список складається з чотирьох плиток, навколо яких розташовуються плитки із відкритого списку.

В результаті роботи програми отримуємо відсортований список: 48 48 54 54 60 60 68 68 74 74 80 80 94. Із приведеного списку вибираємо мінімальне/мінімальні значення функції f(n) – одне або кілька – та заносимо ці дані у закритий список і паралельно оновлюємо відкритий список. Для нового відкритого списку повторюємо процедуру ранжування поки не дійдемо до фінішної плитки. Така процедура, повторена з допомогою цикла for, дозволить досить швидко прокласти оптимальний маршрут, якщо мова йде про одноагентну ігрову ситуацію. У випадку багатоагентної ігрової ситуації, коли необхідно прокласти оптимальні маршрути кільком ігровим агентам, необхідно скористатись алгоритмом Флойда, який застосовується для прокладання оптимальних маршрутів між всіма вершинами графа.

На рис. 7.18 приведено остаточний результат прокладання маршруту у випадку одного ігрового агента. Звернемо увагу, що у плитці 21 маршрут розділяється на дві рівновеликі вітки – A та B. Кожен із цих двох маршрутів має однакову мінімальну вагу, рівну 48 умовним одиницям довжини.

В своїй роботі “Pathfinding in Games” Adi Botea із співавторами приводить наступні висновки, які в перекладі звучать так: «Комерційні ігри можуть бути чудовим тестовим майданчиком для дослідження штучного інтелекту, який є середнім між синтетичними, дуже абстрактними академічними тестами та більш складними проблемами з реального життя. Серед численних технологій штучного інтелекту та проблем, пов'язаних з іграми, такими як навчання, планування та налаштування мов спілкування, прокладання шляху в ігровому полі являється одним із найпоширеніших застосувань штучного інтелекту для ігор… Почнемо із указання популярного, базового підходу до пошуку шляхів. Три основні елементи такого підходу – представлення ігрового поля у вигляді графа, пошук алгоритму та евристичної функції для здійснення пошуку. Ігрові поля у вигляді решітки є популярним способом представлення ігрового поля в пошуковому

графі. При такому підході ігрове поле розділяється на квадратні клітинки, які часто називаються плитками. Залежно від стратегії гри деякі ігрові плитки позначаються як прохідні, інші як заблоковані або непрохідні. Ігровий агент може займати лише

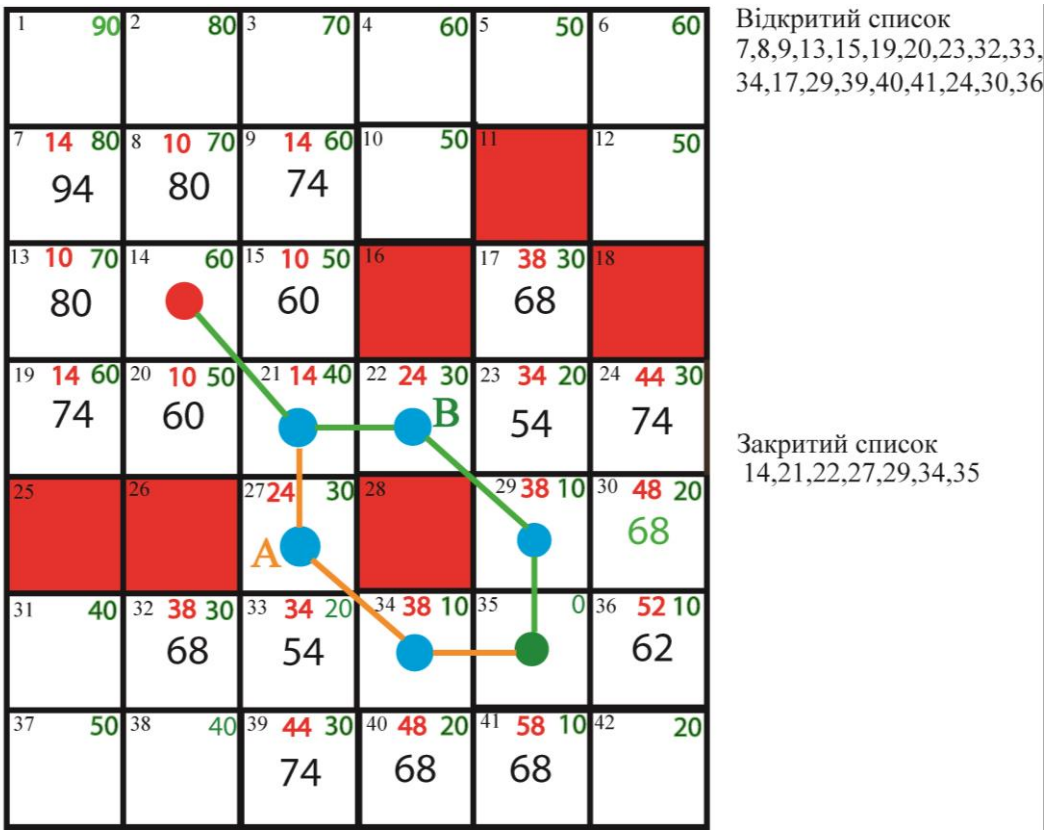

Рис. 7.18. Показані два оптимальні рівноцінні маршрути A та B, довжина кожного з яких складає 48 одиниць шляху.

одну прохідну плитку на певний момент часу. Прохідні плитки стають вузлами в графі пошуку. Ребрами графа стають лінії, що з'єднують суміжні прохідні плитки. Залежно від типу ігрового поля, можна визначити або 4 напрямки руху ігрового агента (вверх-вниз та вправо-вліво) або 8 напрямків (до вказаних напрямків додаються ще напрямки по діагоналях плиток) ».

Найбільш прийнятним варіантом алгоритму, що знаходить оптимальний шлях в ігровому полі, є A*-алгоритм. Вище ми вже розглядали конкретний приклад використання цього алгоритму. Принциповим питанням, як вже підкреслювалось, є питання вибору евристичної функції h(n). Важливо також співставити ігрове поле, складене з плиток, з графом. В цьому відношенні вважатимемо, що центр плитки (tile) являє собою вузол графа, а проміжок між центрами сусідніх плиток – ребро графа. Таким чином, на доступні плитки ігрового поля накладаємо граф (рис. 7.19). Тепер можна скористатись алгоритмом пошуку оптимального маршруту з допомогою програми **package IC-4.** Саме цей алгоритм знаходить оптимальний шлях між двома вибраними вершинами графа, витрачаючи при цьому мінімальні обчислювальні ресурси. В ігровому варіанті використання A*-алгоритму вузли графа можуть представляти собою комірки 2D-решітки ігрового поля. Тут слід зауважити, що графи, на яких використовується A*-алгоритм, можуть мати ребра різного роду, наприклад, ненаправлені, однонаправлені, або двонаправлені. В нашій задачі вважатимемо, що маємо справу з навантаженим неорієнтованим графом. Ваги ребер вздовж горизонталі та вертикалі рівні і складають величину 10. Ваги ребер між сусідніми плитками по горизонталі визначаються як $10 \cdot \sqrt{2} \approx 14$. Тепер можна ввести дані про граф, представлений на рис. 7.19, у код програми **package IC-4.** Треба лише у цій програмі перевизначити номера вузлів та їх евристичні відстані, як це зроблено у лістингу **package IC-5.**

```
package IC-5;
import java.util.PriorityQueue;
import java.util.HashSet;
import java.util.Set;
import java.util.List;
import java.util.Comparator;
import java.util.ArrayList;
import java.util.Collections;
public class AstarSearchAlgo {
public static void main(String[] args) {
//вводимо евристичні відстані від кожної з вершин графа до кінцевої
//вершини  28
        Node n1 = new Node("1", 90);
        Node n2 = new Node("2", 80);
        Node n3 = new Node("3", 70);
        Node n4 = new Node("4", 60);
        Node n5 = new Node("5", 50);
        Node n6 = new Node("6", 60);
        Node n7 = new Node("7", 80);
        Node n8 = new Node("8", 70);
        Node n9 = new Node("9", 60);
        Node n10 = new Node("10",50);
        Node n11 = new Node("11", 50);
        Node n12 = new Node("12", 70);
        Node n13 = new Node("13", 60);
        Node n14 = new Node("14", 50);
        Node n15 = new Node("15", 30);
        Node n16 = new Node("16", 60);
        Node n17 = new Node("17", 50);
        Node n18 = new Node("18", 40);
        Node n19 = new Node("19", 30);
        Node n20 = new Node("20", 20);
        Node n21 = new Node("21", 30);
        Node n22 = new Node("22", 30);
        Node n23 = new Node("23", 10);
        Node n24 = new Node("24", 20);
        Node n25 = new Node("25", 40);
        Node n26 = new Node("26",30);
        Node n27 = new Node("27", 20);
        Node n28 = new Node("28", 0);
        Node n29 = new Node("29", 10);
        Node n30 = new Node("30", 50);
        Node n31 = new Node("31", 40);
        Node n32 = new Node("32", 30);
        Node n33 = new Node("33", 20);
        Node n34 = new Node("34", 10);
        Node n35 = new Node("35", 20);
```

В результаті виконання програми **package IC-4** з урахуванням коду **package IC-5** отримуємо маршрут $14 \to 21 \to 27 \to 34 \to 35$. Цей маршрут співпадає із маршрутом А, що був отриманий у випадку виконання словесного алгоритму, результати роботи якого приведені на рис.7.18.

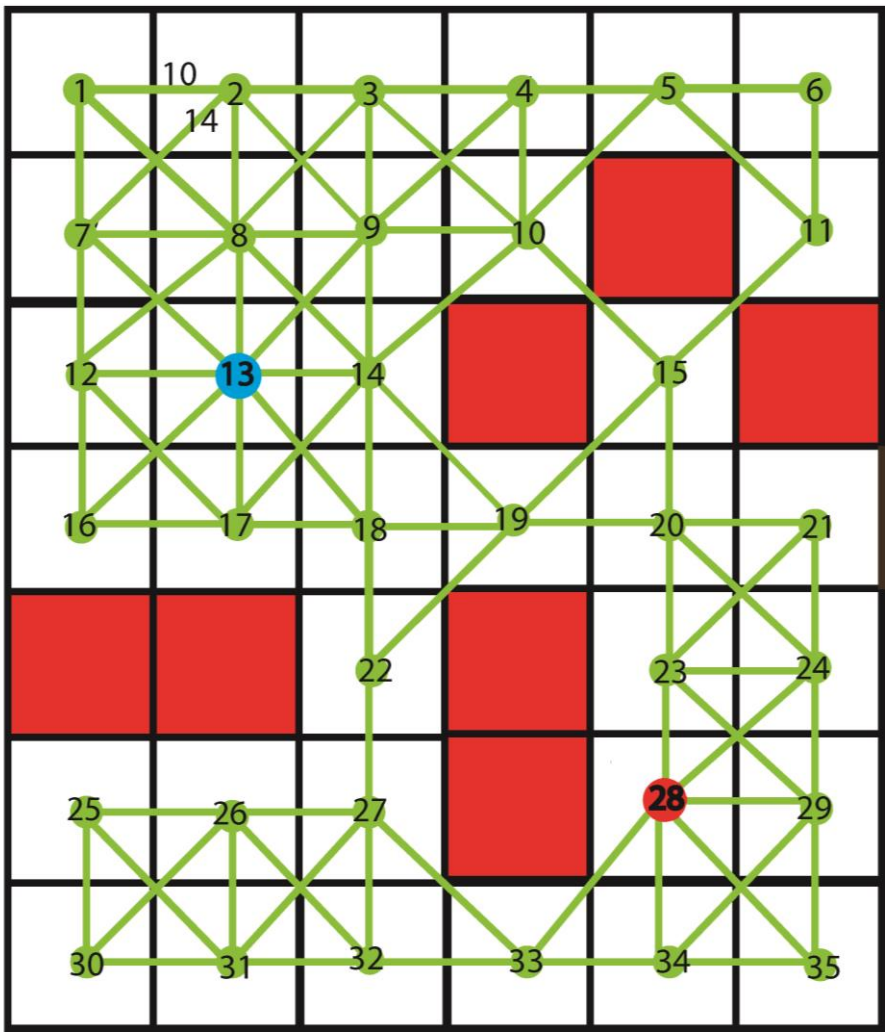

Рис.7.19. На відкриті плитки ігрового поля накладено непланаринй зважений граф. Вага кожного ребра між сусідніми вершинами по горизонталі та вертикалі дорівнює 10, а по діагоналі – 14 (показані ваги лише для двох ребер – інші аналогічні ребра мають такі ж ваги). Стартова позиція (вершина 13) та фінішна (вершина 28) виділені.

Тема 8. Прикладні аспекти програмування на Java

Проблема оптимізації міського трафіку

Важливою задачею програмування є створення прикладних програм, які вирішують конкретні проблеми. В цьому відношенні широкий спектр проблем виникає при оптимізації різних видів трафіку. Що таке трафік? В перекладі з англійської мови traffic означає потік транспорту. В широкому розумінні трафік може означати: 1) інтернет-трафік – об'єм мегабайт, що передаються на смартфон, планшет чи комп'ютер. Пакети такого виду трафіка включаються у тарифи сотових операторів; 2) мобільний трафік – протяжність в хвилинах сотового зв'язку, що споживається абонентом; 3) трафік відвідувачів сайту – число осіб, що зайшли на сайт; 4) реферальний трафік – це кількість переходів на певний сайт по посиланням, що знаходяться на інших інтернет-ресурсах. Часто кажуть веб-трафік, мультимедіа трафік, трафік вантажних/пасажирських перевезень тощо. Отже, в широкому розумінні трафік – це інтенсивність руху, транспортування, потік даних через комунікаційну систему чи мережу. Можна, наприклад, сказати «нічний трафік електроенергії», маючи при цьому на увазі передачу електроенергії в нічний час, коли вартість її споживання зменшується вдвічі по відношенню до денного часу.

З урахуванням сказаного, розглядатимемо трафік як потік транспортних засобів (ТЗ) по міських магістралях. Справа в тому, що на сьогодні міські транспортні мережі неспроможні здійснювати ефективний трафік ТЗ. Наслідком цієї неспроможності є транспортні затори у великих та і в не дуже великих містах. Проблема заторів далеко не нова. Ось що писав у своєму романі «Здобич» знаменитий французький письменник Еміль Золя: «По дорозі додому, серед скупчення екіпажів, які поверталися берегом озера, їхати можна було тільки ступою. Нарешті коляска потрапила в такий затор, що довелося навіть зупинитися». Як бачимо, затори були і тоді, коли ще не було автомобілів. З тих пір проблема набула набагато більш гострішого характеру.

Вирішення проблеми трафіку в мегаполісах є актуальною проблемою всесвітнього характеру. А найголовнішою проблемою в цьому відношенні є затори. Як уникнути цього негативного явища? Як організувати проїзд кожного окремого ТЗ (а таких об'єктів у великому місті може бути понад мільйон) до заявленого водієм пункту призначення з таким розрахунком, щоб поїздка зайняла найменший час? Технологія, що пропонується, дозволяє ефективно вирішувати поставлені задачі.

Заявлені проблеми «стоїть на порядку денному» багатьох крупних міжнародних компаній, що розробляють технічні засоби організації дорожнього руху. Найбільш близькою системою до розглядуваної нами технології навігації є система навігації типу GPS/GLONASS. Кожного року GPS-навігація модифікується. Деяке автомобільне навігаційне обладнання може повідомляти про затори на вулицях міста та пропонувати альтернативний маршрут об'їзду таких місць. Інтеграція з автомобілем стає глибшою, і це дозволяє задавати кожному конкретному водієві голосові маршрут-команди. Особливе значення відіграє взаємодія між автомобілем та дорожньою інфраструктурою. Така взаємодія здійснюється при допомозі так званих точок доступу (access points), розташованих вздовж автомобільної дороги та на перехрестях. Взагалі, в основі технології регулювання трафіку лежать чотири складових: світлофори, детектори черги (queue detectors), дорожні відеокамери і центральна контрольна система (central control system).

Пропонована авторами технологія спрямована на реєстрацію ТЗ, що проїжджають перехрестя. При цьому кожні 10 секунд система знімає дані із спеціальних дорожніх контрольних датчиків та оновлює базу даних, яка використовується для прокладання оптимальних маршрутів а також для коригування протяжності фаз горіння світлофорів у відповідності із завантаженістю напрямків на перехресті. Завдяки цьому досягається оптимізація трафіку.

Важливою проблемою є розташування датчиків, оскільки їх локалізація суттєвим чином впливає на те, які транспортні потоки реєструються і тому можуть бути керованими. На сьогодні застосовується модифікована модель стільникових автоматів (modified cellular automata model) для вивчення процесів взаємодії між автомобілями. Здійснюється широкомасштабне моделювання потоків ТЗ. Часто застосовується «розумна» мережева імітаційна модель, що використовує в якості об'єкта дослідження центральні райони мегаполісів – їх найбільш завантажені транспортні артерії. У таких випадках на карті міст зображаються автомагістралі, звичайні дороги, автобусні

зупинки, комерційні зони, перехрестя та автомобільні розв'язки. Для проведення імітаційних досліджень часто застосовується імітаційна модель PARAMICS. З метою реєстрації потоків ТЗ в деяких випадках використовуються петлеві детектори (loop detectors) та камери спостереження (surveillance cameras). Можливим варіантом покращення трафіка є координоване регулювання автомобільних потоків за допомогою бездротового зв'язку між автомобілями. З розвитком автотранспорту все більшого поширення набуває теорія стільникових автоматів. Ефективним засобом моделювання руху через перехрестя є платформа NetLogo, що є багатоагентним програмним середовищем для моделювання різних динамічних процесів. Тут також слід згадати програму інтерактивного візуального моделювання AnyLogic, яка дозволяє оцінити ситуацію з трафіком у великому місті візуально.

Широкомасштабні дослідження проводяться провідними автомобільними компаніями в плані інтелектуальної системи керування режимом роботи світлофорів через двосторонній обмін інформацією з центральним пунктом керування трафіком (ЦПКТ). Перехрестя використовується переважно для перетворення інформації світлофорів на бездротові сигнали та реалізації інтелектуального управління світлофорною системою. ЦПКТ здійснює управління міськими дорогами та надає інформацію водіям автомобілів а також передає електронні карти маршрутів. Інтелектуальна система управління роботою світлофорів на базі інтелектуального терміналу дозволяє водіям своєчасно отримувати інформацію про перемикання світлофорів. Система світлофорів автоматично визначає кількість автомобілів на дорогах, щоб миттєво змінювати час проїзду для автомобілів різних напрямків. Таким чином досягається оптимізація маршрутів руху кожного ТЗ.

Важливим питанням проблеми трафіку являється проблема паркінгу у великому місті. Ефективна технологія в цьому відношенні запропонована компанією Siemens. Система контролює завантаженість вулиць і передає інформацію автомобілістам, використовуючи для кожної окремої стоянки інформацію, зчитувану за допомогою наземних датчиків або на основі кількості проданих паркувальних дозволів. В обох випадках система направляє водіїв безпосередньо на наявні місця для паркування, що запобігає переповненню вулиць та зменшує навантаження на трафік. Має місце інтеграція в загальну систему керування трафіком. Це дає змогу використовувати бази даних паркування для надання автомобілістам рекомендацій щодо маршрутизації вже при в'їзді в межі міста.

Масштабні дослідження проблем трафіку проводяться по всьому світу, Так, в штаті Юта (США) у розгляд введена система показників ефективності регулювання руху потоків ТЗ на основі аналізу даних, отриманих із спеціальних мікрохвильових датчиків.

Зауважимо, що для вирішення проблеми дорожнього руху використовують різні наукові підходи, зокрема, динаміку рідин. При цьому потоки рідин в трубах асоціюються з потоками ТЗ у містах. Такі модельні підходи досить ефективні та дозволяють в певній мірі вирішити проблеми, пов'язані із дорожнім трафіком.

Важливою проблемою трафіку є питання, присвячене взаємодії приватних автомобілів та громадського транспорту. Для розв'язання проблем такої взаємодії вводиться поняття двотипних (бімодальних) міських мереж (bi-modal urban networks). Для організації ефективної взаємодії вказаних типів ТЗ вводиться у розгляд бімодальна Макроскопічна Фундаментальна Діаграма (МФД), що моделює змішаний трафік ТЗ згаданих видів. Результати показують, що запропонована технологія може значно: (i) зменшити затори в мережі; (ii) поліпшити показники трафіку автобусів з точки зору часу проїзду маршруту руху; (iii) знизити рівень скупченості ТЗ на критичних ділянках транспортної мережі. В якості об'єкта досліджень вибрана транспортна мережа Сан-Франціско.

Загалом, координована взаємодія між ТЗ та ЦПКТ із зворотнім зв'язком має особливі перспективи з огляду на існуючу тенденцію поширення автомобілів з так званими системами автопілота та появою на дорогах безпілотних ТЗ, що у майбутньому дозволить вивести керування транспортними потоками на кардинально новий, більш ефективний рівень зі скоординованими діями всіх учасників руху та упередженням проблемних ситуацій.

Стратегічна мета, яку ставлять перед собою автори цієї роботи, заключається в побудові оптимальних маршрутів для кожного ТЗ та синхронізації потоків ТЗ. Ставиться задача провести кожен ТЗ по місту по оптимальному маршруту з урахуванням можливої корекції такого маршруту

на основі оновлюваної кожні 10 секунд бази даних. Тобто, система регулювання трафіка всієї сукупності автомобілів на трасах міста повинна прокладати динамічний – в режимі реального часу – і оптимальний маршрут кожному ТЗ, що замовляє лише кінцеву позицію маршруту (стартова позиція фіксується автоматично при підключенні до системи навігації). Центральний комп'ютер на ЦПКТ, використовуючи комп'ютерну програму, записану знизу, співпрацює з кожним водієм та передає йому голосові команди щодо маршруту руху до заявленого водієм пункту призначення як при звичайній GPS-навігації. Особливість полягає у тому, що програма аналізує динамічну ситуацію на кожному перехресті, на ділянках дороги між сусідніми перехрестями і по всьому місту та відповідно прокладає маршрут з урахуванням ситуації на кожен конкретний момент часу; при цьому використовується комп'ютерна програма, що реалізує **алгоритм Дейкстри** щодо знаходження оптимального шляху. Таким чином, остаточною метою даної технології є синхронізація транспортних потоків, оптимальне використання транспортних артерій всього міста (за рахунок перенаправлення потоків ТЗ на менш завантажені вулиці міста), запобігання утворенню заторів а також супровід кожного ТЗ до місця призначення з таким розрахунком, щоб затрачений на поїздку час був мінімальним.

Технологія являє собою автоматизовану інтелектуальну систему регуляції дорожнього руху у великих містах, яку умовно можна розділити на два етапи. На першому етапі здійснюється регулювання трафіку через одне окреме перехрестя, що взаємодіє із сусіднім, а на другому – через все місто. Розглянемо тепер детально перший етап регулювання трафіку.

Інтелектуальне перехрестя

Базовим елементом а заодно і базовою проблемою регулювання трафіку у мегаполісі є окреме перехрестя. Дійсно, в першу чергу така проблема пов'язана із заторами на перехрестях і, як наслідок, складністю проїзду кожним ТЗ по вибраному маршруту. З метою запобігання виникнення заторів, які в основному виникають на перехрестях, пропонується застосувати мережу інтелектуальних перехресть, оснащених спеціальними датчиками; причому датчики сусідніх перехресть взаємодіють між собою. Така взаємодія фактично означає, що не тільки окремі перехрестя знаходяться під контролем ЦПКТ, а і під таким контролем знаходяться ділянки дороги між сусідніми перехрестями. Таким чином з'являється можливість тримати під контролем всю транспортну мережу міста, а значить, контролювати маршрути всіх ТЗ на дорогах міста. Виходячи із всього вищесказаного головна проблема, що виникає у даному випадку, полягає у поєднанні алгоритмів регулювання руху ТЗ як через окреме перехрестя, так і по всьому місту.

Отже, базовою проблемою регулювання дорожнього руху у мегаполісі є проблема оптимізації проїзду ТЗ через кожне окреме перехрестя. Організувавши ефективний рух через такий об'єкт, досягнемо високої ефективності трафіку по всьому місту. Для організації проїзду ТЗ необхідно їх реєструвати. Причому способів реєстрації є багато. Для вирішення нашої проблеми найбільш ефективним способом такої реєстрації є спосіб на основі п'єзокристалічних датчиків, вмонтованих в полотно дороги.

Компанія International Road Dynamics Inc. розробила досить ефективну конструкцію п'єзоелектричного датчика дорожнього руху RoadTrax BL9. Такий пристрій легко монтується в дорожнє полотно і досить чутливий. При наїзді колісної пари на зону розташування датчика вихідний сигнал величиною 250 мВ цілком достатній для реєстрації. Взагалі датчики дорожнього руху по своєму типу поділяються на дві групи (рис. 8.1). До першої групи відносяться датчики, що монтуються в полотно дороги. До цих датчиків відносяться: 1) електромеханічні; 2) магнітні; 3) пневмоелектричні; 4) індуктивні та 5) п'єзоелектричні. До другої групи відносяться датчики, що монтуються над дорогою. До цієї групи відносяться: 1) акустичні; 2) інфрачервоні датчики; 3) радарні датчики; 4) відеодетекторні датчики; 5) допплерівські датчики.

Найбільш придатними для вирішення поставленої в нашій роботі задачі є датчики, що монтуються в дорожнє полотно, а серед цієї групи – п'єзокристалічні та індуктивні. Проте монтування індуктивних датчиків в полотно дороги є більш проблематичним. Ось чому доцільно зупинитись на простих, але досить ефективних п'єзоелектричних датчиках. Серед цього класу

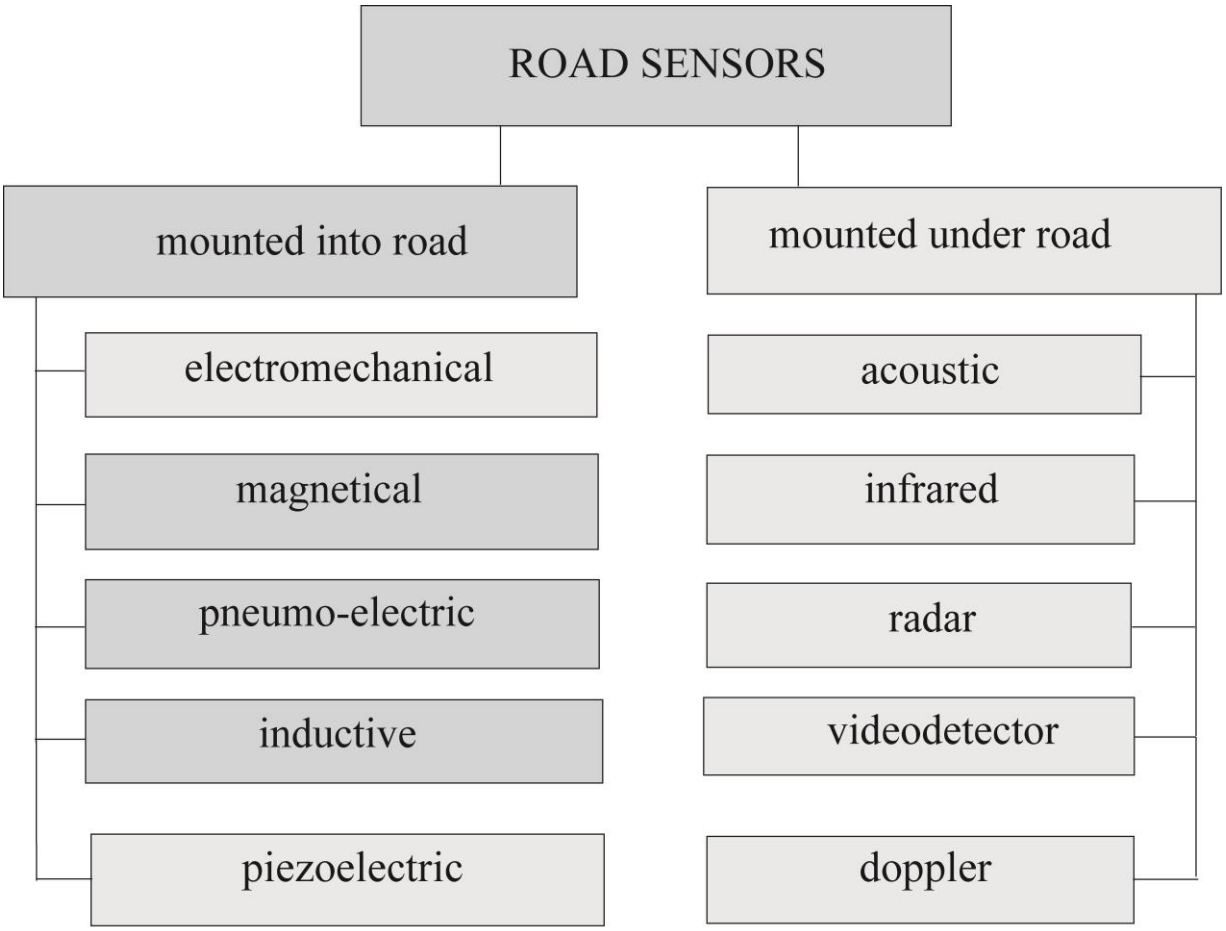

Рис.8.1. Класифікація датчиків дорожнього руху.

датчиків є сенс згадати про п'єзоелектричний датчик типу 12-DOF. Це новий високочутливий датчик, спроможний реєструвати сили в широкому діапазоні величин. Пристрій працює на частоті 11кГц з похибкою не більше 1%.

Теоретичні аспекти оптимізації трафіку на окремому перевантаженому перехресті розглядаються в так званій дискретно-часовій моделі (discrete-time model). Для хрестоподібного перехрестя математична модель, що описує рух ТЗ через таке перехрестя, представляється у вигляді системи рівнянь наступного виду:

$$J_D = \sum_{k=0}^{N} [q_1(k) + q_2(k)] + \frac{a_1 + a_2}{2} \cdot T \cdot \sum_{k=0}^{N-1} u(k) \to \min \tag{8.1}$$

$$q_1(k+1) \geq \max\{q_1(k) + d_1 \cdot T \cdot (u(k) - u_L), a_1 \cdot T \cdot (1 - u(k))\} \tag{8.2}$$

$$q_2(k+1) \geq \max\{q_2(k) + d_2 \cdot T \cdot (u_H - u(k)), 0)\} \tag{8.3}$$

$$u_{\min} \leq u(k) \leq u_{\max} \tag{8.4}$$

$$q_i(0) = q_{i,\text{int}},\ i = 1,2;\ N = 0,1,2 \ldots \text{N} - 1. \tag{8.5}$$

В приведених рівняннях фігурують такі змінні:

J_D – модельна величина, що являє собою число ТЗ на перевантаженому перехресті, які не перетнули його; N – число циклів перемикання світлофора; k – нумератор циклів; $q_1(k), q_2(k)$ – число ТЗ на перехресті в k-му циклі переключення світлофора відповідно на горизонтальному та вертикальному напрямках хрестоподібного перехрестя; a_1, a_2 – число ТЗ, що прибули за одиницю

часу відповідно на горизонтальному та вертикальному напрямках; T – цикл роботи світлофора; $u(k)$ – відношення величини фази горіння зеленого світла до аналогічної величини для червоної фази; d_1, d_2 – число ТЗ, що перетнули перехрестя за одиницю часу; $u_H = 1 - a_2 / d_2$, $u_L = a_1 / d_1$; $u_{\min}, u_{\max}$ – відповідно мінімальне та максимальне значення $u(k)$; $q_{i,\text{int}}$ – початкова величина ТЗ на перехресті для горизонтального ($i = 1$) та вертикального напрямків ($i = 2$).

Цільова функція системи рівнянь (8.1) – (8.5) представляється виразом (8.1) і являє собою число ТЗ, що не перетнули перехрестя. Зрозуміло, що таку величину треба мінімізувати. Шуканими змінними є величини $u(k)$. Рівняння (8.2) і (8.3) представляють еволюцію з часом обох автомобільних черг. Рівняння (8.4) дає верхню і нижню границі $u(k)$ в кожному циклі. Рівняння (8.5) представляє стартові (початкові) довжини автомобільних черг на перехресті. Загалом система рівнянь (8.1) – (8.5) являє собою задачу математичного програмування. Графічний розв'язок цієї задачі представлений на рис.8.2.

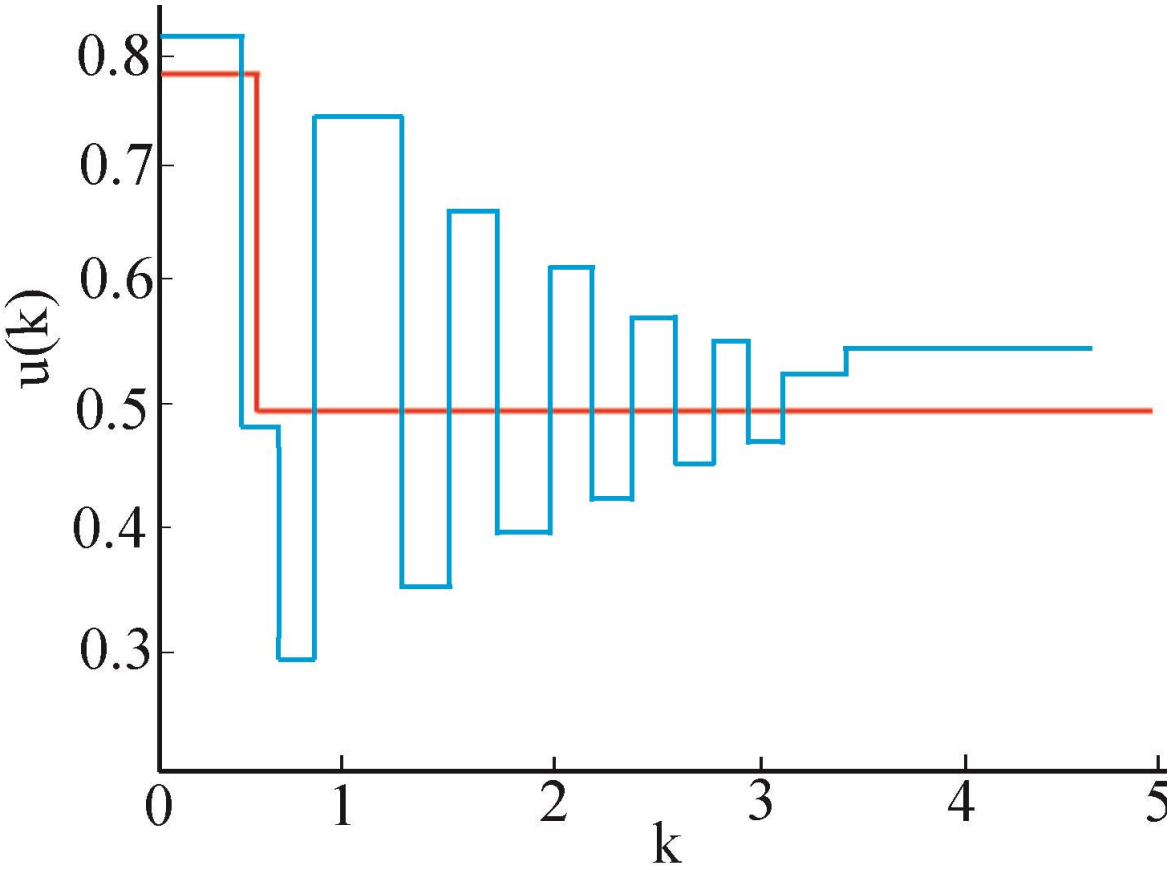

Рис. 8.2. Розв'язок системи рівнянь (8.1)–(8.5): зигзагоподібна лінія – точний розв'язок; двохступінчата – наближений.

Отриманий результат носить скоріше академічний, а не практичний характер та не дозволяє використати отримані результати для практичної мети – інтелектуальної регуляції трафіку через перехрестя з метою покращення його пропускної здатності.

Широкомасштабне дослідження методів об'єднаної інформації, отримуваної із різноманітних дорожніх датчиків, таких як детектори, відеокамери і радари досліджувалось в багатьох роботах. Особливу увагу привертає технологія зв'язаних ТЗ (connected vehicles), яка дає можливість збирати та аналізувати зв'язки типу «ТЗ – Інфраструктура» (V2I) та «ТЗ – ТЗ» (V2V), які дають можливість зменшити імовірність заторів, збільшити безпеку руху та зменшити витрати палива.

На рис.8.3 зображена схема взаємодії автомобілів, що проїжджають перехрестя, із придорожньою інфраструктурою. Система передачі даних OBU забезпечує V2X комунікацію, тобто взаємодію автомобілів з придорожніми ячейками – Road Side Unit (RSU).

Організація приведеної на рис. 8.3 взаємодії між автомобілями та придорожньою інфраструктурою не достатньо ефективна та вимагає складної технічної організації і значних фінансових витрат. В зв'язку з цим автори даної роботи пропонують більш просту, але набагато більш ефективну систему організації дорожнього руху. В цьому випадку проблема базується на наступних складових: 1) з аналітичної сторони необхідно створити алгоритм та комп'ютерну програму, що сумісно забезпечують ефективне перемикання світлофорних фаз у відповідності із завантаженістю дорожніх напрямків; 2) з технічного боку реєстрація ТЗ кожному перехресті здійснюється з допомогою датчиків, які фіксують число колісних пар, пропорційне числу ТЗ, що або в'їжджають на проїзну частину дороги між сусідніми перехрестями (так звані вхідні датчики) або виїжджають із цієї частини дороги – вихідні датчики.

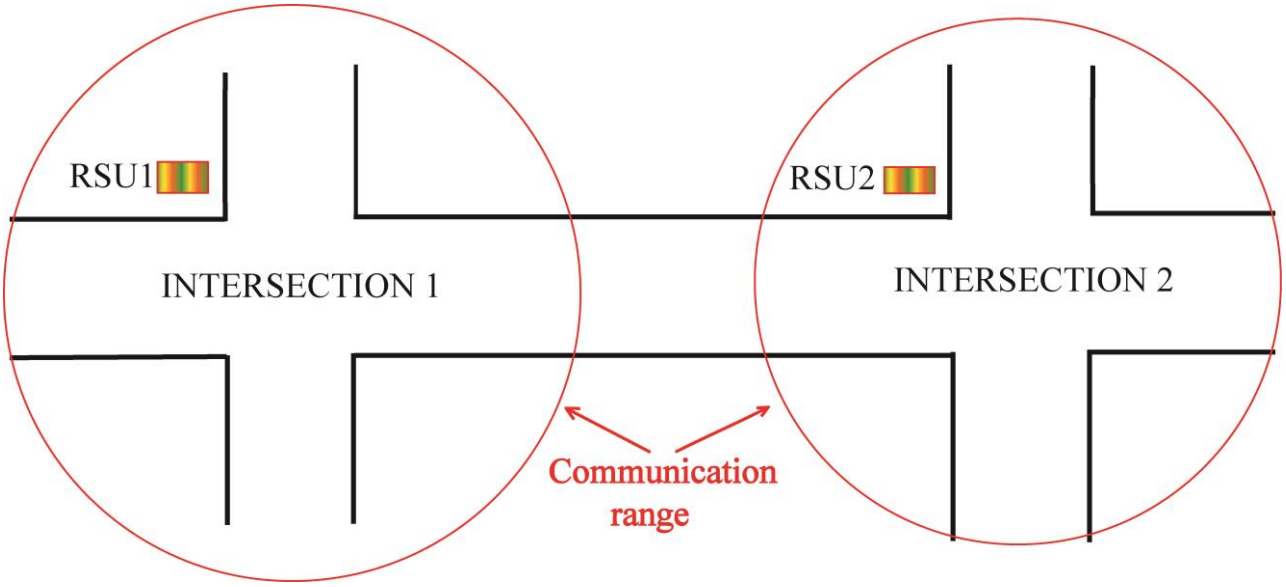

Рис.8.3. Технологія зв'язаних автомобілів, де RSU – Road Side Unit, тобто придорожня ячейка, яка взаємодіє як із світлофором, так і з автомобілем. Діапазон взаємодії RSU зображений у вигляді колової траєкторії. На автомобілях встановлена система передачі даних OBU (On Board Unit).

Реєстрація ТЗ, перетинаючих перехрестя, організована наступним чином (рис.8.4). Кожне окреме перехрестя обладнане п'єзокристалічними датчиками типу RoadTrax BL9 . Ці датчики монтуються перпендикулярно до поздовжньої вісі дороги на відстані 10 м після стоп-лінії ближче до центру перехрестя. Це дозволяє реєструвати автомобілі, що залишили ділянку дороги одного напрямку руху між сусідніми перехрестями. Колісна пара автомобіля створює тиск на полотно дороги, в якому на глибині близько 5 см вмонтована п'єзокристалічна смужка. Стиск полотна приводить до стиску смужки, на поверхні якої з'являються електричні заряди завдяки явищу п'єзоефекту. Іншими словами, виникають електричні імпульси, які реєструються. Власне при проїзді ТЗ через зону розташування датчика реєструються колісні пари, що є більш точним показником завантаженості транспортної магістралі, чим просто число ТЗ. Адже, наприклад, ТЗ великої протяжності (long vehicle) можуть мати 5-7 колісних пар. Саме число колісних пар є більш точним критерієм завантаженості трафіку в порівнянні з числом автомобілів (або ТЗ).

В основі всієї технології регулювання трафіку як вихідна комірка слугує окреме регульоване перехрестя. Вважатимемо, що цикл роботи світлофорів на перехресті – це величина, що, як правило, складається з таких складових

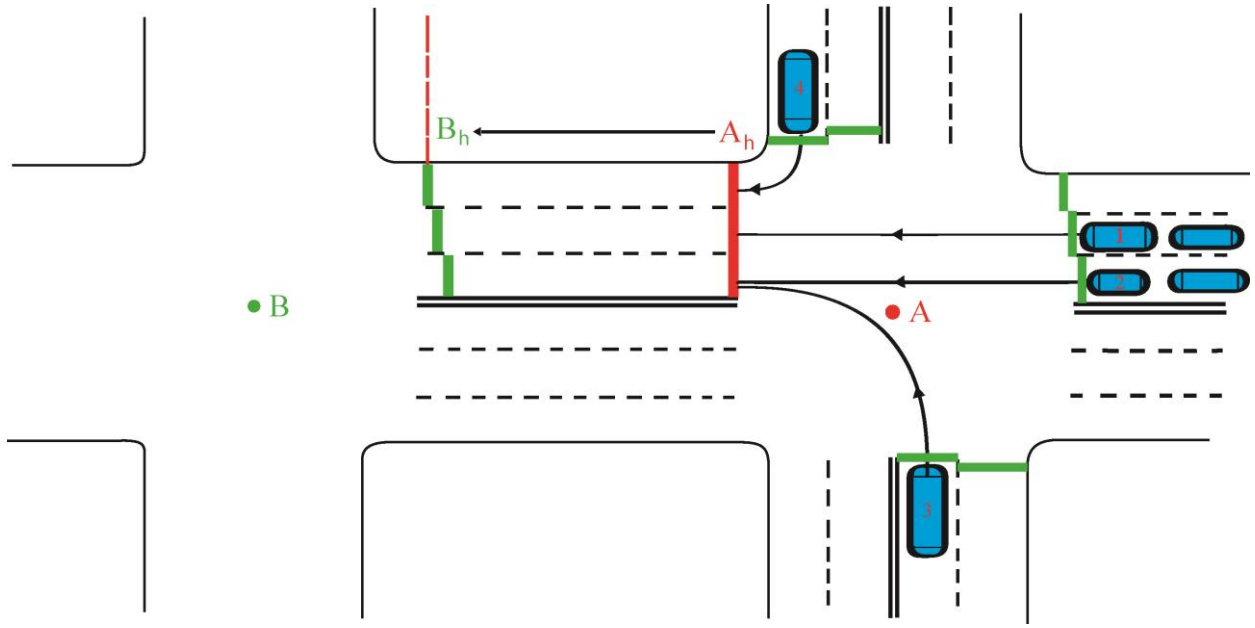

Рис.8.4. Зображено два сусідніх перехрестя A та B. Скругленими прямокутниками зображені ТЗ, маршрути яких пролягають в напрямку $A_h \to B_h$. Показані також вхідні датчики (прямокутна смужка, позначена A_h) та вихідні датчики (три рядом розташовані смужки, позначені B_h). Вхідні та вихідні датчики з'єднані з дорожним контролером (на рисунку зображений лише один сигнальний провід, що виходить від одного із датчиків B_h – це пунктирна лінія).

$$T = trh + tgh + tyh + tp, \tag{8.6}$$

де trh – час горіння червоного світла у горизонтальному напрямку;
tgh – час горіння зеленого світла у горизонтальному напрямку;
tyh – час горіння жовтого світла також у горизонтальному напрямку;
tp – час горіння зеленого світла для пішоходів.

У цьому співвідношенні (8.6) цикл роботи світлофора формується із часових інтервалів, взятих по відношенню до горизонтального напрямку. Таке саме співвідношення можна записати по відношенню до вертикального напрямку перехрестя, тобто

$$T = trv + tgv + tyv + tp, \tag{8.7}$$

де trv – протяжність червоної фази у вертикальному напрямку;
tgv – час горіння зеленого світла у вертикальному напрямку;
tyv – час горіння жовтого світла так само у вертикальному напрямку;
tp – час горіння зеленого світла для пішоходів.

З використанням приведених змінних програму регуляції трафіку на окремому перехресті можна записати у наступному вигляді:

```
package IC-6;
import java.util.Random;
import static java.lang.StrictMath.abs;
interface Lights{
  int REDH =0;
  int YELLOWH =1;
  int GREENH =2;
  int REDV =3;
```

```
int YELLOWV =4;
int GREENV =5;
int GREENP = 6;
int ERROR = -1;
  }
class T implements Lights {
private int delay;
private static int light = REDH;
T(int sec) {
delay = 1000 * sec;
}
public int shift() {
int count = (light++) % 7;
                    try {
                    switch (count) {
                    case REDH:
                        Thread.sleep(delay);break;
                    case YELLOWH:
                        Thread.sleep(delay / 3);break;
                    case GREENH:
                        Thread.sleep(delay / 2);break;
                    case REDV:
                        Thread.sleep(delay);break;
                    case YELLOWV:
                        Thread.sleep(delay / 3);break;
                    case GREENV:
                        Thread.sleep(delay / 2);break;
                    case GREENP:
                        Thread.sleep(delay );break;}}
                    catch (Exception e) {
                    return ERROR;}
                    return count;}}
                    class TrafficRegulator {
      static int T = 96;
      private static IC.T t = new IC.T(1);
      //запускаємо генератори випадкових величин, які
      //імітують завантаженість автомобілями кожного
      //із чотирьох напрямків на перехресті
      static Random gn1 = new Random();
      static Random gn2 = new Random();
      static Random gn3 = new Random();
      static Random gn4 = new Random();
                    public static void main(String[] args) {
                    double k =abs ((gn1.nextDouble()  + gn2.nextDouble() ) /
                    (gn3.nextDouble() + gn4.nextDouble()));
                    double tg = 35;
                    double tp = 23;
                    double tyh = 2;
                    double tgh = k * tg;
                    double trh = (T - tyh - tgh- tp);
                    double tgv = abs(2*tg -tgh);
                    double trv = (T - trh);
                    double tgp = 23;
                    int tyv = 2;
                    for (int j = 0; j < 7; j++)
                    switch (t.shift()) {
                    case Lights.REDH:
   System.out.println("red horizontal!");
   System.out.format("%.1f%n",trh );break;
   case Lights.YELLOWH:
   System.out.println("yellow horizontal!");
   System.out.println(tyh);break;
   case Lights.GREENH:
   System.out.println("green horizontal!");
   System.out.format("%.1f%n",tgh);break;
```

```
case Lights.REDV:
System.out.println("red vertical!");
System.out.format("%.1f%n",trv);break;
case Lights.YELLOWV:
System.out.println("yellow vertical!");
System.out.println(tyv);break;
case Lights.GREENV:
System.out.println("green vertical!");
System.out.format("%.1f%n",tgv);break;
case Lights.GREENP:
System.out.println("green pedestrian!");
System.out.println(tgp);break;
case Lights.ERROR:
System.out.println("Time error!");break;
default:
System.err.println("Unknown light.");
return;
}}}
```

В результаті роботи програми при першому запуску отримуємо такі результати:
red horizontal! – 29,6; yellow horizontal! – 2.0; green horizontal! – 41,4; red vertical! – 66,4; yellow vertical!– 2; green vertical! – 28,6; green pedestrian! – 23.0. Process finished with exit code 0.

Наступний запуск програми видасть нові результати:
red horizontal! – 49,6; yellow horizontal! – 2.0; green horizontal! – 21,4; red vertical!– 46,4; yellow vertical! – 2; green vertical! – 48,6; green pedestrian! – 23.0. Process finished with exit code 0

Таким чином, програма кожен раз видаватиме нові результати в залежності від даних, що поступають від генераторів випадкових величин. Саме з метою імітації реальної ситуації на перехресті в програмі включено чотири генератори випадкових величин, що імітують завантаженість перехрестя автомобілями: `static Random gn = new Random();`. Кожен з цих генераторів задає число, що відповідає завантаженості автомобілями певного напрямку на перехресті. Власне, нас цікавить відношення числа автомобілів, що розташовані на горизонтальному та вертикальному напрямках. В програмі таке відношення задається величиною коефіцієнта k. Варіація величини k імітує якраз зміни навантаженостей ТЗ. Тому спектр чисел на виході програми щораз змінюється випадковим чином. Змінюються, зокрема, величини, що задають час горіння зеленого світла у горизонтальному та вертикальному напрямках. В якості констант вибираються величини типу REDH – червоне світло в горизонтальному напрямку. Принципово те, що програма реагує на завантаженість ТЗ напрямків перехрестя: чим більше завантажений напрямок – горизонтальний чи вертикальний – тим більше горітиме зелене світло у відповідному напрямку, звичайно в межах періоду перемикання світлофора, який в даному разі вибраний рівним 96 с.

Отже, презентована програма дає можливість суттєво покращити пропускну здатність перехрестя завдяки «розумному» режиму його роботи в плані кореляції між протяжністю горіння різних фаз та завантаженісю напрямків на перехресті.

Оптимізація міського трафіку: алгоритм Дейкстри

Перехрестя оснащається двома типами датчиків – вхідними та вихідними (рис.8.4). Число вихідних датчиків залежить від числа смуг проїзної частини дороги одного напрямку (на рис. 8.4 таких датчиків три у відповідності з числом смуг). Отже, кожна смуга оснащається власним вихідним датчиком. Для чого? Справа в тому, що автомобілі розташовуються по смугах у відповідності із маршрутом руху кожного з них. Наприклад, якщо автомобіль розташований у смузі, що примикає до осьової лінії дороги, то водій цього ТЗ може рухатись прямо, наліво або здійснювати розворот на перехресті, але ні в якому разі не повертати направо: такий маневр заборонений провилами дорожнього руху (ПДР). Таким чиним, завдяки сумісній роботі вхідних та вихідних датчиків, є можливість реєструвати ТЗ, що в'їжджають на ділянку дороги між сусідніми перехрестями. Мається на увазі, що вхідний датчик A_h працює сумісно із комплектом вихідних

датчиків B_h і завдяки цьому кожна смуга ділянки дороги одного напрямку $A_h \to B_h$ знаходиться під контролем, що дає можливість визначити як завантаженість ділянки дороги $A_h \to B_h$ в цілому, так і завантаженість кожної окремої смуги на цій ділянці. Власне, будь-який маршрут складається із певної сукупності смуг на ділянках дороги одного напрямку, з'єднаних перехрестями. Отже, вся транспортна мережа міста знаходиться під контролем. Вираз «під контролем» означає, що є можливість визначати як динаміку руху ТЗ на кожній ділянці дороги в цілому, так і на кожній окремій смузі руху цієї ділянки дороги. Отже, можна говорити про пропускну здатність як вибраної ділянки дороги $A_h \to B_h$, так і окремої смуги на цій ділянці. Мовою теорії графів, можна визначати вагу кожного ребра та всього маршруту в цілому. Завдання полягає в тому щоб знайти маршрут найменшої ваги.

На кількісному рівні динаміка руху вздовж ділянки дороги $A_h \to B_h$ обумовлена співвідношенням між величиною ТЗ, що в'їхали за певний час (наприклад, за час, рівний циклу переключення світлофора) та виїхали із даної ділянки дороги. Технічно процес організований наступним чином: датчик A_h (вхідний датчик) реєструє автомобілі, які в'їжджають на ділянку дороги $A_h \to B_h$ з усіх можливих напрямків перехрестя А (рис.8.4). В свою чергу, комплекс вихідних датчиків на перехресті В реєструє автомобілі, що виїхали за межі цієї ділянки. Відношення цих величин якраз і свідчить про динаміку руху на конкретній ділянці дороги.

Фактично наявність вхідних та вихідних датчиків дозволяє визначати величини у лівій частині наступного виразу

$$\sum_{h=1}^{f} (N_{A_h B_h} / (\sum_{i=1}^{l} n^i_{A_h B_h}))l_{A_h B_h} \to min \qquad (8.8)$$

Тут h – індекс, що позначає смуги руху вздовж маршруту (іншими словами, ребра графа), які зчеплені між собою через перехрестя та утворюють нерозривну траєкторію, з'єднуючи початкову та кінцеву позиції маршруту. Символ i символізує собою смугу руху проїзної частини дороги одного напрямку (позначатимемо на проїзній частині одного напрямку смуги руху як a, b, c, d і т.д.). Символ l визначає число смуг руху в межах однієї ділянки дороги. Символ f означає число смуг $A_h B_h$, які формують прокладений маршрут. Для кожного заявленого маршруту програма формує із мультиплікат типу (8.8) нерозривний ланцюг, що пов'язує початкову та кінцеву позиції кожного конкретного маршруту. Отже, технологія дозволяє супроводжувати ТЗ по будь-якому міському маршруту оптимальним чином, мінімізуючи вираз (8.8).

Оскільки всі перехрестя міста а також ділянки дороги між перехрестями знаходяться під контролем ЦПКТ, то в даному разі є можливість застосування «розумної» технології регулювання проїзду ТЗ з номером i із стартової позиції S_i до фінішної F_i (ці позиції кожен водій задає ЦПКТ). При цьому важливо знайти оптимальний маршрут руху для кожного ТЗ (у великих містах таких об'єктів може бути понад мільйон) на основі використання отримуваних з кожного перехрестя даних, що дуже швидко змінюються у відповідності із динамікою дорожнього трафіку. Останнє означає, що потрібно використовувати динамічну базу даних, спроможну оновлюватись, скажімо, кожні 10 с. Іншими словами, канали «Перехрестя ↔ ЦПКТ» працюють в режимі реального часу, постійно оновлюючи дані про завантаженість ділянок дороги між перехрестями.

У відповідь на вхідну інформацію ЦПКТ видає керуючі сигнали на світлофори кожного перехрестя, з однієї сторони, а з іншої, прокладає оптимальний трафік кожному ТЗ з урахуванням співвідношення (8.8). Така ситуація дозволяє не тільки покращити проїзд через окреме перехрестя, а і дає можливість прокладати маршрут кожному i-му ТЗ ($i = 1 \div N$, де N число автомобілів, що замовили маршрут ЦПКТ, тобто заявили свою пару (S_i, F_i). Крім того водій – при необхідності – заявляє також мову супроводу. Наприклад, водій-українець у Сакраменто (США) заявляє в якості звукового гіда українську мову.

Технічно ситуація виглядає наступним чином: програма на ЦПКТ працює з кожним ТЗ і для кожного такого об'єкта розраховує оптимальний маршрут руху і передає дані водієві на GPS-навігатор чи на мобільний телефон із спеціальним додатком. «Знаючи» розрахований оптимальніший маршрут – на даний момент часу – навігатор «веде» водія по цьому маршруту, з

використанням постійно оновлюваної, «свіжої» інформації. При цьому водій отримує інформацію про те, де він має повернути і в який бік, розвернутись чи перелаштуватись на іншу смугу руху і т.п. Але дорожня ситуація у сучасному місті змінюється кожну мить. І тому по мірі руху кожного ТЗ програма контролює маршрут та знаходиться в постійному пошуку найефективнішого маршруту. Якщо для даного водія був прокладений певний маршрут на момент часу t = 0 і в момент часу t>0 ситуація на цьому конкретному маршруті погіршилась (затор, ДТП тощо), то програма «поведе» водія по новому, але оптимальному маршруту. І так далі, поки ТЗ із номером i не досягне кінцевого пункту F_i .

Процедура прокладання маршруту здійснюється за допомогою програми, яка використовує **алгоритм Дейкстри**. Для складання такої програми в нашій роботі використовується мова програмування **Java.**

Розглянемо частину міської транспортної мережі міста Парижа, представлену на рис. 8.5. Співставимо з цією транспортною мережею орієнтований навантажений граф, вершини якого представляють міські перехрестя, а ребра – вулиці, що виділені жирними лініями. Нехай поставлена задача прокласти оптимальний маршрут від перехрестя 1 (START) до перехрестя 13 (FINISH). Програма використовує в якості вхідних даних спектр величин, розрахованих за допомогою виразу (8.8), величини яких (для деякого моменту часу!) представлені на рис.8.6 біля ребер графа.

Перед технічним впровадженням пропонованого проекту необхідно здійснити імітаційне моделювання, тобто провести електронну апробацію даної технології. Дуже перспективним методом апробації такої технології є програма візуального імітаційного моделювання AnyLogic Profesional 8.4, в якій представлено ряд імітаційних моделей, зокрема чотири імітаційні моделі, що регулюють дорожній трафік. Найбільш ілюстративною є імітаційна модель Traffic Light Phases Optimization, що візуально імітує проїзд ТЗ через три сусідні перехрестя – два Т-подібні перехрестя та одне хрестоподібне. Програма дозволяє максимізувати інтенсивність проїзду ТЗ через кожне із названих перехресть за рахунок зміни вручну з допомогою слайдерів (спеціальних регуляторів) протяжності фаз горіння зеленого світла у взаємно перпендикулярних напрямках проїзних частин дороги. Суттєвим недоліком даної імітаційної моделі є саме «ручний» вибір протяжності горіння фаз світла.

Тепер слід звернути увагу на ту обставину, що граф, зображений на рис.8.6, являється двореберним планарним навантаженим орієнтованим графом. Це суттєво важливий момент. Принципова особливість полягає в тому, що прокладання шляху руху по місту кожного ТЗ здійснюється за допомогою спрощеної програми, представленої знизу. Ця програма прокладає маршрут, не враховуючи тієї обставини, що «ділянка дороги одного напряму між сусідніми перехрестями» власне складається, як правило, з кількох смуг руху, але програма не реєструє ТЗ на кожній окремій смузі. Реєструються лише транспортні засоби, що в'їжджають на ділянку дороги одного напрямку та виїжджають з неї. Важливо при цьому знати відношення між числом в'їжджаючих та числом виїжджаючих автомобілів. Якщо таке відношення близьке до одиниці, то це свідчить про відкриту для трафіку ділянку дороги між сусідніми перехрестями. Навпаки, якщо згадане відношення набагато більше одиниці, то це свідчить про дуже низьку динаміку транспортних засобів на вказаній ділянці. У такому випадку вага ребра відповідного графа буде дуже високою і значить таке ребро буде виключене із трафіку. Однак, може трапитись так, що деякі із смуг руху заблоковані, а інші – ні, хоча вірогідність таких ситуацій невелика. Отже, не дивлячись на сказане спрощення, представлений тут алгоритм цілком придатний та спроможний здійснювати супровід кожного ТЗ – водієві лише потрібно буде самостійно здійснювати перестроювання на суміжні смуги руху, знаючи із GPS-карти ділянку дороги на інцидентному (сусідньому) плечі прокладеного маршруту. Це не створюватиме особливих перешкод. Проте в подальших дослідженнях планується усунути такий недолік.

Для кожної пари (S_i, F_i) програма `package IC-7` вибирає оптимальний маршрут, тобто такий набір із (8.8), який є простим та самим коротким із всіх можливих варіантів ланцюгом. Застосування приведеної вище програми до графа, приведеного на рис.8.6 дозволяє прокласти самий короткий маршрут від пункту 1 (START) до пункту 13 (FINISH).

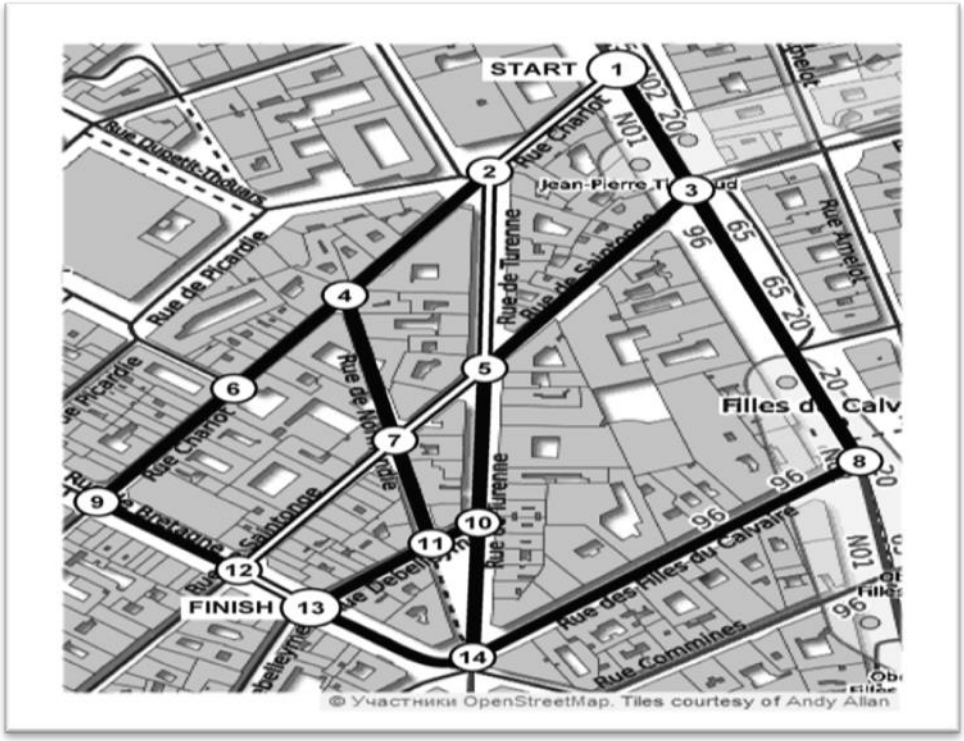

Рис.8.5. Транспортна мережа одного із кварталів у Парижа. Тут зроблені позначення перехресть (цифри від 1 до 14), що відповідають графу,представленому на рис.8.6. Перехрестя 1 відмічене надписом START, а перехрестя 13 визначено як кінцевий пункт поїздки –(FINISH).

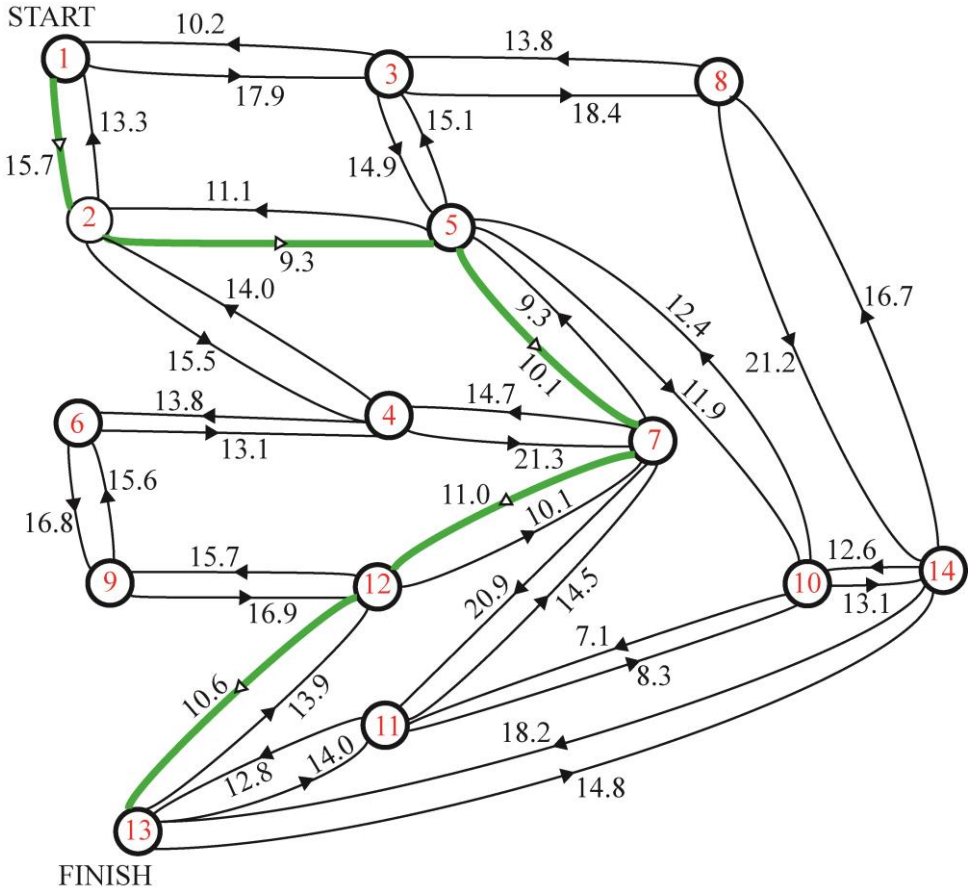

Рис.8.6.Орієнтований зв’язний навантажений граф. Написи «START» та «FINISH» відповідають вершинам графа відповідно 1 та 13 і символізують собою початок та кінець подорожі конкретного ТЗ. Один із можливих маршрутів між пунктами 1 і 13 для певного моменту часу виділений.

```
package IC-7;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.StringTokenizer;
public class GrDeicstra {
private static int INF = Integer.MAX_VALUE / 2;
double weightU;
int u;
private int n; //кількість вершин у графі
private int m; //кількість дуг у графі
private ArrayList adj[]; //список суміжності
private ArrayList weight[]; //вага ребра в орграфі
private boolean used[]; //масив для зберігання інформації про пройдені  та не
пройдені вершини
private double dist[]; //массив для зберігання відстані від стартової вершини
private int[] pred;//масив предків, необхідних  для відновлення найкоротшого шляху
від стартової вершини
int start; //стартова вершина, від якої знаходимо відстань до всіх інших
private BufferedReader cin;
private PrintWriter cout;
private StringTokenizer tokenizer;
private void dejkstra(int s) {//процедура запуска алгоритма Дейкстри із  стартової
вершини
dist[s] = 0; //найкоротша відстань до стартової вершини рівна 0
for (int k = 0; k < n; ++k)
{ int v = -1; double distV = INF;
for (int i = 0; i < n; ++i)
{ if (used[i])
{ continue; }
if (distV < dist[i])
{ continue; }
v = i;
distV =  dist[i]; }
for (int i = 0; i < adj[v].size(); ++i)
{ int u = (int) adj[v].get(i);
double weightU =(double) weight[v].get(i);
if  (dist[v] + weightU < dist[u])
{ dist[u] = dist[v] + weightU; pred[u] = (int)v; } }
used[v] = true;
} }
private void readData() throws IOException
{ cin = new BufferedReader(new InputStreamReader(System.in));
cout = new PrintWriter(System.out);
tokenizer = new StringTokenizer(cin.readLine());
n = Integer.parseInt(tokenizer.nextToken());
m = Integer.parseInt(tokenizer.nextToken());
start = Integer.parseInt(tokenizer.nextToken()) - 1;
adj = new ArrayList[n];
for (int i = 0; i < n; ++i)
{ adj[i] = new ArrayList();
}
//ініціалізація списку, в якому зберігаються ваги ребер
weight = new ArrayList[n];
for (int i = 0; i < n; ++i)
{ weight[i] = new ArrayList();
 }
//зчитуємо граф, заданий списком ребер
for (int i = 0; i < m; ++i)
{ tokenizer = new StringTokenizer(cin.readLine());
int u = Integer.parseInt(tokenizer.nextToken());
int v = Integer.parseInt(tokenizer.nextToken());
```

```
double w = Double.parseDouble(tokenizer.nextToken());
u--;
v--;
adj[u].add(v);
weight[u].add(w); }
used = new boolean[n];
Arrays.fill(used, false);
pred = new int[n];
Arrays.fill(pred, -1);
dist = new double[n];
Arrays.fill(dist, INF); }
void printWay(int v)
{ if (v == -1)
{ return; }
printWay(pred[v]);
cout.print((v + 1) + " "); }
private void printData() throws IOException
{ for (int v = 0; v < n; ++v)
{ if (dist[v] != INF)
{ cout.print(dist[v] + " "); }
else
{ cout.print("-1 "); }
}
cout.println();
for (int v = 0; v < n; ++v)
{ cout.print((v + 1) + ": ");
if (dist[v] != INF)
{ printWay(v); }
cout.println();
}
cin.close();
cout.close(); }
private void run() throws IOException { readData();
dejkstra(start);
printData();
cin.close();
cout.close(); }
public static void main(String[] args) throws IOException
{ GrDeicstra solution = new GrDeicstra();
solution.run();
} }
```

Для виконання програми слід ввести базові дані, що задають граф. Приведені знизу величини означають наступне: ***14*** – число вершин графа; ***40*** – число ребер графа; ***1*** – номер вершини, від якої стартує маршрут. Далі приведено 40 тріад чисел, перше з яких означає номер вихідної вершини, друге – номер вершини, до якої прокладається маршрут, третє – це дійсне число, що являє собою навантаженість ребра графа. Особливо наголосимо, що приведений спектр дійсних чисел (третя колонка) змінюється кожні 10 секунд. Ці числа представляють собою мультиплікати виду (8.8), отримувані в результаті аналізу даних із вхідних та вихідних датчиків.

14 40 1

1 2 15.7

2 1 13.3

1 3 17.9

3 1 10.2

3 8 18.4

13 14 14.8

7 5 9.3

7 4 14.7

4 7 21.3

4 6 13.8

6 4 13.1

7 11 20.9

11 7 14.5

10 5 12.4

5 10 11.9

8 14 21.2

8 3 13.8
3 5 14.9
5 3 15.1
2 5 9.3
5 2 11.1
2 4 15.5
4 2 14.0
5 7 10.1
6 9 16.8
9 6 15.6
9 12 16.9
12 9 15.7
12 7 10.1
7 12 11.0
12 13 10.6
13 12 13.9
14 8 16.7
13 11 14.0
11 13 12.8
11 10 8.3
10 11 7.1
10 14 13.1
14 10 12.6
14 13 18.2

Результат роботи програми – спектр чисел (виділених жирним шрифтом), що представляють собою відстані від вершини 1 до інших 14 вершин. Потім ідуть 14 рядків, що являють собою оптимальні маршрути від вершини 1 до інших вершин графа. Наприклад, нас цікавить маршрут «$1 \to 13$». Цей маршрут буде передавати водієві (на певний момент часу!) GPS-навігатор (при умові, що дорожня ситуація суттєво не зміниться).

0.0 15.7 17.9 31.2 25.0 45.0 35.1 36.3 61.8 36.9 44.0 46.1 *56.7* 50.0

1: 1
2: 1 2
3: $1 \to 3$
4: $1 \to 2 \to 4$
5: $1 \to 2 \to 5$
6: $1 \to 2 \to 4 \to 6$
7: $1 \to 2 \to 5 \to 7$
8: $1 \to 3 \to 8$
9: $1 \to 2 \to 4 \to 6 \to 9$
10: $1 \to 2 \to 5 \to 10$
11: $1 \to 2 \to 5 \to 10 \to 11$
12: $1 \to 2 \to 5 \to 7 \to 12$
13: ***1*** $\to$ ***2*** $\to$ ***5*** $\to$ ***7*** $\to$ ***12*** $\to$ ***13***
14: $1 \to 2 \to 5 \to 10 \to 14$

Отримавши приведені вище послідовності чисел, наприлад послідовність ***1*** $\to$ ***2*** $\to$ ***5*** $\to$ ***7*** $\to$ ***12*** $\to$ ***13***, програма в режимі GPS-навігації передає дані кожному водієві, причому ці дані, як і кожне ребро графа на рис.8.6, завдяки автоматизації кожного перехрестя в місті моніторяться кожні 10 секунд; іншими словами це означає, що ваги ребер постійно оновлюються у відповідності із даними, отримуваними із кожного перехрестя. В цьому, зокрема, є перевага нашої програми над дорожніми картами Google Maps, які працюють із запізненням і тому констатують дорожню ситуацію постфактум, що власне і знецінює таку інформацію, оскільки це не дає можливості водієві своєчасно реагувати на зміну дорожньої обстановки по його дорожній карті. В результаті – дорожний колапс у вигляді довготривалих міських заторів.

Впровадження пропонованої системи дозволить здійснити синхронізацію руху величезної маси автомобілів (наприклад, у місті Пекін на вулицях щоденно курсує понад 1 млн. ТЗ). Кожен водій, що під'єднався до ЦПКТ, отримує вказівки щодо руху по маршруту. Більше водію немає про що турбуватись – із ЦПКТ будуть передаватись голосові команди, як це робить звичайний GPS-навігатор. При зміні дорожньої обстановки (заблоковане перехрестя, ДТП по курсу і т.д.) програма миттєво розрахує новий, але оптимальний маршрут.

Зауважимо, що важливим чинником покращення надійності та достовірності даних на основі

аналізу результатів вхідних та вихідних датчиків, є інформація, отримувана із GPS-повідомлень. Справа в тому, що майже кожен водій працює якщо не з GPS-навігатором, то із мобільним телефоном чи смартфоном з програмою Google Maps. Це дозволяє використати GPS-дані в якості додаткового джерела інформації в плані розташування кожного автомобіля на карті міської транспортної мережі. Тут важливо зауважити, що GPS-повідомлення надають інформацію про місце розташування автомобіля з певною похибкою (2 – 12 м). В нашій задачі такі дані носять додатковий характер з точки зору прокладання маршруту. Враховуючи специфіку поставленої проблеми, наближений характер отримуваних даних цілком прийнятний і немає необхідності точно визначати положення кожного ТЗ: важливим є лише масив повідомлень, отримуваних з певною долею достовірності, що не перевищує 10%. Чому важливо поєднати дані, отримувані із дорожніх датчиків, з GPS-даними? Оскільки пропонована технологія передбачає використання GPS-навігаторів, то цілком логічно використати також і таку інформацію. На доцільність поєднання стаціонарних даних та мобільних, отримуваних із GPS-пристроїв чи смартфонів, вказується в ряді досліджень. Алгоритм використання GPS-даних для прокладання оптимальних маршрутів використовує набори GPS-повідомлень, що характеризуються географічною широтою, довготою та часовим маркером.

Оптимізація міського трафіку: A*-алгоритм

Алгоритм Дейкстри використовується для знаходження оптимальних маршрутів між вибраною вершиною графа та всіма іншими його вершинами. Проте коли потрібно знаходити найкоротший маршрут лише між двома конкретними вершинами графа, тоді більш доцільно використати A*-алгоритм, оскільки даний алгоритм має меншу обчислювальну складність, чим алгоритм Дейкстри (обчислювальна складність алгоритму Дейкстри $O(n^2)$, а обчислювальна складність A*-алгоритму пропорційна кількості n вузлів графа, тобто $O(n)$).

Проблема трафіка у великому місті є надзвичайно актуальною на сьогодні, але не вирішеною. В першу чергу така проблема пов'язана із заторами на перехрестях і, як наслідок, складністю проїзду кожним ТЗ по вибраному маршруту.

Як вже підкреслювалось, базовим елементом у технології регулювання міського трафіку являється перехрестя. Саме цей об'єкт, де перетинаються міські дороги, є основною причиною та джерелом заторів. Тому першочергово необхідно здійснити інтелектуальну регуляцію проїзду ТЗ через окреме перехрестя. Організувавши ефективний рух через такий об'єкт, досягнемо більш високої ефективності трафіку по всьому місту.

З метою запобігання колапсу трафіку у вигляді довготривалих заторів, автори пропонують застосувати мережу «розумних» перехресть, охоплюючих всі основні транспортні вузли міста. Але цього мало. Потрібно організувати оптимальний проїзд кожного ТЗ вздовж вибраного маршруту (нехай стартова позиція ТЗ з номером i позначається як S_i, а фінішна – F_i). Таким чином, кожному ТЗ приписується своєрідний маркер (S_i, F_i). Водій кожного такого об'єкта з допомогою мобільного телефону із спеціальним додатком чи GPS-навігатора задає свій маркер та координує маршрут з ЦПКТ. Таким чином з'являється можливість контролювати та прокладати оптимальні маршрути для всіх ТЗ (водіїв таких ТЗ називатимемо IR-водіями). Виходячи із всього вищесказаного, головна проблема полягає у поєднанні алгоритмів регулювання руху ТЗ як через окреме перехрестя, так і по всьому місту. При такому підході затори у великих містах виникатимуть значно рідше, ніж це має місце на сьогодні, а трафік перейде на якісно новий рівень функціонування.

З однієї сторони, необхідно створити алгоритм та комп'ютерну програму, що забезпечать ефективне перемикання світлофорних фаз у відповідності із завантаженістю дорожніх напрямків. З іншої, необхідно контролювани не тільки перехрестя, а і всі смуги руху проїзної частини дороги одного напрямку. На рис.8.7 схематично зображено траєкторію руху окремого ТЗ.

На другому етапі необхідно прокласти оптимальний маршрут для кожного ТЗ, використовуючи, наприклад, A*-алгоритм та спектр даних, отримуваних із інфраструктури міської дорожньої мережі. В результаті застосування описаних двох фаз регулювання міського трафіку

досягається оптимальний режим руху всього рухомого транспорту міста. Таким чином, пропонована технологія носить практичний характер та спрямована на вирішення конкретної проблеми – проблеми трафіку у великому місті.

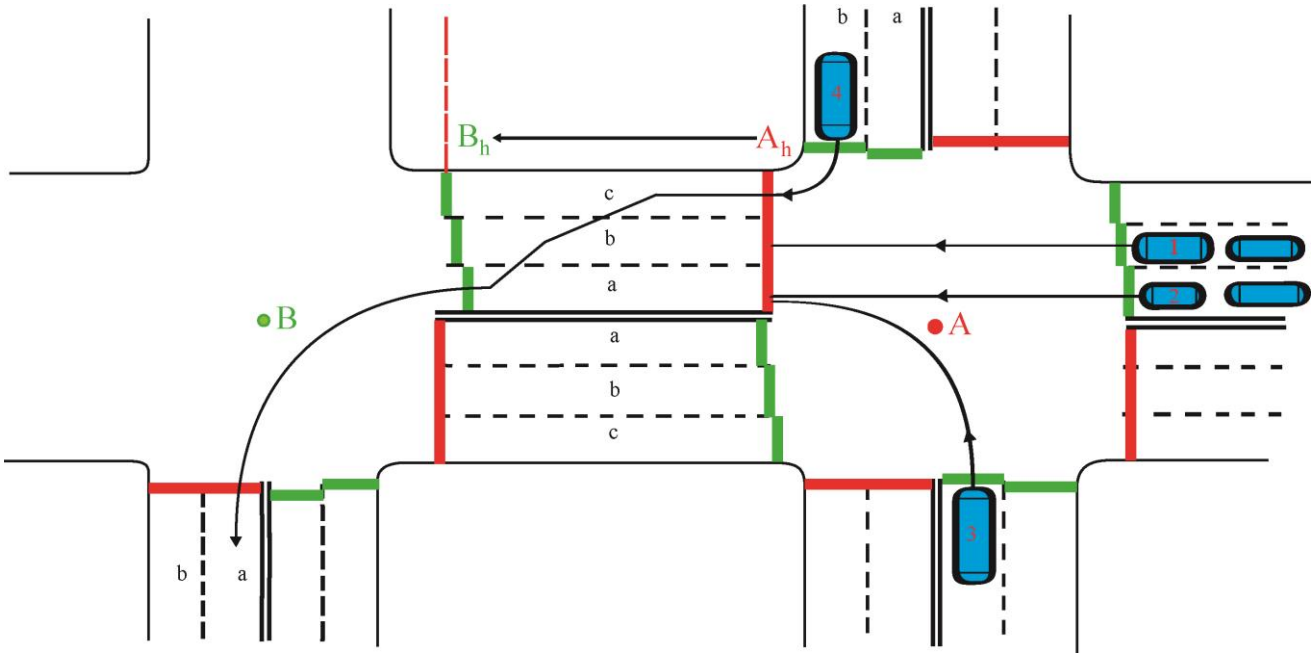

Рис. 8.7. Показано групи автомобілів 1,2,3 і 4, що під'їхали до перехрестя А, маршрут яких пролягає через ділянку дороги одного напрямку A_h ® B_h в напрямку перехрестя В. Для прикладу зображено траєкторію руху автомобіля 4. Цей автомобіль на ділянці руху A_h ® B_h реєструється вхідним датчиком A_h та вихідним датчиком B_h^a. Спочатку даний автомобіль в'їжджає на смугу руху $(A_h$ ® $B_h)_c$, потім перелаштовується на смугу руху $(A_h$ ® $B_h)_b$ і остаточно займає смугу руху $(A_h$ ® $B_h)_a$ і тому реєструється датчиком B_h^a. Далі вказаний автомобіль здійснює поворот наліво на перехресті В, де реєструється вхідним датчиком цього перехрестя.

В принциповому плані найважливіша проблема трафіку зводиться до того, щоб кожен IR-водій проїжджав заявлений ним маршрут за мінімально короткий час. Для цієї мети потрібно вирахувати такий маршрут на даний конкретний момент часу при відповідній завантаженості перехресть та смуг руху, що їх проїжджатиме кожен автомобіль типу IR.

Як і у випадку ситуації з алгоритмом Дейкстри, так і при використанні A*-алгоритму представимо транспортну мережу міста як навантажений мультиграф (рис.8.8). Це перший ключовий месідж. Тепер можна використати теорію графів з метою прокладання оптимальних маршрутів. Найбільш прийнятним варіантом буде А*-алгоритм, оскільки цей алгоритм дозволяє прокласти оптимальний маршрут в графі між двома заданими вершинами, іншими словами між заданими початковою $A(або\, S_i)$ та кінцевою $V(або\, F_i)$ точками маршруту, заявленого IR-водієм i.

Другий месідж полягає в тому, що ваги ребер є величинами швидкозмінними, високо динамічними, тому їх треба реєструвати в режимі «on-line», постійно оновлюючи базу даних, де зберігаються ці величини. Вхідні та вихідні датчики видають спектр величин $N_{A_hB_h}$ та n_h^i. Перша величина – це число автомобілів, що в'їхали на ділянку дороги між сусідніми перехрестями (на мал.8.7 це ділянка дороги A_h ® B_h) на протязі циклу роботи світлофора, а друга – це число автомобілів, що виїхали із i-ої смуги цієї ділянки дороги на сусідньому перехресті В за той самий

час. Чим ближчим є відношення $N_{A_hB_h} / \sum_{i=1}^{l} n_h^i$ до одиниці, тим динаміка руху на вибраній ділянці дороги між перехрестями є кращою. Навпаки, якщо відношення $N_{A_hB_h} / \sum_{i=1}^{l} n_h^i \gg 1$ то це свідчить або про заблокованість перехрестя B (рис.8.7) або про стан, близький до заблокованого. Відповідно вага ребра (іншими словами, завантаженість ділянки дороги $A_h \to B_h$) здобуває великих значень. Значить, програма, шукаючи в графі оптимальний маршрут, омине таку завантажену ділянку дороги.

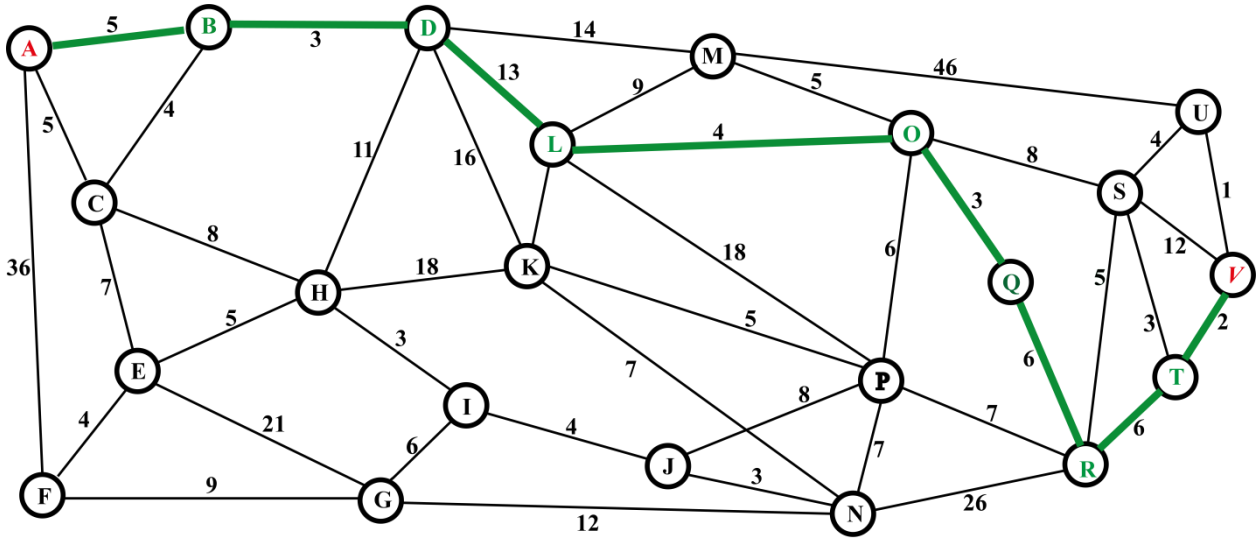

Рис.8.8. Зважений граф, в якому з допомогою A*-алгоритму прокладений оптимальний маршрут між вершинами A і V. Зеленим кольором забражений оптимальний маршрут руху з урахуванням ваг ребер на конкретний момент часу.

Приведені у лівій частині виразу (8.8) величини складатимуть ваги ребер графа (рис. 8.8). Ці величини динамічні (швидкозмінні) і відповідно система інтелектуального регулювання трафіку прокладатиме маршрути для всіх IR-водіїв з урахуванням дорожньої ситуації на даний конкретний момент часу. Іншими словами, система працює в режимі «on-line», що корінним чином відрізняє її від існуючих подібних систем. Для забезпечення роботи такого режиму використовується передача даних, отриманих на ЦПКТ від датчиків міської мережі, до кожного користувача цими даними, тобто IR-водія. Але оскільки дані від міських датчиків оновлюються кожні 10 с, то і отримувані IR-водіями дані також матимуть квазі-онлайн режим.

Java-програма, що відповідає графу на рис.8.7, виглядає наступним чином:

```
package IC-8;
    import java.util.PriorityQueue;
    import java.util.HashSet;
    import java.util.Set;
    import java.util.List;
    import java.util.Comparator;
    import java.util.ArrayList;
    import java.util.Collections;
    public class AstarSearchAlgo {
    public static void main(String[] args) {
    //евристичні відстані, виміряні по прямій,від кожної із
```

```
    //вершин графа  до кінцевої вершини маршруту V – рис.8.7
    Node n1 = new Node("A", 34);
    Node n2 = new Node("B", 31);
    Node n3 = new Node("C", 30);
    Node n4 = new Node("D", 27);
    Node n5 = new Node("E", 29);
    Node n6 = new Node("F", 33);
    Node n7 = new Node("G", 27);
    Node n8 = new Node("H", 24);
    Node n9 = new Node("I", 23);
    Node n10 = new Node("J", 19);
    Node n11 = new Node("K", 19);
    Node n12 = new Node("L", 19);
    Node n13 = new Node("M", 20);
    Node n14 = new Node("N", 16);
    Node n15 = new Node("O", 13);
    Node n16 = new Node("P", 13);
    Node n17 = new Node("Q", 9);
    Node n18 = new Node("R", 8);
    Node n19 = new Node("S", 6);
    Node n20 = new Node("T", 3);
    Node n21 = new Node("U", 3);
    Node n22 = new Node("V", 0);
  //ініціалізація інцидентних кожному вузлу ребер
n1.adjacencies = new Edge[]{
            new Edge(n2, 5),
            new Edge(n3, 5),};
n2.adjacencies = new Edge[]{
            new Edge(n4, 3),
            new Edge(n3, 4),
            new Edge(n6, 36) };
n3.adjacencies = new Edge[]{
            new Edge(n2, 4),
            new Edge(n4, 7),
            new Edge(n5, 7),
            new Edge(n8, 8)};
n4.adjacencies = new Edge[]{
            new Edge(n13, 14),
            new Edge(n12, 13),
            new Edge(n11, 16),
            new Edge(n8, 11),
            new Edge(n3, 7),
            new Edge(n2, 3),};
n5.adjacencies = new Edge[]{
            new Edge(n3, 7),
            new Edge(n8, 5),
            new Edge(n6, 4),
            new Edge(n7, 21),};
n6.adjacencies = new Edge[]{
            new Edge(n5, 4),
            new Edge(n7, 9),
            new Edge(n1, 36),};
n7.adjacencies = new Edge[]{
            new Edge(n6, 9),
```

```
            new Edge(n14, 12),
            new Edge(n9, 6),
            new Edge(n5, 21),};
        n8.adjacencies = new Edge[]{
            new Edge(n4, 11),
            new Edge(n5, 5),
            new Edge(n9, 3),
            new Edge(n3, 8),
            new Edge(n11, 18) };
        n9.adjacencies = new Edge[]{
            new Edge(n10, 4),
            new Edge(n8, 3),
            new Edge(n9, 6), };
        n10.adjacencies = new Edge[]{
            new Edge(n16, 8),
            new Edge(n14, 3),
            new Edge(n9, 4) };
        n11.adjacencies = new Edge[]{
            new Edge(n12, 5),
            new Edge(n16, 5),
            new Edge(n14, 7),
            new Edge(n4, 16),
            new Edge(n8, 18) };
        n12.adjacencies = new Edge[]{
            new Edge(n13, 9),
            new Edge(n15, 4),
            new Edge(n11, 5),
            new Edge(n4, 13),
            new Edge(n16, 18)};
        n13.adjacencies = new Edge[]{
            new Edge(n15, 5),
            new Edge(n12, 9),
            new Edge(n4, 14),
            new Edge(n21, 46),  };
        n14.adjacencies = new Edge[]{
            new Edge(n11, 7),
            new Edge(n16, 7),
            new Edge(n7, 12),
            new Edge(n10, 3),
            new Edge(n18, 26) };
        n15.adjacencies = new Edge[]{
            new Edge(n13, 5),
            new Edge(n12, 4),
            new Edge(n17, 3), };
        n16.adjacencies = new Edge[]{
            new Edge(n11, 5),
            new Edge(n10, 8),
            new Edge(n14, 7),
            new Edge(n17, 4),
            new Edge(n18, 7),
            new Edge(n12, 18),};
        n17.adjacencies = new Edge[]{
            new Edge(n16, 4),
            new Edge(n15, 3),
```

```
            new Edge(n18, 6),};
    n18.adjacencies = new Edge[]{
            new Edge(n16, 7),
            new Edge(n17, 6),
            new Edge(n19, 5),
            new Edge(n20, 6),
            new Edge(n14, 26), };
    n19.adjacencies = new Edge[]{
            new Edge(n15, 8),
            new Edge(n18, 5),
            new Edge(n21, 4),
            new Edge(n20, 3),
            new Edge(n22, 12),};
    n20.adjacencies = new Edge[]{
            new Edge(n22, 2),
            new Edge(n18, 6),
            new Edge(n19, 3),};
    n21.adjacencies = new Edge[]{
            new Edge(n19, 4),
            new Edge(n22, 1),
            new Edge(n13, 46), };
    n22.adjacencies = new Edge[]{
            new Edge(n21, 1),
            new Edge(n20, 2),
            new Edge(n19, 12), };
AstarSearch(n1, n22);
List<Node> path = printPath(n22);
System.out.println("Path: " + path);}
public static List<Node> printPath(Node target) {
List<Node> path = new ArrayList<Node>();
for (Node node = target; node != null; node = node.parent) {
path.add(node);}
Collections.reverse(path);
return path;}
public static void AstarSearch(Node source, Node goal) {
Set<Node> explored = new HashSet<Node>();
PriorityQueue<Node> queue = new PriorityQueue<Node>(30,new Comparator<Node>() {
//визначаємо метод порівняння
public int compare(Node i, Node j) {
if (i.f_scores > j.f_scores) {
return 1;
} else if (i.f_scores < j.f_scores) {
return -1;
} else {
return 0;
}}});
  source.g_scores = 0;
queue.add(source);
boolean found = false;
while ((!queue.isEmpty()) && (!found)) {
//вузол, що має найменшу величину f_score
Node current = queue.poll();
explored.add(current);
if (current.value.equals(goal.value)) {
```

```
found = true;}
/перевірка кожного сусіднього вузла
for (Edge e : current.adjacencies) {
Node child = e.target;
double cost = e.cost;
double temp_g_scores = current.g_scores + cost;
double temp_f_scores = temp_g_scores + child.h_scores;
/*якщо сусідній вузол оцінений і нова величина
 f_score  є більшою, то тоді здійснюємо перехід*/
 if ((explored.contains(child)) &&
(temp_f_scores >= child.f_scores)) {
 continue;}
 else if ((!queue.contains(child)) ||
 (temp_f_scores < child.f_scores)) {
 child.parent = current;
 child.g_scores = temp_g_scores;
 child.f_scores = temp_f_scores;
 if (queue.contains(child)) {
 queue.remove(child);}
 queue.add(child);
 }}}}}
 class Node {
 public final String value;
 public double g_scores;
 public final double h_scores;
 public double f_scores = 0;
 public Edge[] adjacencies;
 public Node parent;
 public Node(String val, double hVal) {
 value = val;
 h_scores = hVal;}
 public String toString() {
    return value;
 }}
 class Edge {
 public final double cost;
 public final Node target;
 public Edge(Node targetNode, double costVal) {
 target = targetNode;
 cost = costVal;
 }}
```

На початку програми використовуються ряд конструкторів, що вводять евристичні відстані, тобто значення функції h(n) для кожного вузла графа. Далі ініціалізуються всі вузли графа та описуються сусідні до кожного вузла ребра, тобто вводяться їх ваги. Аналізуючи введені дані, програма прокладе оптимальний маршрут ***Path: [A, B, D, L, O, Q, R, T, V]***, що є оптимальним лише на певний проміжок часу при відповідній завантаженості ребер графа. Такий маршрут виділено на рис. 8.8 – це потовщена ламана лінія. Дані щодо такого маршруту будуть передаватись IR-водієві із ЦПКТ, поки ситуація на трасі руху не зміниться. В такому разі програма обрахує новий маршрут. І так для всіх ТЗ по всьому місту. В результаті відбудеться цілковита синхронізація транспортних потоків, що приведе до повного зникнення заторів у транспортній мережі та дозволить кожному водієві прибувати до місця призначення за мінімально короткий проміжок часу. Таким чином, міський трафік перейде на якісно новий рівень.

Тестування приведеної програми проводилось на графах з числом вершин до 150 та числом ребер 430. При цьому час виконання складав складав 2 с 580 мс. Взагалі, обчислювальна складність A*- алгоритму лінійна $O(n)$, що робить його досить ефективним для задач з великим об'ємом вхідних даних.

Література

1. Руководство по языку программирования Java. – https://metanit.com/java/tutorial/.
2. Шилдт Г. Java. Полное руководство. – 8-е изд.: Пер. с англ. – М .: ООО «И.Д. Вильямс», 2012. – 1104 с.
3. Хорстман К.С., Корнелл Г. Java 2. – Том 1 Основы. – М.: Издательский дом «Вильямс», 2007. – 896 с. – (Серия «Библиотека профессионала»).
4. Oracle. The Java™ Tutorials. – https://docs.oracle.com/javase/tutorial/.
5. Java Tutorial. – https://www.tutorialspoint.com/java/.
6. Хабибуллин И. Самоучитель Java – 3-е изд.:– Санкт-Петербург: «БХВ-Петербург», 2012.– 750 с.
7. Арнольд К., Гослинг Д. Язык программирования Java. – Санкт-Петербург: «издательский дом Питер», 2013.– 745 с.
8. Эккель Б. Философия Java. – 4-е изд.– Санкт-Петербург: «издательский дом Питер», 2009.– 637 с.
9. Флэнаган Д. Java в примерах.Справочник.– Санкт-Петербург– Москва: «издательский дом Питер», 2009.– 642 с.
10. Аккуратов Е.Е. Самоучитель. Знакомтесь: Java. – М.: Издательский дом «Вильямс», 2012. – 250 с. – (Серия «Библиотека профессионала»).
11. Horstman Cay. Core Java SE9 for the Impatient. – Second edition.– Addison Wesley, 2018.– 1818 p.