

А. Г. Горбань



Програмування
в
JAVATM
2008

Caros Colegas!

Нічого й думати освоїти Java технології за 16 тижнів, працюючи по дві години на тиждень. Інша справа, познайомитися з цією технологією, знати її особливості і можливості і щось навчитися в ній робити. А потім, коли обставини заставлять вас глибше вникнути в цю технологію, ви вже не повинні її боятися і можете сміливо приступати до роботи.

Я пропоную вам курс програмування у Java з 19 уроків. Скільки уроків ви зможете засвоїти залежить від нас з вами. В такому обсязі курс програмування у Java читається в нашому університеті вперше. Я готував його майже рік. Але не все ще доведено до досконалості. Я перечитую текст декілька разів і кожного разу знаходжу опечатки. Впевнений, що ви будете знаходити їх і далі.

По ходу вивчення ви повинні виконати 14 або 15 лабораторних робіт. В кінці семестру залік — можливо ви не знаєте, що в програмуванні це гірше за екзамен. **Ви можете одержати його лише у випадку, коли не пропустите жодного заняття і здастес всі лабораторні роботи.**

За кожне пропущене заняття треба буде розрахуватися програмою, яка ілюструє всі методи указаного мною класу, на зразок того, як це зроблено в одному з уроків для класу String.

Спішити ми не будемо, тому рекомендую сильнішим студентам, не чекаючи нас із слабенькими та ледаченькими студентами, іти вперед і постаратися освоїти як можна більше.

А. Горбань

4.02.2008

ЗМІСТ

Урок 1. Вступ	4
Урок 2. Базовий курс Java	11
Урок 3. Об'єктно-орієнтоване програмування в Java	42
Урок 4. Класи-оболонки	74
Урок 5. Робота з рядками	86
Урок 6. Класи-колекції	99
Урок 7. Класи-утиліти	115
Урок 8. Принципи побудови графічного інтерфейса	121
Урок 9. Графічні примітиви	128
Урок 10. Основні компоненти	152
Урок 11. Розміщення компонентів	175
Урок 12. Обробка подій	183
Урок 13. Створення меню	202
Урок 14. Аплети	210
Урок 15. Зображення і звук	226
Урок 16. Обробка виключчів ситуацій	252
Урок 17. Підпроцеси	262
Урок 18. Потоки введення/виведення	280
Урок 19. Мережеві засоби Java	301

Програмування в Java

Урок 1. Вступ

Поздоровляю вас зі вступом в ряди програмістів на Java — розроблювачів технології початку ХХІ століття. Всі уроки ви повинні прочитувати вдома, а на лекції і лабораторних роботах ви повинні звітуватися за виконані завдання. Перше серйозне завдання ви знайдете в кінці Уроку 2. Всі завдання індивідуальні, щоб одержати залік в кінці семестру їх треба обовязково виконати. Отже, встановлюйте на своїх комп'ютерах **JDK** (що це таке дивись далі) - його можна скачати з Інтернет безкоштовно або діставайте яке-небудь інтегроване середовище програмування (**JBuilder**, **Eclipse...**). Останнє теж можна безкоштовно скачати з Інтернет, але вам прийдеться самостійно навчитися ним користуватися. Більше всього, що в лабораторії ми будемо користуватися починаючи з третього уроку середовищем **JBuilder**.

1.1. Що таке Java

Це острів Ява в Малайському архіпелазі, територія Індонезії. Це сорт кофе, який полюбляють пити творці **Java**. А якщо серйозно, то відповісти на це питання досить важко, тому що граници **Java**, і без того розмиті, весь час розширяються. Спочатку **Java** (офіційний день народження технології **Java** — 23 травня 1995 р.) призначалась для програмування побутових електронних пристрій, таких як мобільні телефони. Потім **Java** стала застосовуватися для програмування браузерів — з'явилися *аплети*. Потім виявилось, що на **Java** можна створювати повноцінні аплікації. Їх графічні елементи стали оформляти у вигляді компонентів — з'явилися *JavaBeans*, з котрими **Java** ввійшла в світ розподілених систем і проміжного програмного забезпечення, тісно повязаних з технологією **CORBA**. Остався один крок до програмування серверів — цей крок був зроблений — з'явилися *сервлети* і *EJB* (Enterprise JavaBeans). Сервери повинні взаємодіяти з базами даних — з'явилися драйвери **JDBC** (Java DataBase Connection). Взаємодія виявилася ефективною, і багато систем управління базами даних і навіть операційні системи включили **Java** в своє ядро, наприклад **Oracle**, **Linux**, **MacOS X**, **AIX**. Що ще не охоплено? Назвіть, і через півроку почуєте, що **Java** уже застосовується і там. Із-за такої розмитості самого поняття його описують таким же розмитим словом — *технологія*.

Прочитавши цей абзац, ви, без сумніву, відчуєте комплекс неповноцінності — скільки в області інформатики, до якої ви і себе причисляєте, існує речей, про які ви не маєте жодного уявлення. Та замість розчарування це повинно надати вам запалу до найскорішого освоєння хоча би програмування у **Java**.

Таке швидке і широке розповсюдження технології **Java** не в останню чергу повязано 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++** і ін., увібравши їх кращі, на думку творців, риси і відкинувши гірші. На цей рахунок єсть різні думки, але безперечно, що мова виявилася зручною для вивчення, написані на ній програми легко читаються і налаштовуються: першу програму можна написати уже через годину після початку вивчення мови. Мова **Java** становиться мовою навчання об'єкто-орієнтованому програмуванню, так само, як мова **Pascal** була мовою навчання структурному програмуванню. Недарма на **Java** уже написано величезна кількість програм, бібліотек класів, а власний аплет не написав тільки вже зовсім лінівий.

Для повноти картини належить сказати, що створювати аплікації для технології **Java** можна не тільки в мові **Java**, уже з'явилися і інші мови, єсть навіть компілятори з мов **Pascal** і **C++**, але краще все-таки використовувати мову **Java**; на ній всі аспекти технології викладаються простіше і зручніше.

Ясно, що всю технологію **Java** неможливо викласти в декількох лекціях, повне її описання складе цілу бібліотеку. Ми торкнемося тільки мови **Java**. Після цього ви зможете створити **Java** аплікації будь-якої складності, вільно розбиратися в літературі і лістингах програм, продовжувати вивчення аспектів технології **Java** по спеціальній літературі. Мова **Java** теж дуже бурхливо розвивається, деякі її методи оголошуються застарілими (**deprecated**), появляються нові конструкції, збільшується вбудована бібліотека класів, але єсть стабільне ядро мови, зберігається її дух і стиль. Ось на ньому ми і зконцентруємо нашу увагу.

1.2. Виконання Java-програми

Як ви знаєте, програма, написана на одній із мов високого рівня, до котрих відноситься і мова **Java**, так званий *вихідний модуль* ("сирець" на жаргоні, від англійського "source"), не може бути зразу ж виконана. Її спочатку треба скомпілювати, тобто перевести в послідовність машинних команд — *об'єктний модуль*. Але і він, як правило, не може бути зразу ж виконаним: об'єктний модуль треба ще зкомпонувати із бібліотеками використовуваних в модулі функцій і організувати перехресні посилки між секціями об'єктного модуля, одержавши в результаті *завантажувальний модуль* — повністю готову до виконання програму.

Вихідний модуль, написаний на **Java**, не може уникнути цих процедур, але тут проявляється головна особливість технології **Java** — програма компілюється не зразу в машинні команди, 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. Для спеціалістів скажемо, що віртуальна машина **Java** повністю стекова, так що не вимагається складна адресація комірок пам'яті і велика кількість реєстрів. Тому команди **JVM** короткі, більшість з них має довжину 1 байт, звідси команди **JVM** називають *байткодами (bytecodes)*, хоча єсть команди довжиною 2 і 3 байти. Згідно статистичних досліджень середня довжина команди складає 1,8 байта. Повне описання команд і всієї архітектури **JVM** міститься в *специфікації віртуальної машини Java (VMS, Virtual Machine Specification)*.

Друга особливість **Java** — всі стандартні функції, що викликаються в програмі, підключаються до неї тільки на етапі виконання, а не включаються в байт-коди. Як говорять спеціалісти, відбувається *динамічне компонування (dynamic binding)*. Це теж сильно зменшує обем скомпільованої програми.

Отже, на першому етапі програма, написана на мові **Java**, переводиться компілятором в байт-коди. Ця компіляція не залежить від типу якого-небудь конкретного процесора і архітектури деякого конкретного комп'ютера. Вона може бути виконана зразу ж після написання програми. Байт-коди записуються в одному або декількох файлах, можуть зберігатися у зовнішній пам'яті або передаватися по мережі. Це особливо зручно дякуючи невеликому розміру файлів з байт-кодами. Потім одержані в результаті компіляції байт-коди можна виконувати на будь-якому комп'ютері, котрий має систему реалізації **JVM**. При цьому не має значення ні тип процесора, ні архітектура комп'ютера. Так реалізується принцип **Java "Write once, run anywhere"** — "Написано раз, виконуєтьсяде завгодно".

Інтерпретація байт-кодів і динамічне компонування значно сповільнюють виконання програм. Це не має значення в тих ситуаціях, коли байт-коди передаються по мережі, мережа все рівно повільніша любої інтерпретації, але в інших ситуаціях вимагається потужний і швидкий комп'ютер. Тому постійно йде вдосконалення інтерпретаторів в сторону збільшення швидкості інтерпретації. Разроблені **JIT-компілятори (Just-In-Time)**, запам'ятуючі уже інтерпретовані частки кода в машинних командах процесора і просто виконуючи ці участки при повторному зверненні, наприклад, в циклах. Це значно збільшує швидкість обчислень, що повторяються. Фірма **SUN** розробила цілу технологію **Hot-Spot** і включає її в свою віртуальну машину **Java**. Але, звичайно, найбільшу швидкість може дати тільки спеціалізований процесор.

Фірма **SUN Microsystems** випустила мікропроцесори **PicoJava**, що працюють на системі команд **JVM**, і збирається випускати цілу серію все більш потужних Java-процесорів. Єсть уже і **Java**-процесори інших фірм. Ці процесори безпосередньо виконують байт-коди. Але при виконанні програм **Java** на інших процесорах вимагається ще інтерпретація команд **JVM** в команди конкретного процесора, а значить, потрібна програма-інтерпретатор, причому для кожного типу процесорів, і для кожної архітектури комп'ютера треба написати свій інтерпретатор.

Це завдання уже виїшено практично для всіх комп'ютерних платформ. На них реалізовані віртуальні машини **Java**, а для найбільш розповсюджених платформ існує декілька реалізацій **JVM** різних фірм. Все більше операційних систем і систем управління базами даних включають реалізацію **JVM** в своє ядро. Створена і спеціальна операційна система **JavaOS**, яка застосовується в електронних пристроях. В більшості браузерів вбудована віртуальна машина Java для виконання аплетів.

Уважний читч уже помітив, що крім реалізації **JVM** для виконання байт-кодів на комп'ютері ще потрібно

мати набір функцій, які викликаються із байт-кодів і динамічно компонуються з байт-кодами. Цей набір оформляється у вигляді бібліотеки класів [Java](#), яка складається з одного або декількох *пакетів*. Кожна функція може бути записана байт-кодами, але, оскільки вона буде зберігатися на конкретному комп'ютері, її можна записати прямо в системі команд цього комп'ютера, уникнувши тим самим інтерпретації байт-кодів. Такі функції називають *"рідними" методами* ([native methods](#)). Застосування "рідних" методів прискорює виконання програми.

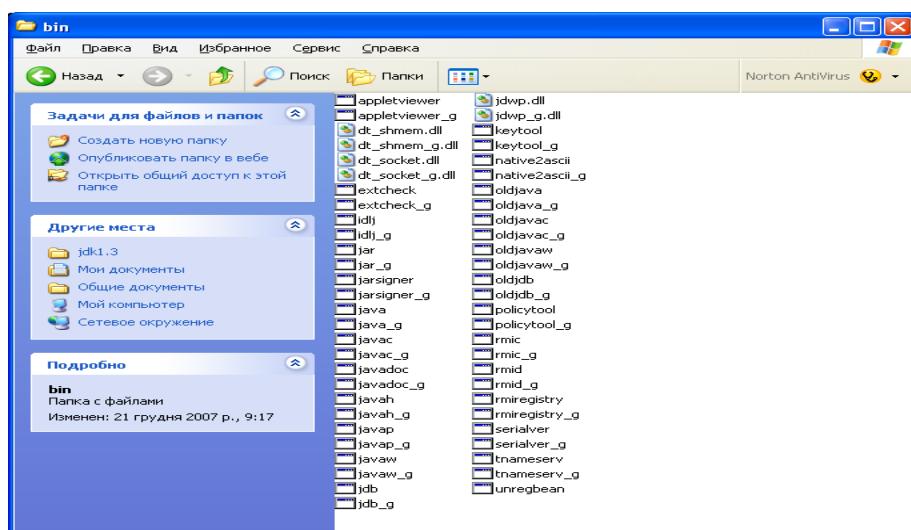
Фірма **SUN Microsystems** — творець технології **Java** — безкоштовно розповсюджує набір необхідних програмних інструментів для повного циклу роботи з цією мовою програмування: компіляції, інтерпретації, налаштовування, який включає і багату бібліотеку класів, під назвою **JDK (Java Development Kit)**. Є набори інструментальних програм і інших фірм. Наприклад, великою популярністю користується **JDK** фірми **IBM**.

1.3. Що таке JDK

Набір програм і класів **JDK** містить:

- компілятор **javac** із вихідного тексту в байт-коди;
 - інтерпретатор **java**, який містить реалізацію **JVM**;
 - полегшений інтерпретатор **jre** (в останніх версіях відсутній);
 - програму перегляду аплетів **appletviewer**, що замінє браузер;
 - налаштовувач **jdt**;
 - дизассемблер **javap**;
 - програму архівації і стиснення **jar**;
 - програму збору документації **javadoc**;
 - програму **javah** генерації заголовкових файлів мови C;
 - програму **javakey** додавання електронного підпису;
 - програму **native2ascii**, яка перетворює бінарні файли в текстові;
 - програми **rmic** і **rmiregistry** для роботи з віддаленими об'єктами;
 - програму **serialver**, яка визначає номер версії класу;
 - бібліотеки і заголовкові файли "рідних" методів;
 - бібліотеку класів **Java API (Application Programming Interface)**.

В попередні версії **JDK** включались і налаштовувальні варіанти виконуваних програм: **javac_g**, **java_g** і т.д. Компанія **SUN** Microsystems постійно розвиває і обновляє **JDK**, кожний рік появляються нові версії. Відкрийте папку **bin**, яка входить у **JDK** і ви побачите, які програми містить ваша версія. Ось що знаходиться на моєму комп'ютері.



Напряму я користувався лише трьома з цих програм (`javac` – компілятор, `java` – інтерпретатор, `appletviewer` – перегляд аплетів). Цього досить, щоб написати і випробувати програми, пов’язані з обчисленим і графікою, чим ми власне і займаємося. Дивлячись на інші програми цієї папки можете уявити, що технологія Java не вичерпується обчисленими і графікою.

У 1996 р. була випущена перша версія **JDK 1.0**, яка модифікувалась до версії з номером 1.0.2. У цій версії бібліотека класів **Java API** містила 8 пакетів. Весь набір **JDK 1.0.2** поставлявся в упакованому вигляді в одному файлі розміром близько 5 Мбайтів, а після розпакування займав близько 8 Мбайтів на диску.

В 1997 р. з'явилась версія **JDK 1.1**, остання її модифікація, 1.1.8, випущена в 1998 р. У цій версії було 23 пакети класів, займала вона 8,5 Мбайтів в упакованому вигляді і близько 30 Мбайтів на диску.

У перших версіях **JDK** всі пакети бібліотеки **Java API** були упаковані в один архівний файл `classes.zip` і викликались безпосередньо із цього архіву, його не треба розпаковувати.

Потім набір інструментальних засобів був сильно перероблений. Версія **JDK 1.2** вийшла в грудні 1998 р. і містила уже 57 пакетів класів. В архівному вигляді цей файл розміром майже 20 Мбайт і ще окремий файл розміром більше 17 Мбайт з упакованою документацією. Повна версія займає 130 Мбайт, із них близько 80 Мбайт займає документація.

Починаючи з версії **JDK 1.2**, всі продукти технології **Java** власного виробництва компанія **SUN** стала називати **Java 2 Platform, Standard Edition**, скорочено **J2SE**, а **JDK** перейменувала в **Java 2 SDK, Standard Edition** (**Software Development Kit**), скорочено **J2SDK**, оскільки випускається ще **Java 2 SDK Enterprise Edition** і **Java 2 SDK Micro Edition**. Між іншим, сама компанія **SUN** часто використовує і стару назву, а в літературі закріпилася назва **Java 2**. Крім 57 пакетів класів, обов’язкових на любій платформі і одержавших назву **Core API**, в **Java 2 SDK v1.2** входять ще додаткові пакети класів, названих **Standard Extension API**. У версії **2 SDK SE, v1.3**, яка вийшла в 2000 р., уже 76 пакетів класів, які складають **Core API**. В упакованому вигляді це файл розміром близько 30 Мбайт, і ще файл з упакованою документацією розміром 23 Мбайти. Все це розпаковується в 210 Мбайт дискового простору. Ця версія вимагає процесор **Pentium 166** і вище і не менше 32 Мбайтів оперативної пам’яті.

Зараз версія **JDK 1.0.2** уже не використовується. Версія **JDK 1.1.5** з графічною бібліотекою **AWT** вбудована в популярні браузери **Internet Explorer 5.0** і **Netscape Communicator 4.7**, тому вона застосовується для створення аплетів. Технологія **Java 2** широко використовується на серверах і в клієнт-серверних системах.

Щоб уявити всю складність **Java** технології, відкрийте файл **Index** в папці **Packages**. Тут ви знайдете близько сотні пакетів, і в деяких з них міститься по декілька десятків класів. Відкрити ці пакети можна з папки **JBuilder"\jdk"\src\src\java**. (Замість зірочок треба поставити номери ваших версій програмного продукту).

Крім **JDK**, компанія **SUN** окрім розповсюджує ще і набір **JRE (Java Runtime Environment)**.

1.4. Що таке **JRE**

Набір програм і пакетів класів **JRE** містить все необхідне для виконання байт-кодів, в тому числі інтерпретатор **java** (в попередніх версіях полегшений інтерпретатор **jre**) і бібліотеку класів. Це частина **JDK**, яка не містить компілятори, налаштовувачі і інші засоби розробки. Якраз **JRE** або його аналог інших фірм міститься в браузерах, що вміють виконувати програми на **Java**, в операційних системах і системах управління базами даних.

Хоча **JRE** входить до складу **JDK**, фірма **SUN** розповсюджує цей набір і окремим файлом.

Версія **JRE 1.3.0** — це архівний файл розміром близько 8 Мбайтів, що розвертається у 20 Мбайтів на диску.

Після установки **Java** ви одержите каталог з назвою, наприклад, **jdk1.3**, а в ньому підкаталоги:

- **bin**, який містить виконувані файли;
- **demo**, який містить приклади програм;
- **docs**, котрий містить документацію, якщо ви її установили;
- **include**, котрий містить заголовкові файли "рідних" методів;
- **jre**, котрий містить набір JRE;
- **old-include**, для сумісності зі старими версіями;
- **lib**, котрий містить бібліотеки класів і файли властивостей;
- **src**, з вихідними текстами програм **JDK**. В нових версіях замість каталогу знаходиться упакований файл **src.jar**.

Так-так! Набір **JDK** містить вихідні тексти більшості своїх програм, написані на **Java**. Це дуже зручно. Ви завжди можете з точністю дізнатись, як працює той чи інший метод обробки інформації із **JDK**, проглянувши вихідний код даного метода. Це дуже корисно і для вивчення **Java** на "живих" працюючих прикладах.

Якщо ваш **JDK** являється частиною якогось інтегрованого середовища програмування, наприклад **JBuilder**, то деякі з перечислених вище папок можуть знаходитися в іншому місці. Прогляньте вашу папку **JDK**.

1.5. Як користуватися **JDK**

Найпопулярнішим у нас є середовище програмування **JBuilder**, дуже схоже на **Delphi**, так як створені однією й тією ж фірмою. На ньому ми і зупинимося. Як користуватися **JBuilder** – розглянемо згодом. Але існує велика кількість і інших можливостей, аж до написання програми в Блокнот.

1.6. Java в Internet

Розроблена для використання в мережах, **Java** просто не могла не знайти відображення на сайтах **Internet**. Дійсно, маса сайтів повністю присвячена або містить інформацію про технологію **Java**. Одна тільки фірма **SUN** має декілька сайтів з інформацією про **Java**:

- <http://www.sun.com/> — тут всі посилки, звідси можна скопіювати **JDK**;
- <http://java.sun.com/> — основний сайт **Java**, звідси теж можна скопіювати **JDK**;
- <http://developer.java.sun.com/> — маса корисних речей для розроблювача;
- <http://industry.java.sun.com/> — новини технології **Java**;
- <http://www.javasoftware.com/> — сайт фірми **JavaSoft**, підрозділу **SUN**;
- <http://www.gamelan.com/>.

На сайті фірми **IBM** є великий розділ <http://www.ibm.com/developer> . /Java/, де можна знайти дуже багато корисного для програміста.

Компанія **Microsoft** містить інформацію про **Java** на своєму сайті: <http://www.microsoft.com/java/>.

Великий внесок в розвиток технології **Java** робить корпорація **Oracle**: <http://www.oracle.com/>.

Існує багато спеціалізованих сайтів:

- <http://java.iba.com.by/> — Java team IBA (Білорусія);
- <http://www.artima.com/>;
- <http://www.freewarejava.com/>;
- <http://www.jars.com/> — Java Review Service;

- <http://www.javable.com> — російськомовний сайт;
- <http://www.javaboutique.com/>;
- <http://www.javalobby.com/>;
- <http://www.javalogy.com/>;
- <http://www.javaranch.com/>;
- <http://www.javareport.com/> — незалежне джерело інформації для розроблювачів;
- <http://www.javaworld.com> — електронний журнал;
- <http://www.jfind.com> — збірник програм і статей;
- <http://www.jguru.com> — поради спеціалістів;
- [http://www.novocode.com/](http://www.novocode.com);
- <http://www.sigs.com/jro/> — Java Report Online;
- <http://www.sys-con.com/java/>;
- <http://theserverside.com/> — питання створення серверних Java-аплікацій;
- <http://servlets.chat.ru/>;
- <http://javapower.da.ru/> — збірник FAQ російською мовою;
- <http://www.purejava.ru/>;
- <http://java7.da.ru/>;
- <http://codeguru.earthweb.com/java/> — великий збірник аплетів і інших програм;
- <http://securingjava.com/> — обговорюються питання безпеки;
- <http://www.servlets.com/> — питання по написанню аплетів;
- <http://www.servletsource.com/>;
- <http://cool servlets.com/>;
- <http://www.servletforum.com/>;
- <http://www.javacats.com/>.

Персональні сайти:

- <http://www.bruceeckel.com> — сайт Bruce Eckel;
- <http://www.davidreilly.com/java/>;
- <http://www.comita.spb.ru/users/sergeya/java> — питання, що стосуються русифікації Java.

На жаль, адреси сайтів часто міняються. Можливо, ви вже не знайдете деяких із перечислених сайтів, зате появляється багато інших.

1.7. Література по Java

Повне і строго описання мови викладено в книзі *The Java Language Specification, Second Edition. James Gosling, Bill Joy, Guy Steele, Gilad Bracha*. Ця книга в електронному вигляді знаходиться за адресою http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html і займає в упакованому вигляді близько 400 Кбайт.

Take ж повне і строго описання віртуальної машини Java викладено в книзі «*The Java Virtual Machine Specification, Second Edition. Tim Lindholm, Frank Yellin*». В електронному вигляді вона знаходиться за адресою <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSSpecTOC.doc.html>.

Тут же необхідно відзначити книгу "батька" технології Java Джеймса Гослінга, написану разом з Кеном Арнольдом. Є російський переклад Гослінг Дж., Арнольд К. Язык программирования Java: Пер. с англ. — СПб.: Пітер, 1997. — 304 с.: ил.

Компанія SUN Microsystems має на своєму сайті постійно поновлюваний електронний підручник Java Tutorial, розміром уже більше 14 Мбайт: <http://java.sun.com/docs/books/tutorial/>. Час від часу

зяється його друковане видання *The Java Tutorial, Second Edition: Object-Oriented Programming for the Internet. Mary Campione, Kathy Walrath.*

Повне описання Java API міститься в документації, але є друковане видання *The Java Application Programming Interface. James Gosling, Frank Yellin and the Java Team, Volume 1: Core Packages; Volume 2: Window Toolkit and Applets.*

Програмування в Java

Урок 2. Базовий курс Java

Реквізити: Текст лекції і опис класу System

2.1. Вступ

Приступаючи до вивчення нової мови, корисно поцікавитися, які вихідні дані можуть опрацьовуватися засобами цієї мови, в якому вигляді їх можна задавати і які стандартні методи опрацювання цих даних закладені в мову. Це досить нудне заняття, тому що в кожній розвиненій мові програмування багато типів даних і ще більше методів їх використання. Однак невиконання цих правил приводить до появи скритих помилок, виявити які часто буває дуже важко. Але в кожному ремеслі спочатку приходиться "грати гами", і ми не можемо цього уникнути.

Всі правила мови **Java** вичерпно викладені в її специфікації, яка скорочено називається **JLS**. Інколи, щоб зрозуміти, як виконується та чи інша конструкція мови **Java**, приходиться звертатися до специфікації, але, на щастя, це буває рідко, правила мови **Java** досить прості і той, хто має досвід програмування в інших мовах, легко їх освоїть.

В цій главі приведені примітивні типи даних, операції над ними, оператори управління, показані "підводні камені", яких треба уникати при їх використанні. Але почнемо, по традиції, з найпростішої програми.

2.2. Перша програма на Java

Java програма існує у вигляді класу. Починається службовим словом **class**, за яким слідує ім'я класу, а далі, в фігурних дужках, тіло класу – поля змінних і методи. Ось наша перша **Java** програма:

```
class Zero {}
```

Програма може бути написана в будь-якому текстовому редакторі, наприклад **Notepad**. Потім її треба зберегти у файлі, ім'я якого співпадає з іменем класу і з розширенням **.java**, в даному випадку **Zero.java**. Потім програму треба скомпілювати, що в даному випадку означає переведення її в байт коди. Для цього треба мати набір файлів **JDK – Java Developer Kit**, який можна безкоштовно зкачати з Інтернет або зкопіювати з іншого комп'ютера. У мене ця папка знаходиться на диску **C**. Щоб спростити компіляцію і запуск програми, я збережу цей файл також на диску **C**. На вашому комп'ютері версія **JDK** може бути відмінною від моєї (вони обновлюються майже щорічно). Те ж саме стосується і її місця знаходження на диску, тому треба при потребі внести корективи у нижче наведену адресу папки **bin**. Знаходимо в комп'ютері програму **Командна строка** і відкриваємо її. Ось як було на моєму комп'ютері.

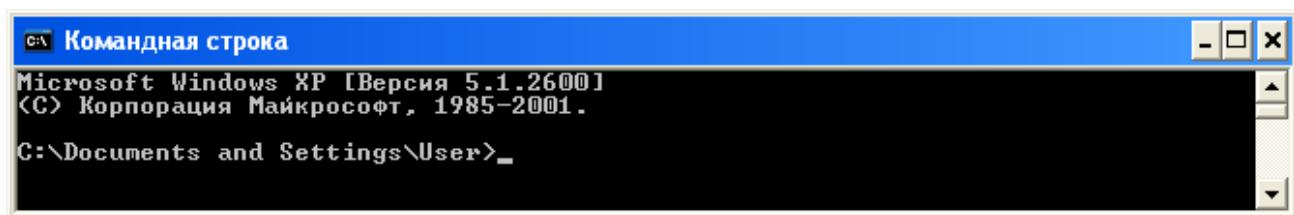


Рис. 2.1. Програма Командна строка

Нам треба повернутися до корінної директорії **C**, де знаходиться **JDK**. Для цього набираємо в командному рядку **cd C:** і натискаємо **Enter** (**cd – change directory**). Результат ви бачите нижче.

Рис. 2.2. Програма Командна строка готова працювати з файлами диску С

Тепер викликаємо компілятор **javac**, що знаходиться в папці **bin** і вказуємо йому файл для компіляції. Тиснемо **Enter** і компілятор створить файл з байт-кодами, дасть йому ім'я **Zero.class** і запише цей файл в поточний каталог – перевірте. Якщо все пройшло як треба, **Командна строка** повернеться до корінної директорії **C**.

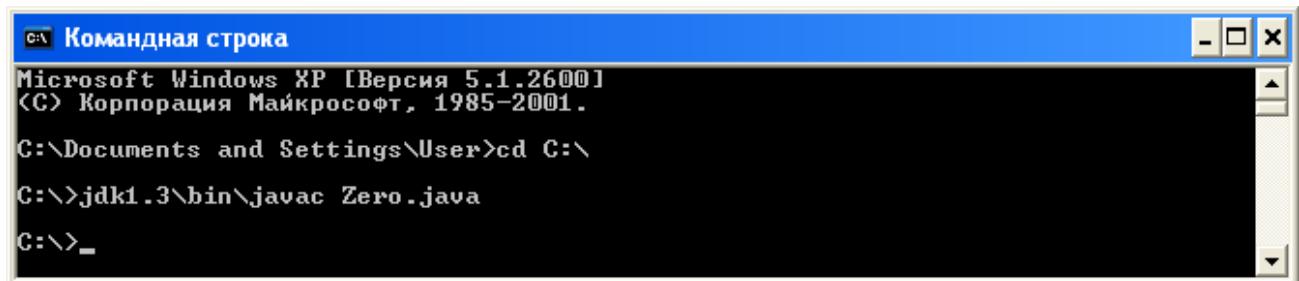


Рис. 2.3. Програма скомпільована

Залишилось викликати інтерпретатор **java (JVM)**, передавши йому в якості аргумента ім'я класу (а не файла). Після натиску **Enter** нас чекає перше розчарування - **JVM** не змогла запустити програму на виконання і сповістила чому процес (**thread**) виключився (**exception**). В нашому класі відсутній стартер – метод **main()**.

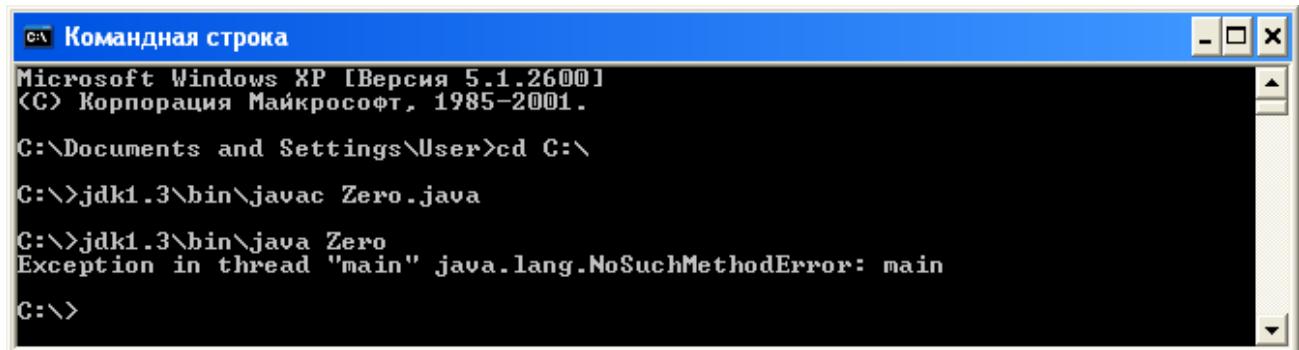


Рис. 2.4. Невдача спроба запустити програму на виконання

Ситуація аналогічна з **C++**, але трохи складніша. Попробуємо її відправити.

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

Чому крім знайомого нам **void** перед **main()** з'явилася ще й **public static**, а в аргументі **String[] args** я поясню пізніше, а поки що візьміть це за правило. Перекомпілюйте файл і запустіть програму заново.

```

Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\User>Cd C:\

C:\>jdk1.3\bin\javac Zero.java

C:\>jdk1.3\bin\java Zero

C:\>

```

Рис. 2.5. Програма спрацювала

Цього разу все получилося, хоча програма і спрацювала вхолосту. Тепер заставимо її зробити щось корисне.

2.3. Перша повноцінна програма на мові Java

По давній традиції, що започаткована в мові С, підручники по мовах програмування починаються з програмами, яка виводить на екран привітання "Hello, World!". Не будемо порушувати цю традицію.

Лістинг 2.1. Перша повноцінна програма на мові Java

```

class Zero{
public static void main(String[] args){
System.out.println("Hello, World!");
}
}

```

Перекомпілюйте цю програму і запустіть на виконання. Цього разу все повинно вийти як треба.

```

Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\User>cd C:\

C:\>jdk1.3\bin\javac Zero.java

C:\>jdk1.3\bin\java Zero
Hello, World!

C:\>_

```

Рис. 2.6. Перша результативна Java програма

На цьому простому прикладі можна помітити ряд суттєвих особливостей мови Java.

- Всяка програма являє собою один або декілька класів, в цьому найпростішому прикладі тільки один *клас* (*class*).
- Початок класу позначається службовим словом *class*, за яким іде ім'я класу, що вибирається довільно, в даному випадку *Zero*. Все, що міститься в класі, записується в фігурних дужках і складає *тіло класу* (*class body*).

- Всі дії виконуються за допомогою методів обробки інформації, коротко говорять просто *метод* (*method*). Цей термін вживається в мові **Java** замість назви "функція", що використовується в інших мовах.
- Методи розрізняються по іменах. Один із методів обовязково повинен називатися **main**, з нього починається виконання програми. В нашій найпростішій програмі тільки один метод, а значить, ім'я йому **main**.
- Як і положено функції, метод завжди видає в результат (частіше говорять, повертає (*returns*)) тільки одне значення, тип якого обовязково указується перед іменем метода. Метод може і не повертати ніякого значення, виконуючи роль процедури, як у нашому випадку. Тоді замість типу значення записується слово **void**, як це і зроблено у прикладі.
- Після імені метода в дужках, через кому, перечисляються *аргументи* (*arguments*) - або *параметри* метода. Для кожного аргумента указується його тип і, через пробіл, ім'я. В прикладі тільки один аргумент, його тип — масив, що складається з рядків символів. Рядок символів — це вбудований в **Java API** тип **String**, а квадратні дужки — ознака масива. Ім'я масива може бути довільним, в прикладі вибрано ім'я **args**.
- Перед типом значення, що повертається методом, можуть бути записані *модифікатори* (*modifiers*). В прикладі їх два: слово **public** означає, що цей метод доступний звідсіль; слово **static** забезпечує можливість виклику метода **main ()** в самому початку виконання програми. Модифікатори взагалі необовязкові, але для метода **main ()** вони необхідні.

Зауваження.

В тексті після імені метода ставляться дужки, щоб підкреслити, що це є ім'я метода, а не просто змінної.

- Все, що містить метод, *тіло метода* (*method body*), записується в фігурних дужках.

Єдину дію, яку виконує метод **main ()** в прикладі, заключається у виклику іншого метода зі складним іменем **System.out.println()** і передачі йому на опрацювання одного аргумента, текстової константи "**Hello, World!**". Текстові константи записуються в лапках, які являються тільки обмежувачами і не входять в склад текста.

Складне ім'я **System.out.println()** означає, що в класі **System**, який входить в **Java API**, визначається змінна з іменем **out**, котра містить екземпляр одного із класів **Java API**, класа **PrintStream**. В ньому є метод **println()**. Все це стане ясно пізніше, а поки що просто будемо писати це довге ім'я.

Дія метода **println ()** заключається у виведенні свого аргумента у вихідний потік, пов'язаний, як правило, з виведенням на екран текстового термінала, у вікно **MS-DOS Prompt** або **Command Prompt**, в залежності від вашої системи. Після виведення курсор переходить на початок наступного рядка екрана, на що указує закінчення **ln**, слово **println** — скорочення слів **print line**. У складі обєкта **out** є і метод **print ()**, що залишає курсор в кінці виведеного рядка. Зрозуміло, це прямий вплив мови **Pascal**.

Відкрийте опис класу **System** в однійменному файлі папки **Java2** і ознайомтеся з іншими методами класу і його полів, в першу чергу поля **out**.

Зробимо зразу важливе зауваження. Мова **Java** розрізняє загальні і прописні літери, імена **main**, **Main**, **MAIN** різні з "точки зору" компілятора **Java**. В прикладі важливо писати **String**, **System** з заглавної літери, а **main** з маленької. Але всередині текстової константи не має значення, писати **World** чи **world**, компілятор взагалі не "дивиться" на неї, різниця буде помітна лише на екрані.

Зауваження

Мова **Java** розрізняє прописні і заглавні літери.

Свої імена можна записувати як завгодно, можна було б дати класу ім'я **helloworld** чи **Helloworld**, але між **Java**-програмістами заключено догвір під іменем "[Code Conventions for the Java Programming Language](http://java.sun.com/docs/codeconv/index.html)", який можна знайти за адресою <http://java.sun.com/docs/codeconv/index.html>. Ось декілька пунктів

цього договора:

- імена класів починаються з заглавної літери; якщо ім'я містить декілька слів, то кожне слово починається із заглавної літери;
- імена методів і змінних починаються з прописної літери; якщо ім'я містить декілька слів, то кожне наступне слово теж починається з прописної літери;
- імена констант записуються повністю заглавними літерами; якщо ім'я містить декілька слів, то між ними ставиться знак підкреслювання.

Звичайно, ці правила необов'язкові, хоча вони і входять в [JLS](#), п. 6.8, але дуже полегшують розуміння кода і надають програмі характерний для [Java](#) стиль.

Стиль визначають не лише імена, але і розташування тексту програми по рядках, наприклад, розташування фігурних дужок: чи залишати відкриваючу дужку в кінці рядка з заголовком класа або метода чи переносити на наступний рядок? Чому це дрібне питання викликає запеклі суперечки, деякі засоби розробки, наприклад [JBuilder](#), навіть пропонують вибрати певний стиль розташування фігурних дужок. Деякі фірми встановлюють свій, внутріфірмений стиль. Ми постараемся слідувати стилю "[Code Conventions](#)" і в тому, що стосується розбиття текста програми на рядки (компілятор же розглядає всю програму як один довгий рядок, для нього програма — це просто послідовність символів), і в тому, що стосується відступів ([indent](#)) у тексті.

[Порада](#)

Називайте файл з програмою іменем класа, що містить метод [main \(\)](#), дотримуйтесь реєстру літер.

[Зауваження:](#)

Не указуйте розширення [class](#) при виклику інтерпретатора.

[2.4. Коментарі:](#)

В тексті програми можна вставити коментарі, котрі компілятор не буде враховувати. Вони дуже корисні для пояснення по ходу програми. В період налаштування можна виключати із програми один або декілька операторів, помітивши їх символами коментаря, як говорять програмісти, "закоментувавши" їх. Коментарі вводяться таким чином:

- за двома похилими рисками підряд [//](#), без пробілу між ними, починається коментар, що продовжується до кінця рядка;
- за похилою рискою і зірочкою [/*](#) починається коментар, котрий може займати декілька рядків, до зірочки і похилої риски [*/](#) (без пробілів між цими знаками).

Коментарі дуже зручні для читання і розуміння коду, вони перетворюють програму в документ, що описує її дії. Програму з хорошими коментарями називають [самодокументованою](#). Тому в [Java](#) введені коментарі третього типу, а в склад [JDK](#) — програму [javadoc](#), що поміщає ці коментарі в окремі файли формату [HTML](#) і створює гіперпосилання між ними: за похилою рискою і двома зірочками підряд, без пробілів, [/**](#) починається коментар, котрий може займати декілька рядків до зірочки з однією похилою рискою [*/](#) і опрацьовується програмою [javadoc](#). В такий коментар можна вставити вказівки програмі [javadoc](#), котрі починаються символом [@](#). Якраз так створюється документація в [JDK](#). Додамо коментарі до нашого прикладу (лістинг 1.2).

[Лістинг 2.2. Перша програма з коментарями](#)

```
class HelloWorld{
 /**
 * Пояснення змісту і особливостей програми...
 * @author Ім'я Прізвище (автора)
```

```

* @version 1.0 (це версія програми)
*/
// HelloWorld – це лише ім'я
// Наступний метод починає виконання програми
public static void main(String[] args){ // args не використовується
/* Наступний метод просто виводить свій аргумент на екран дисплея */
System.out.println("Hello, 21st Century World!");
// наступний виклик закоментований, метод не буде виконуватися
// System.out.println("Farewell, 20th Century!");
}
}

```

Зірочки на початку строчки ніякої ролі не відіграють, вони лише допомагають нам слідкувати за коментарем. Приклад, звичайно, перевантажений поясненнями (це поганий стиль), тут просто показані різні форми коментарів.

2.5. Константи

В мові **Java** можна записувати константи різних типів у різних виглядах. Перечислимо їх.

2.5.1. Цілі константи

Цілі константи можна записувати в трьох системах числення:

- в десятичній формі: +5, -7, 12345678;
- у восьмеричній формі, починаючи з нуля: 027, -0326, 0777; в запису таких констант недопустимі цифри 8 і 9;

Зауваження

Число, що починається з нуля, записано в восьмеричній формі, а не в десятковій.

- в шістнадцятирічній формі, починаючи з нуля і латинської літери x або X: 0xff0a, 0xFC2D, 0x45a8, 0X77FF; тут заглавні і прописні літери не розрізняються.

Цілі константи зберігаються у форматі типу **int** (див. нижче).

В кінці цілої константи можна записати літеру заглавну L або прописну l, тоді константа буде зберігатися в довгому форматі типу **long** (див. нижче): +25L, -0371, 0xffL, 0xDFDF1.

Порада

Не використовуйте при запису довгих цілих констант заглавну латинську літеру l, її легко переплутати з одицією.

2.5.2. Дійсні константи

Дійсні константи записуються лише в десятковій системі числення в двох формах:

- з фіксованою точкою: 37.25, -128.678967, +27.035;
- з плаваючою точкою: 2.5e34, -0.345e-25, 37.2E+4; можна писати заглавну або прописну латинську літеру E; пробіли і дужки недопустимі.

В кінці дійсної константи можна поставити літеру F або f, тоді константа буде зберігатися в форматі типу **float** (див. нижче): 3.5f, -45.67F, 4.7e-5f. Можна приписати і літеру D (або d): 0.045D, -456.77889d, що

означає тип `double`, але це вже зайве, оскільки дійсні константи і так зберігаються в форматі типа `double`.

2.5.3. Символьні константи

Для запису одиноких символів використовуються наступні форми.

- Друковані символи можна записувати в апострофах: 'a', 'N', '?'. Замініть у програмі `HelloWorld` команду

`System.out.println("Hello, World!");` на

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

і випробуйте її. Буде надрукована літера Н. Якщо ж ви напишете

`System.out.println('Hello, World!');`

то компілятор відмовиться працювати з такою програмою.

Керуючі символи записуються в апострофах з оберненою похилою рискою:

- '\n' — символ переводу рядка `newline` з кодом ASCII 10;
- '\r' — символ повернення каретки `CR` з кодом 13;
- '\f' — символ переводу сторінки `FF` з кодом 12;
- '\b' — символ повернення на крок `BS` з кодом 8;
- '\t' — символ горизонтальної табуляції `HT` з кодом 9;
- '\\' — обернена похила риска;
- '\"' — лапка;
- '\"' — апостроф.
- Код будь-якого символу з десятичним кодуванням від 0 до 255 можна задати, записавши його не більше ніж трьома цифрами у восьмеричній системі числення в апострофах після оберненої похилої риски: '123' — буква `S`, '346' — буква `ц`. Не рекомендується використовувати цю форму запису для друкованих і керуючих символів, перечислених у попередньому пункті, оскільки компілятор зараз же переведе восьмеричний запис у вказану вище форму. Найбільший код '\377' — десяткове число 255.
- Код будь-якого символу в кодуванні Unicode набирається в апострофах після оберненої похилої риски і латинської літери `u` рівно чотирма шістнадцятирічними цифрами: '\u0053' — буква `S`, '\u0416' — знак питання `?`.

Випробуйте програму з командами

`System.out.println('\123'); System.out.println('\346'); System.out.println('\u0053'); System.out.println('\u0416');`

Символи зберігаються в форматі типу `char` (див. нижче).

Примітка

Заглавні російські літери в кодуванні Unicode займають діапазон від '\u0410' — заглавна літера А, до '\u042F' — заглавна Я, прописні літери '\u0430' — а, до '\u044F' — я.

В якій би формі не записувалися символи, компілятор переводить їх в Unicode, включаючи і вихідний текст програми.

Зауваження. Компілятор і виконуюча система `Java` працюють тільки з кодуванням `Unicode`.

Порада. Користуйтесь `Unicode` напряму лише у крайніх випадках

2.5.4. Рядкові константи

Рядки символів поміщаються в лапки. Керуючі символи і коди записуються в рядках точно так же, з оберненою похилою рискою, але без апострофів, і викликають ті ж самі дії. Рядки можуть розташовуватися лише в одному рядку вихідного кода, не можна відкриваючі лапки поставити в одному рядку, а закриваючі — в наступному.

Ось декілька прикладів:

"Цей рядок\nз переносом"

"\"Динамо\" — Чемпіон!"

Примітка

Рядок символів не можна починати в одному рядку вихідного кода, а закінчувати в іншому.

Для рядкових констант визначена операція зєднання, позначається плюсом.

"Зєднання" + "рядків" дає в результаті рядок "Зєднання рядків".

2.6. Імена

Імена (names) змінних, класів, методів і інших обєктів можуть бути простими (загальна назва — ідентифікатори (identifiers)) і складними (qualified names). Ідентифікатори в Java складаються з так званих літер Java (Java letters) і арабських цифр 0 - 9, причому першим символом ідентифікатора не може бути цифра. (Дійсно, як розуміти запис 2e3: як число 2000,0 чи як імя змінної?) В число літер Java обов'язково входять прописні і заглавні латинські літери, знак долара \$ і знак 'підкреслювання _', а також символи національних алфавітів.

Зауваження

Не вживайте в іменах знак долара. Компілятор Java використовує його для запису імен вкладених класів.

Ось приклади правильних ідентифікаторів:

`a1 , my_var, var3_5, _var, veryLongVarName, aName, theName, a2Vh36kBnMt456dx`

В іменах краще не вживати прописну літеру I, котру легко сплутати з одиницею, і літеру O, котру легко прийняти за нуль.

Не забувайте про рекомендації "Code Conventions".

В класі `Character`, що входить до складу Java API, є два методи, що перевіряють, чи придатний даний символ для використання в ідентифікаторі: `isJavaIdentifierStart()`, перевіряє, чи являється символ літерою Java, придатною для першої літери ідентифікатора і `isJavaIdentifierPart()`, що виясняє, чи можна взагалі вживати цей символ в ідентифікаторі. Випробуйте наступні дві програми і ви побачите який із ASCII символів на що придатний.

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 256; i++)
            if (Character.isJavaIdentifierStart((char)i))
```

```

        System.out.println(Integer.toString(i) + " " + (char)i + " is Java
Identifier Start Symbol" );
        else
        System.out.println(Integer.toString(i) + " " + (char)i + " isn't Java
Identifier Start Symbol" );
    }
}
i

class Test {
public static void main(String[] args) {
for (int i = 0; i < 256; i++)
{ if (Character.isJavaIdentifierPart((char)i))

    System.out.println(Integer.toString(i) + " " + (char)i + " is Java Identifier
Part Symbol" );
else
    System.out.println(Integer.toString(i) + " " + (char)i + " isn't Java Identifier
Part Symbol" );
}
}
}

```

Службові слова Java, такі як `class`, `void`, `static`, зарезервовані, їх не можна використовувати в якості ідентифікаторів.

Складне імя (*qualified name*) — це декілька ідентифікаторів, розділених точками, без пробілів, наприклад, нам уже зустрічалося ім'я `System.out.println`.

Порада. Ознайомтеся з іншими методами класу `Character` по одноіменному файлу.

2.7. Примітивні типи даних і операції

2.7.1. Загальна класифікація

Всі типи вихідних даних, вбудованих в мову Java, діляться на дві групи: **примітивні типи** (*primitive types*) і посилочні **типи** (*reference types*).

Посилочні типи діляться на **масиви** (*arrays*), **класи** (*classes*) і **інтерфейси** (*interfaces*).

Примітивних типів всього вісім. Їх можна розділити на **логічний** (інколи говорять **булів**) тип `boolean` і **числові** (*numeric*).

До числових типів відносяться **цілі** (*integer*) і **дійсні** (*floating-point*) типи.

Цілих типів п'ять: `byte`, `short`, `int`, `long`, `char`.

Символи можна використовувати всюди, де вживається тип `int`, тому JLS причисляє їх до цілих типів. Наприклад, їх можна використовувати в арифметичних обчисленнях, скажімо, можна написати `2 + 'b'`, до двійки буде доданий ASCII код 98 літери 'b' і в результаті додавання одержимо 100.

Нагадаємо, що в записі `2 + "b"` плюс розуміється як зєднання рядків, двійка буде перетворена в рядок, в результаті одержимо рядок `"2b"`.

Дійсних типів два: `float` і `double`. Оскільки по імені змінної неможливо визначити її тип, всі змінні обов'язково повинні бути описані перед їх використанням. Описання полягає в тому, що записується ім'я типу, потім, через пробіл, список імен змінних, розділених комою. Для всіх або деяких змінних можна вказати початкові значення після знака рівності, котрими можуть бути будь-які константні вирази того ж типа. Описання кожного типа закінчується точкою з комою. В програмі може бути скільки завгодно описань кожного типа.

Зауваження для спеціалістів

Java — мова зі строгою типізацією ([strongly typed language](#)).

Розберемо кожний тип детальніше.

2.7.2. Логічний тип

Значення логічного типа `boolean` виникають в результаті різних порівнянь, вроді `2 > 3`, і використовуються, головним чином, в умовних операторах і операторах циклів. Логічних значень всього два: `true` (істина) і `false` (хиба). Це службові слова `Java`. Описання змінних цього типа виглядає так:

```
boolean b = true, bb = false, bool2;
```

Над логічними даними можна виконувати операції присвоювання, наприклад, `bool2 = true`, в тому числі й сумісні з логічними операціями; порівняння на рівність `b == bb` і на нерівність `b != bb`, а також логічні операції.

2.7.3. Логічні операції

Логічні операції:

- заперечення (**NOT**) `!` (позначається знаком оклику);
- кон'юнкція (**AND**) `&` (амперсанд);
- диз'юнкція (**OR**) `|` (вертикальна риска);
- виключне АБО (**XOR**) `^` (каре).

Вони виконуються над логічними даними, їх результатом буде також логічне значення `true` або `false`. Нагадаємо таблицю логічних операцій.

Таблиця 1.1. Логічні операції

b1	b2	!b1	b1&b2	b1 b2	b1^b2
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Словами ці правила можна виразити так:

- заперечення змінює значення істинності;

- конюнкція істинна, тільки якщо обидва операнди істинні;
- дизюнкція хибна, тільки якщо обидва операнди хибні;
- виключне АБО істинно, тільки якщо значення операндів різні.

Зауваження

Якби Шекспір був програмістом, то фразу "To be or not to be" він написав би так: `2b | ! 2b`.

Крім перечислених чотирьох логічних операцій єсть ще дві логічні операції скороченого обчислення:

- скорочена конюнкція ([conditional-AND](#)) `&&`;
- скорочена дизюнкція ([conditional-OR](#)) `||`.

Здвоєні знаки амперсанда і вертикальної риски слід записувати без пробілів.

Правий operand скорочених операцій обчислюється тільки в тому випадку, якщо від нього залежить результат операції, тобто якщо лівий operand конюнкції має значення `true`, або лівий operand дизюнкції має значення `false`.

Це правило дуже зручне і вправно застосовується, наприклад, можна записувати вирази `(n != 0) && (m/n > 0.001)` або `(n == 0) || (m/n > 0.001)` не боячись ділення на нуль.

Зауваження

Практично завжди в Java використовуються якраз скорочені логічні операції.

2.7.4. Цілі типи

Специфікація мови [Java](#), [JLS](#), визначає розрядність (кількість байтів, що відводяться для зберігання значень типу в оперативній памяті) і діапазон значень кожного типу. Для цілих типів вони наведені в табл. 1.2.

Таблиця 1.2. Цілі типи

Тип	Розрядність (байт)	Діапазон
<code>byte</code>	1	от -128 до 127
<code>short</code>	2	от -32768 до 32767
<code>int</code>	4	от -2147483648 до 2147483647
<code>long</code>	8	от -9223372036854775808 до 9223372036854775807
<code>char</code>	2	від '\u0000' до '\uFFFF', в десятковій формі від 0 до 65535

Між іншим, для Java розрядність не стільки важлива, на деяких комп'ютерах вона може відрізнятися від наведеної в таблиці, а от діапазон значень повинен витримуватись беззастережно.

Хоча тип `char` займає два байти, в арифметичних обчисленнях він працює як тип `int`, йому виділяється 4 байти, два старших байти заповнюються нулями.

Приклади визначення змінних цілих типів:

```
byte b1 = 50, b2 = -99, b3;
short det = 0, ind = 1;
int i = -100, j = 100, k = 9999;
long big = 50, veryBig = 2147483648L;
char c1 = 'A', c2 = '?', newLine = '\n';
```

Цілі типи зберігаються в двійковому вигляді з додатковим кодом. Останнє означає, що для від'ємних чисел зберігається не їх двійкове представлення, а **доповняльний код** цього двійкового представлення.

Доповняльний же код отримують так: в двійковому представленні всі нулі замінюються на одиниці, а одиниці на нулі, після чого до результату прибавляється одиниця, розуміється, в двійковій арифметиці.

Наприклад, значення 50 змінної `b1`, визнаної вище, буде зберігатися в одному байті з вмістом 00110010, а значення -99 змінної `b2` — в байті із вмістом, котрий обчислюємо так: число 99 переводимо в двійкову форму, одержимо 01100011, міняємо одиниці з нулями, одержуємо 10011100, і прибавляємо одиницю, одержавши нарешті байт із вмістом 10011101.

Зміст всіх цих складностей в тому, що додавання числа з його доповняльним кодом в двійковій арифметиці дасть в результаті нуль, старший біт просто втрачається. Це означає, що в такій дивній арифметиці доповняльний код числа являється протилежним йому числом, числом з оберненим знаком. А це, в свою чергу, означає, що замість того, щоб відняти від числа А число В, можна до А прибавить доповняльний код числа В. Таким чином, операція віднімання виключається із набору машинних операцій.

Над цілими типами можна виконувати масу операцій. Їх набір визначився в мові С, він виявився зручним і кочує з мови в мову майже без змін. Особливості застосування цих операцій в мові `Java` показані на прикладах.

2.7.5. Операції над цілими типами

Всі операції, котрі виконуються над цілими числами, можна розділити на наступні групи.

2.7.5.1. Арифметичні операції

До арифметичних операцій відносяться:

- додавання `+` (плюс);
- віднімання `-` (дефіс);
- множення `*` (зірочка);
- ділення `/` (похила риска — слеш);
- остача від ділення (ділення по модулю) `%` (процент);
- інкремент (збільшення на одиницю) `++`;
- декремент (зменшення на одиницю) `--`

Між спареним плюсами і мінусами не можна залишати пробіли. Додавання, віднімання і множення цілих значень виконується як звичайно, а от ділення цілих значень в результаті дає знову ціле (так зване "ціле ділення"), наприклад, $5/2$ дасть в результаті 2, а не 2.5, а $5/(-3)$ дасть -1. Дробова частина попросту відкидається, відбувається урізання частки. Це спочатку дивує, а потім виявляється зручним для урізання чисел.

Зауваження

В Java прийнято ціличисельне ділення.

Це дивне для математики правило природне для програмування: якщо обидва операнди мають один і той же тип, то і результат має той же тип. Досить написати $5/2.0$ або $5.0/2$ або $5.0/2.0$ і одержимо 2.5 як результат ділення дійсних чисел.

Операція **ділення по модулю визначається** так: $a \% b = a - (a / b) * b$; наприклад, $5 \% 2$ дасть в результаті 1 , а $5 \% (-3)$ дасть 2 , тому що $5 = (-3) * (-1) + 2$, але $(-5) \% 3$ дасть -2 , оскільки $-5 = 3 * (-1) - 2$.

Операції **інкремент і декремент** означають збільшення або зменшення значення змінної на одиницю і застосовуються тільки для змінних, а не для констант чи виразів, не можна написати $5++$ або $(a + b)++$.

Наприклад, після приведених вище описань $i++$ дасть -99 , а $j--$ дасть 99 .

Цікаво, що ці операції можна записати і перед змінною: **$++i$, $--j$** . При першій формі запису (**постфіксній**) у виразі приймає участь старе значення змінної і лише потім відбувається збільшення чи зменшення її значення. При другій формі запису (**префіксній**) спочатку зміниться змінна і її нове значення буде приймати участь у виразі.

Наприклад, після приведених вище описань, $(k++) + 5$ дасть в результаті 10004 , а змінна k прийме значення 10000 . Але в тій же ситуації $(++k) + 5$ дасть 10005 , а змінна k стане рівною 10000 .

2.7.5.2. Приведення типів

Результат арифметичної операції має тип **int**, крім того випадку, коли один із операндів є типу **long**. В цьому випадку результат буде типу **long**.

Перед виконанням арифметичної операції завжди відбувається **підвищення** (**promotion**) типів **byte**, **short**, **char**. Вони перетворюються в тип **int**, а може бути, і в тип **long**, якщо інший операнд має тип **long**. Операнд типу **int** підвищується до типу **long**, якщо інший операнд є типу **long**. Звичайно, числове значення операнда при цьому не змінюється.

Це правило приводить інколи до неочікуваних результатів. Спроба скомпілювати просту програму, представлена в лістингі 1.3, приведе до повідомлення компілятора про помилку. Перевірте!

Лістинг 2.3. Невірне визначення змінної

```
class InvalidDef{
public static void main (String [] args) {
byte b1 = 50, b2 = -99;
short k = b1 + b2; // Невірно!
System.out.println("k=" + k);
}
}
```

Це повідомлення означає, що в файлі **InvalidDef.java**, в рядку 4, виявлена можлива втрата точності (**possible loss of precision**). Потім приводиться виявлений (**found**) і потрібний (**required**) типи, виводиться рядок, в якому виявлена (а не зроблена) помилка, і відмічається символ, при аналізі якого знайдена помилка. В кінці вказано загальне число виявлених (а не зроблених) помилок (1 error).

В таких випадках треба виконувати явне приведення типу. В даному випадку це буде **зуження** (**narrowing**) типа **int** до типа **short**. Воно здійснюється операцією явного приведення, котра записується перед потрібним значенням у вигляді імені типа в дужках. Визначення

```
short k = (short)(b1 + b2);
```

буде вірним.

Звуження відбувається просто відкиданням старших бітів, що необхідно враховувати для великих значень. Наприклад, визначення

```
byte b = (byte) 300;
```

дасть змінній **b** значення 44. Дійсно, в двійковому представленні числа 300, рівному 100101100, відкидається старший біт і получается 00101100. Таким же чином можна привести і явне *розширення (widening)* типу, якщо в цьому єсть необхідність.

Якщо результат цілої операції виходить за діапазон свого типу **int** або **long**, то автоматично відбувається приведення по модулю, рівному довжині цього діапазона, і обчислення продовжуються, переповнення ніяк не відмічається.

Зауваження

В мові Java немає цілочисленого переповнення.

2.7.5.3. Операції порівняння

В мові **Java** єсть шість звичайних операцій порівняння цілих чисел по величині:

- більше **>**;
- менше **<**;
- більше або дорівнює **>=**;
- менше або дорівнює **<=**;
- рівно **==**;
- не рівно **!=**.

Спарені символи записуються без пробілів, їх не можна переставляти місцями, запис **=>** буде невірним. Результат порівняння — логічне значення: **true**, в результаті, наприклад, порівняння **3 != 5**; або **false**, наприклад, в результаті порівняння **3 == 5**.

Для запису складних порівнянь слід застосовувати логічні операції. Наприклад, в обчисленнях часто приходиться робити перевірку типу **a < x < b**. Подібний запис в мові **Java** приведе до повідомлення про помилку, оскільки перше порівняння, **a < x**, дасть **true** або **false**, а **Java** не знає, більше це, ніж **b**, чи менше. В даному випадку слід написати вираз **(a < x) && (x < b)**, причому тут дужки можна опустити, написати просто **a < x && x < b**, але про це трохи пізніше.

2.7.5.4. Побітові операції

Інколи приходиться змінювати значення окремих бітів в цілих даних. Це виконується за допомогою побітових (**bitwise**) операцій шляхом накладання маски. В мові **Java** єсть чотири побітові операції:

- доповнення (**complement**) **~** (тильда);
- побітова конюнкція (**bitwise AND**) **&**;
- побітова дизюнкція (**bitwise OR**) **|**;
- побітове виключне АБО (**bitwise XOR**) **^**.

Вони виконуються порозрядно, після того як обидва операнди будуть приведені до одного типу **int** або **long**, так же як і для арифметичних операцій, а значить, і до однієї розрядності. Операції над кожною парою бітів виконуються згідно табл. 1.3.

В нашому прикладі $b1 == 50$, двійкове представлення 00110010, $b2 == -99$, двійкове представлення 10011101. Перед операцією відбувається підвищення до типу [int](#). Одержано представлення із 32-х розрядів для $b1 — 0...00110010$, для $b2 — 1...10011101$. В результаті побітових операцій одержимо:

- $\sim b2 == 98$, двійкове представлення 0...01100010;
- $b1 \& b2 == 16$, двійкове представлення 0...00010000;
- $b1 | b2 == -65$, двійкове представлення 1...10111111;
- $b1 ^ b2 == -81$, двійкове представлення 1...10101111.

Двійкове представлення кожного результату займає 32 біти.

Зверніть увагу, що доповнення $\sim x$ завжди еквівалентно $(-x) - 1$.

Таблиця 1.3. Побітові операції

n1	n2	$\sim n1$	$n1 \& n2$	$n1 n2$	$n1 ^ n2$
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Те, що побітові операції мають ті ж самі позначення, що і логічні операції, не повинно викликати непорозуміння. Перші виконуються над цілими числами в двійковій формі, а другі над булевими змінними.

2.7.5.5. Зсуви

В мові Java є три операції зсуву двійкових розрядів:

- зсув вліво `<<`;
- зсув вправо `>>`;
- беззнаковий зсув вправо `>>>`.

Ці операції своєрідні тим, що лівий і правий операнди в них мають різний зміст. Зліва стоїть значення цілого типу, а права частина показує, на скільки двійкових розрядів зсувається значення, що стоїть в лівій частині.

Наприклад, операція $b1 << 2$ зсуне вліво на 2 розряди попередньо підвищене значення 0...00110010 змінної $b1$, що дасть в результаті 0...01100100, десяткове 200. Звільнені справа розряди заповнюються нулями, ліві розряди, що знаходяться за 32-м бітом, втрачаються.

Операція $b2 << 2$ зсуне підвищене значення 1...10011101 на два розряди вліво. В результаті одержимо 1...1001110100, десяткове значення -396.

Зверніть увагу, що зсув вліво на n розрядів еквівалентний множенню числа на 2 в степені n .

Операція $b1 >> 2$ дасть в результаті 0...00001100, десяткове 12, а $b2 >> 2$ — результат 1..11100111, десяткове -25, тобто зліва зсувається старший біт, праві біти втрачаються. Це так званий *арифметичний*

зсув.

Операція беззнакового зсуву у всіх випадках ставить зліва на звільнені місця нулі, здійснюючи логічний зсув. Але внаслідок попереднього підвищення це має ефект лише для декількох старших розрядів від'ємних чисел. Так, $b2 >>> 2$ має результатом 001...100111, десяткове число 1 073 741 799.

Якщо ж ми хочемо одержати логічний зсув початкового значення змінної $b2$, тобто, 0...00100111, треба попередньо накласти на $b2$ маску, обнуливши старші біти: $(b2 \& 0xFF) >>> 2$.

Зауваження

Будьте обережні при використанні зсувів вправо. Вам зараз важко уявити, в яких програмах може знадобитися зсув. Але він досить вживаний в програмуванні. Якщо ви знайдете його у якісь програмі, то не полініться розібратися в його механізмі і доцільності застосування.

2.7.6. Дійсні типи

Дійсних типів у Java два: [float](#) і [double](#). Вони характеризуються розрядністю, діапазоном значень і точністю представлення, що відповідає стандарту IEEE 754-1985 з деякими змінами. До звичайних дійсних чисел додаються ще три значення»

1. Додатня нескінченість, що виражається константою [POSITIVE_INFINITY](#). Вона виникає при переповненні додатнього значення, наприклад, в результаті операції множення $3.0*6e307$.
2. Від'ємна нескінченість [NEGATIVE_INFINITY](#).
3. "Не число", що записується константою [NaN \(Not a Number\)](#). Вона виникає при діленні дійсного числа на нуль або при множенні нуля на нескінченість.

Крім того, стандарт розрізняє додатній і від'ємний нуль, що виникають при діленні на нескінченість відповідного знака, хоча порівняння [0.0 == -0.0](#) дає [true](#).

Операції з нескінченістю виконуються по звичайним математичним правилам.

Усьому останньому дійсні типи — це звичайні, дійсні значення, до яких можна застосувати всі арифметичні операції і порівняння, перечислені для цілих типів. Характеристики дійсних типів приведені в табл. 1.4.

Знавцям C/C++

В мові Java остатча від ділення [%](#), інкремент [++](#) і декремент — застосовується і для дійсних чисел.

Таблиця 1.4. Дійсні типи

Тип	Розрядність	Діапазон	Точність
-----	-------------	----------	----------

float	4	$3,4e-38 < x < 3,4e38$	7—8 цифр
double	8	$1,7e-308 < x < 1,7e308$	17 цифр

Приклади визначення дійсних типів:

`float x = 0.001, y = -34.789;`

`double z1 = -16.2305, z2;`

Оскільки до дійсних типів застосовуються всі арифметичні операції і порівняння, цілі і дійсні значення можна змішувати в операціях. При цьому правило приведення типів доповнюється такими умовами:

- якщо в операції один операнд має тип `double`, то і другий приводиться до типу `double`;
- якщо один операнд має тип `float`, то і другий приводиться до типу `float`;
- в протилежному випадку діє правило приведення цілих значень.

2.7.7. Операції присвоювання

Просто операція присвоєння ([simple assignment operator](#)) записується знаком рівності `=`, зліва від котрого стоїть змінна, а справа вираз, сумісний з типом змінної: `x = 3.5, y = 2 * (x - 0.567) / (x + 2), bb = x >= y && b`.

Операція присвоювання діє так: вираз, що стоїть після знака рівності, обчислюється і приводиться до типу змінної, що стоїть зліва від знака рівності. Результатом операції буде приведене значення правої частини.

Операція присвоювання має ще одну, побічну, дію: змінна, що стоїть зліва, одержує приведене значення правої частини, старе її значення втрачається.

В операції присвоювання ліва і права частини нерівноправні, не можна написати `3.5 = x`. Після операції `x = y` змінна `x`, ставши рівною `y`, а після `y = x` зміниться `y`.

Крім простої операції присвоювання є ще 11 складних операцій присвоювання ([compound assignment operators](#)): `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=`. Символи записуються без пробілів, не можна переставляти їх місцями.

Всі складні операції присвоювання діють по одній схемі:

`x op= a` еквівалентно `x = (тип x)`, тобто `(x op a)`.

Нагадаємо, що змінна `ind` типу `short` визначена у нас із значенням 1. Присвоювання `ind += 7.8` дасть в результаті число 8, те ж саме значення одержить і змінна `ind`. Ця операція еквівалентна простій операції присвоювання `ind = (short)(ind + 7.8)`.

Перед присвоюванням, при необхідності, автоматично відбувається приведення типу. Тому:

```
byte b = 1;
b = b + 10; // Помилка!
```

```
b += 10; // Правильно!
```

Перед додаванням `b + 10` відбувається підвищення `b` до типу `int`, результат додавання теж буде тип `int` і, в першому випадку не може бути присвоєний змінний `b` без явного приведення типу. В другому випадку перед присвоюванням відбудеться звуження результату додавання до типу `byte`.

2.7.8. Умовна операція

Ця своєрідна операція має три операнди. Спочатку записується довільний логічний вираз, тобто такий, що має результата `true` або `false`, потім ставиться знак питання, потім два довільних вирази, розділених двокрапкою, наприклад,

```
x < 0 ? 0 : x
```

```
x > y ? x - y : x + y
```

Умовна операція виконується так. Спочатку обчислюється логічний вираз. Якщо одеждано значення `true`, то обчислюється перший вираз після знака `?` і його значення буде результатом всієї операції. Останній вираз при цьому не обчислюється. Якщо ж отримане значення `false`, то обчислюється тільки останній вираз, його значення буде результатом операції.

Це дозволяє написати `n == 0 ? : m / n` не боячись ділення на нуль. Умовна операція спочатку здається дивною, але вона дуже зручна для запису невеликих розгалужень.

2.8. Вирази

Із констант і змінних, операцій над ними, викликів методів і дужок складаються *вирази* (*expressions*). Розуміється, всі елементи виразу повинні бути сумісними, не можна написати, наприклад, `2 + true`. При обчисленні виразу виконуються чотири правила:

1. Операції одного пріоритету обчислюються зліва направо: $x + y + z$ обчислюється як $(x + y) + z$. Виняток: операції присвоювання виконуються справа наліво: $x = y = z$ обчислюється як $x = (y = z)$.
2. Лівий операнд обчислюється раніше правого.
3. Операнди повністю обчислюються перед виконанням операції.
4. Перед виконанням складної операції присвоювання значення лівої частини зберігається для використання в правій частині.

Наступні приклади показують особливості застосування перших трьох правил. Нехай

```
int a = 3, b = 5;
```

Тоді результатом виразу `b + (b = 3)` буде число 8; але результатом виразу `(b = 3) + b` буде число 6. Вираз `b += (b = 3)` дасть в результаті 8, тому що обчислюється як перший із наведених вище виразів.

Знавцям C/C++

Більшість компіляторів мови C++ у всіх цих випадках вичислять значення 8.

Четверте правило можна продемонструвати так. При тих же визначеннях `a` і `b` в результаті обчислення виразу `b+= a += b += 7` одержимо 20. Хоча операції присвоювання виконуються справа наліво і після першої, правої, операції значення `b` становиться рівним 12, але в останньому, лівому, присвоюванні

приймає участь старе значення **b**, рівне 5. А в результаті двох послідовних обчислень **a += b += 7; b += a;** одержимо 27, оскільки в другому виразі приймає участь уже нове значення змінної **b**, рівне 12.

Знавцям C/C++

Більшість компіляторів C++ в обох випадках обчислять 27.

Вирази можуть мати складний і заплутаний вигляд. В таких випадках виникає питання про пріоритет операцій, про те, які операції будуть виконані в першу чергу. Природно, множення і ділення виконуються раніше додавання і віднімання.

Решта правил перечислені в наступному розділі.

Порядок обчислень виразу завжди можна відрегулювати дужками, їх можна ставити скільки завгодно. Ale тут важливо зберігати "золоту середину". При великій кількості дужок знижується наочність виразу і легко помилитися в розстановці дужок. Якщо вираз з дужками коректний, то компілятор може відслідкувати тільки парність дужок, але не правильність їх розстановки.

2.9. Пріоритет операцій

Операції перечислені в порядку спадання пріоритета. Операції в одному рядку мають одинаковий пріоритет.

1. Постфіксні операції **++ i** **--**
2. Префіксні операції **++ i** **- -**, доповнення **~ i** заперечення **!**.
3. Приведення типу (тип).
4. Множення *****, ділення **/** і знаходження решти **%**.
5. Додавання **+** і віднімання **-**.
6. Здиги **<<**, **>>**, **>>>**.
7. Порівняння **>**, **<**, **>=**, **<=**.
8. Порівняння **==**, **!=**.
9. Побітова конюнкція **&**.
10. Побітове виключне АБО **^**.
11. Побітова дизюнкція **|**.
12. Конюнкція **&&**.
13. Дизюнкція **||**.
14. Умовна операція **? :**.
15. Присвоювання **=**, **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **^=**, **|=**, **<<**, **>>**, **>>>**.

Тут перечислені не всі операції мови Java, список буде доповнюватися по мірі вивчення нових операцій.

2.10. Оператори

Як ви знаєте, будь-який алгоритм призначений для виконання на комп'ютері, можна розробити, використовуючи лише лінійні обчислення, розгалуження і цикли.

Записати його можна в різних формах: у вигляді блок-схеми, на псевдокоді, на звичайній мові, як ми записуємо кулінарні рецепти, або як-небудь ще.

Всяка мова програмування повинна мати свої методи запису алгоритмів. Вони називаються **операторами (statements)** мови. Мінімальній набір операторів повинен містити оператор для запису лінійних обчислень, умовний оператор для запису розгалужень і оператор циклу.

Звичайно склад операторів мови програмування ширше: для зручності записи алгоритмів у мову включаються декілька операторів циклу, оператор варіанта, оператори переходу, оператори описування обєктів.

Набір операторів мови Java включає:

- оператори описування змінних і інших обєктів (вони були розглянуті вище);
- оператори-вирази;
- оператори присвоювання;
- умовний оператор **if**;
- три оператори циклу **while**, **do-while**, **for**;
- оператор варіанта **switch**;
- Оператори переходу **break**, **continue** і **return**;
- блок **{}**;
- пустий оператор — просто крапка з комою.

Тут приведений не весь набір операторів **Java**, він буде доповнюватися по мірі вивчення мови.

Зауваження

В мові **Java** немає оператора **goto**.

Всякий оператор закінчується крапкою з комою.

Можна поставити крапку з комою в кінці любого виразу, і він стане оператором (**expression statement**). Але це має зміст тільки для операцій присвоювання, інкременту, декременту і виклику методів. В решті випадків це не має змісту, тому що обчислене значення виразу буде втрачено.

Знавцям Pascal

Крапка з комою в **Java** не розділяє оператори, а являється частиною оператора.

Лінійне виконання алгоритма забезпечується послідовним записом операторів. Перехід з рядка на рядок у вихідному тексті не має ніякого значення для компілятора, він здійснюється тільки для наочності і читабельності тексту.

2.10.1. Блок

Блок заключає в собі нуль або декілька операторів з метою використовувати їх як один оператор в тих місцях, де по правилах мови можна записати тільки один оператор. Наприклад, **{x = 5; y = ?;}**. Можна записати і пустий блок, просто пару фігурних дужок **{}**.

Блоки операторів часто використовуються для обмеження області дії змінних і просто для легшого читання

тексту програми.

2.10.2. Оператори присвоювання

Крапка з комою в кінці будь-якої операції присвоювання перетворює її в оператор присвоювання. Побічна дія операції — присвоювання — становиться в операторі основним.

Різниця між операцією і оператором присвоювання носить лише теоретичний характер. Присвоювання частіше застосовується як оператор, а не як операція.

2.10.3. Умовний оператор

Умовний оператор ([if-then-else statement](#)) у мові [Java](#) записується так:

`if (логічний вираз) оператор1 else оператор2`

і діє наступним чином. Спочатку обчислюється логічний вираз. Якщо результат [true](#), то діє [оператор1](#) і на цьому дія умовного оператора завершується, [оператор2](#) не діє, далі буде виконуватися наступний за [if](#) оператор. Якщо результат [false](#), то діє [оператор2](#), при цьому [оператор1](#) взагалі не виконується.

Умовний оператор може бути скороченим ([if-then statement](#)):

`if (логвир) оператор1`

і у випадку [false](#) не виконується нічого.

Синтаксис мови не дозволяє записувати декілька операторів ні у гілці [then](#), ні у гілці [else](#). При необхідності створюється блок операторів у фігурних дужках. "[Code Conventions](#)" рекомендує завжди використовувати фігурні дужки і розташовувати оператор в декількох рядках з відступами, як в наступному прикладі:

```
if (a < x) {
    x = a + b; } else {
    x = a - b;
}
```

Це полегшує додавання операторої в кожну гілку при зміні алгоритма. Ми не будемо строго слідувати цьому правилу, щоб не збільшувати розмір тексту.

Дуже часто одним із операторів являється знову умовний оператор, наприклад:

```
if (n == 0) {
    sign = 0;
} else if (n < 0) {
    sign = -1;
} else {
    sign = 1;
}
```

При цьому може виникнути така ситуація ("dangling else"):

```
int ind = 5, x = 100;

if (ind >= 10) if (ind <= 20) x = 0; else x = 1;
```

Збереже змінна [x](#) значення 100 чи стане рівною 1? Тут необхідно вольове рішення, і спільне для

більшості мов, в тому числі і **Java**. Правило таке: гілка **else** відноситься до найближчої зліва умови **if**, що не має своєї гілки **else**. Тому в нашому прикладі змінна **x** залишиться рівною 100.

Змінити цей порядок можна за допомогою блоку:

```
if (ind > 10) {if (ind < 20) x = 0; else x = 1;}
```

Взагалі не варто використовувати складні вкладені умовні оператори. Перевірка умов займає багато часу. По можливості краще скористатися логічними операціями, наприклад, в нашому прикладі можна написати

```
if (ind >= 10 && ind <= 20) x = 0; else x = 1;
```

Користуйтесь тим, що ви узнали про пропозиції в Дискретній Математиці. Одну з них можна замінити на еквівалентну, більш простішу для розуміння.

В лістингі 1.4 обчислюються корені квадратного рівняння $ax^2 + bx + c = 0$ для любих коефіцієнтів, в тому числі і нульових.

Лістинг 2.4. Обчислення коренів квадратного рівняння

```
class QuadraticEquation{
public static void main(String[] args){
double a = 0.5, b = -2.7, c = 3.5, d, eps=1e-8;
if (Math.abs(a) < eps)
if (Math.abs(b) < eps)
if (Math.abs(c) < eps) // Всі коефіцієнти рівні нулю
System.out.println("Розвязок -будь-яке число");
else
System.out.println("Розвязків немає");
else
System.out.println("x1 = x2 = " +(-c / b) );
else { // Коефіцієнти не рівні нулю
if((d = b**b - 4*a*c)< 0.0){ // Комплексні корені
d = 0.5 * Math.sqrt(-d) / a;
a = -0.5 * b / a;
System.out.println("x1 = " +a+ " +i " +d+
",x2 = " +a+ " -i " +d);
} else {
// Дійсні корені
d =0.5 * Math.sqrt(d) / a;
a = -0.5 * b / a;
System.out.println("x1 = " + (a + d) + ", x2 = " +(a - d));
}
}
}
}
```

В цій програмі використані методи обчислення модуля **abs()** і квадратного кореня **sqrt()** з дійсного числа із вбудованого в **Java API** класу **Math**. Оскільки всі обчислення з дійсними числами виконуються наближено, ми вважаємо, що коефіцієнт рівняння дорівнює нулю, якщо його модуль менше 0,00000001. Зверніть увагу на те, як в методе **println()** використовується зєднання рядків, і на те, як операція присвоювання при обчисленні дискримінанта вкладена в логічний вираз. Прослідкуйте складну ієархію операторів **if ... else**. Якщо це вам здається дуже заплутаним, переробіть програму використовуючи логічні функції.

["Продвинутим" користувачам](#)

Вам уже хочеться вводити коєфіцієнт **a**, **b** і **c** прямо з клавіатури? Будь ласка, користуйтесь методом **System. in. read (byte [] bt)**, але майте на увазі, що цей метод записує введені цифри в масив байтів **bt** в кодуванні **ASCII**, в кожний байт по одній цифрі. Масив байтів потім треба перетворити в дійсне число, наприклад, методом **Double(new String(bt)).doubleValue()**. Незрозуміло? Але це ще не все, треба опрацювати виключні ситуації, які можуть виникнути при введенні. Пізніше, можливо, ми розглянемо це питання більш детально.

2.10.4. Оператори циклу

2.10.4.1. Оператор while

Основний оператор циклу — оператор **while** — виглядає так:

while (логвир) оператор

Спочатку обчислюється логічний вираз **логвир**; якщо його значення **true**, то виконується оператор, що утворює цикл. Потім знову обчислюється **логвир** і діє оператор, і так до тих пір, поки не отримаємо значення **false**. Якщо **логвир** з самого початку рівний **false**, то **оператор** не буде виконуватися ні разу. Попередня перевірка забезпечує безпеку виконання циклу, дозволяє уникнути переповнення, ділення на нуль і інші неприємності. Тому оператор **while** являється основним, а в деяких мовах і єдиним оператором циклу.

Оператор в циклі може бути і пустим, наприклад, наступний фрагмент коду:

```
int i = 0;
double s = 0.0;
while ((s += 1.0 / ++i) < 10);
```

обчислює суму членів гармонічного ряду до тих пір, поки вона досягне значення 10. Такий стиль характерний для мови **C**. Не варто ним захоплюватися, щоб не перетворити текст програми в шифровку, на яку ви самі через пару тижнів будете дивитися зі здивуванням.

Можна організувати і нескінчений цикл:

while (true) оператор

Звичайно, із такого циклу треба передбачити якийсь вихід, наприклад, оператором **break**, як в лістинзі 1.5. В протилежному випадку програма зациклиться, і вам прийдеться закінчувати її виконання "комбінацією з трьох пальців" **<Ctrl>+<Alt>+** у **MS Windows 95/98/ME**, або через **Task Manager** у **Windows NT/2000**.

Якщо в цикл треба включити декілька операторів, то слід створити блок операторів **{}**.

2.10.4.2. Оператор do-while

Другий оператор цикла — оператор **do-while** — має вигляд **do** оператор **while (логвир)**

Тут спочатку виконується оператор, а потім відбувається обчислення логічного виразу **логвир**. Цикл виконується, поки **логвир** залишається рівним **true**.

Знавцям Pascal

В циклі **do-while** перевіряються умови продовження, а не закінчення циклу.

Суттєва різниця між цими двома операторами циклу тільки в тому, що в циклі **do-while** оператор обовязково виконується хоча б один раз.

Наприклад, нехай задана якась функція `f(x)`, що має на відрізку $[0; b]$ рівно один корінь. В лістингі 1.5 приведена програма, що обчислює цей корінь наближено методом ділення пополам (бісекції, дихотомії).

Лістинг 2.5. Знаходження кореня нелінійного рівняння методом бісекції

```
class Bisection {
    static double f(double x){
        return x*x*x - 3*x*x +3; // Або щось інше
    }
    public static void main(String[] args){
        double a = 0.0, b = 1.5, c, y, eps = 1e-8;
        do{
            c = 0.5 * (a + b); y = f(c);
            if (Math.abs(y) < eps) break;
            // Корінь знайдено. Виходимо із циклу
            // Якщо на кінцях відрізка [a; c] функція має різні знаки:
            if (f (a) * y < 0.0) b = c;
            // Значить, корінь тут. Переносимо точку b в точку c
            // В протилежному випадку:
            else a = c;
            // Переносимо точку a в точку c
            // Продовжуємо, доки відрізок [a; b] не стане малим
        } while (Math. abs (b -a) >= eps);
        System.out.println("x = " +c+ ", f(" +c+ ") = " +y) ;
    }
}
```

Клас `Bisection` складніший за попередні приклади: в ньому крім метода `main()` єсть іще метод обчислення функції `f(x)`. Тут метод `f()` дуже простий: він обчислює значення многочлена і повертає його в якості значення функції, причому все це виконується одним оператором:

`return` вираз

В методі `main()` появився ще один новий оператор `break`, який просто закінчує виконання циклу, якщо ми по щастівій випадковості наткнулися на наближене значення кореня. Уважний читач помітив і появу модифікатора `static` в оголошенні метода `f()`. Він необхідний тому, що метод `f()` викликається із статичного метода `main()`.

2.10.4.3. Оператор for

Третій оператор цикле — оператор `for` — виглядає так:

`for (спісокВир1; логВир; спісокВир2) оператор`

Перед виконанням циклу обчислюється список виразів `спісокВир1`. Це нуль або декілька виразів, перечислених через кому. Вони обчислюються зліва направо, і в наступному виразі уже можна використовувати результат попереднього виразу. Як правило, тут задаються початкові значення змінних цикле.

Потім обчислюється логічний вираз `логВир`. Якщо він істинний, `true`, то діє оператор, потім обчислюється зліва направо вирази із списку виразів `спісокВир2`. Далі знову перевіряється `логВир`. Якщо він істинний, то виконується оператор і `спісокВир2` і т. д. Як тільки `логВир` стане рівним `false`, виконання циклу закінчується.

Коротше кажучи, виконується послідовність операторів

```
списокВир1; while (логВир) {
    оператор
    списокВир2; }
```

з тим виключенням, що, коли оператором в циклі являється оператор `continue`, то `списокВир2` все-таки виконується.

Замість `списокВир1` може стояти одне визначення змінних обовязково з початковим значенням. Такі змінні діють лише в межах цього цикла.

Будь-яка частина оператора `for` може бути відсутня: цикл може бути пустим, вираз в заголовку теж, при цьому крапки з комою зберігаються. Можна задати нескінчений цикл:

```
for (;;) оператор
```

В цьому випадку в тілі циклу слід передбачити який-небудь вихід.

Хоча в операторі `for` закладені великі можливості, використовується він, головним чином, для перечислень, коли їх число відомо, наприклад, фрагмент коду ,

```
int s=0;
for (int k = 1; k <= N; k++) s += k * k;
```

// Тут змінна k вже невідома

обчислює суму квадратів перших N натуральних чисел.

2.10.5. Оператор `continue` і мітки

Оператор `continue` використовується тільки в операторах циклу. Він має дві форми. Перша форма складається тільки із слова `continue` і здійснює негайний переход до наступної ітерації циклу. В черговому фрагменті коду оператор `continue` дозволяє обійти ділення на нуль:

```
for (int i = 0; i < N; i++){
if (i == j) continue;
s += 1.0 / (i - j);
}
```

Друга форма містить мітку:

`continue` мітка

мітка записується, як і всі ідентифікатори, із літер `Java`, цифр і знака підкреслювання, але не вимагає ніякого описання. Мітка ставиться перед оператором або відкриваючій фігурній дужці і відокремлюється від них двокрапкою. Так виходить помічений оператор або помічений блок.

Знавцям Pascal

Мітка не вимагає описання і не може починатися з цифри.

Друга форма використовується тільки у випадку декількох вкладених циклів для негайного переходу до чергової ітерації одного із зовнішніх циклів, а саме, поміченого цикла.

2.10.6. Оператор `break`

Оператор `break` використовується в операторах циклу і операторі варіанта для негайногого виходу із цих конструкцій.

Оператор `break` мітка

Наступна схема пояснює цю конструкцію.

```
М1: { // Зовнішній блок
М2: { // Вкладений блок — другий рівень
М3: { // Третій рівень вкладеності...
if (щось трапилося) break M2;
// Якщо true, то тут нічого не виконується
}
// Тут теж нічого не виконується
}
// Сюди передається управління
}
```

Спочатку збиває в пантелику та обставина, що мітка ставиться перед блоком або оператором, а управління передається за цей блок або оператор. Тому не варто захоплюватися оператором `break` з міткою.

2.10.7. Оператор варіанта

Оператор варіанта `switch` організує розгалуження по декількох напрямках. Кожна гілка відмічається константою або константним виразом якого-небудь цілого типу (крім `long`) і вибирається, якщо значення певного виразу співпадає з цією константою. Вся конструкція виглядає так.

```
switch (цілVir){
case констVir1: оператор1
case констVir2: оператор2
.....
case констVirN: операторN
default: операторDef
}
```

Вираз в дужках `цілVir` може бути типу `byte`, `short`, `int`, `char`, але не `long`. Цілі числа або цілочисельні вирази, складені із констант, `констVir` теж не повинні мати тип `long`.

Оператор варіанта виконується так. Всі константні вирази обчислюються заздалегідь, на етапі компіляції, і повинні мати відмінні один від одного значення. Спочатку обчислюються цілочисельний вираз `цілVir`. Якщо він співпадає з однією із констант, то виконується оператор, помічений цією константою. Потім виконуються ("fall through labels") всі наступні оператори, включаючи і `операторDef`, і робота оператора варіант закінчується.

Якщо ж жодна константа не рівна значенню виразу, то виконується `операторDef` і всі наступні оператори. Тому гілка `default` повинна записуватися останньою. Гілка `default` може бути відсутня, тоді в цій ситуації оператор варіанта взагалі нічого не робить.

Таким чином, константи у варіантах `case` грають роль тільки міток, точок входу в оператор варіанта, а далі виконуються всі інші оператори в порядку їх запису.

Знавцям Pascal

Після виконання одного варіанта оператор `switch` продовжує виконувати всі інші варіанти.

Частіше всього необхідно "пройти" тільки одну гілку операторів. В такому випадку використовується оператор `break`, який зразу ж зупиняє виконання оператора `switch`. Може знадобитися виконати один і той же оператор в різних гілках `case`. В цьому випадку ставимо декілька міток `case` підряд. Ось простий приклад.

```
switch(dayOfWeek) {
    case 1: case 2: case 3: case 4:case 5:
        System.out.println("Week-day");, break;
    case 6: case 7:
        System.out.println("Week-end"); break;
    default:
        System.out.println("Unknown day");
}
```

Зауваження

Не забувайте завершати варіанти оператором `break`.

2.11. Масиви

Як завжди в програмуванні **массив** — це сукупність змінних одного типу, що зберігаються в суміжних комірках оперативної пам'яті.

Масиви в мові Java відносяться до посилкових типів і описуються своєрідно, але характерно для посилкових типів. Описання проходить в три етапи.

Перший етап — **оголошення (declaration)**. На цьому етапі визначається тільки змінна типу **посилання (reference)** на **массив**, і містить тип масиву. Для цього записується ім'я типу елементів масиву, квадратними дужками указується, що оголошується посилання на масив, а не проста змінна, і перечислюються імена змінних типу посилання, наприклад,

```
double[] a, b;
```

Тут визначені дві змінні — посилання **a** і **b** на масиви типа **double**. Можна поставити квадратні дужки і безпосередньо після імені. Це зручно робити серед визначень звичайних змінних:

```
int l = 0, ar[], k = -1;
```

Тут визначені дві змінні цілого типа **i** і **k**, і оголошено посилання на цілочисельний масив **ar**.

Другий етап — **визначення (installation)**. На цьому етапі указується кількість елементів масиву, це називається його **довжиною**, виділяється місце для масиву в оперативній пам'яті, змінна-посилання одержує адресу масиву. Всі ці дії виконуються ще однією операцією мови Java — операцією **new тип**, що виділяє місце в оперативній пам'яті для об'єкта указаного в операції типу і повертає в якості результату адресу цього місця. Наприклад,

```
a = new double[5];
b = new double[100];
ar = new int[50];
```

Індекси масивів завжди починаються з 0. Масив **a** складається із пяти змінних **a[0]**, **a[1]**, , **a[4]**. Елемента

`a[5]` в масиві немає. Індекси можна задавати любими цілочисельними виразами, крім типа `long`, наприклад, `a[i+j]`, `a[i%5]`, `a[++i]`. Виконуюча система `Java` слідкує за тим, щоб значення цих виразів не виходили за межі довжини масива.

Третій етап — *ініціалізація* (`initialization`). На цьому етапі елементи масиву отримують початкові значення. Наприклад,

```
a[0] = 0.01; a[1] = -3.4; a[2] = 2.89; a[3] = 4.5; a[4] = -6.7;
```

```
for (int i = 0; i < 100; i++) b[i] = 1.0 /i;
```

```
for (int i = 0; i < 50; i++) ar[i] = 2 * i + 1;
```

Перші два етапи можна сумістити:

```
a = new double[5], b = new double[100];
```

```
int i = 0, ar[] = new int[50], k = -1;
```

Можна зразу задати і початкові значення, записавши їх в фігурних дужках через кому у вигляді констант або константних виразів. При цьому навіть необов'язково указувати кількість елементів масиву, вона буде рівною кількості початкових значень;

```
double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};
```

Можна сумістити другий і третій етап:

```
a = new double[] {0.1, 0.2, -0.3, 0.45, -0.02};
```

Можна навіть створити безіменний масив, зразу ж використовуючи результат операції `new`, наприклад, так:

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

Посилання на масив не являється частиною описаного масива, ії можна перекинуть на другий масив того ж типу операцією присвоювання. Наприклад, після присвоювання `a = b` обидві посилки `a` і `b` указують на один і той же масив із 100 дійсних змінних типу `double` і містять одну і ту ж адресу.

Посилання може присвоїти "пусте" значення `null`, що не указує на жодну адресу оперативної пам'яті:

```
ar = null;
```

Після цього масив, на який указувало дане посилання, втрачається, якщо на нього не було других посилань.

Крім простої операції присвоювання, з посиланнями можна виконувати ще тільки порівняння на рівність, наприклад, `a = b`, і нерівність, `a != b`. При цьому співставляються адреси, що містяться в посиланнях, ми можемо віднати, чи не посилаються вони на один і той же масив.

Зауваження для спеціалістів

Масиви в `Java` завжди визначаються динамічно, хоча посилання на них задаються статично.

Крім посилання на масив, для кожного масиву автоматично визначається ціла константа з одним і тим же

іменем `length`. Вона рівна довжині масива. Для кожного масиву ім'я цієї константи уточнюється іменем масиву через точку. Так, після наших визначень, константа `a.length` рівна 5, константа `b.length` рівна 100, а `ar.length` рівна 50.

Останній елемент масиву `a` можна записати так: `a[a.length - 1]`, передостанній — `a[a.length - 2]` і т. д. Елементи масиву звичайно перебираються в циклі виду:

```
double aMin = a[0], aMax = aMin;
for (int i = 1; i < a.length; i++) {
    if (a[i] < aMin) aMin = a[i];
    if (a[i] > aMax) aMax = a[i];
}
double range = aMax - aMin;
```

Тут обчислюється діапазон значень масиву.

Елементи масиву — це звичайні змінні свого типу, з ними можна виконувати всі операції, допустимі для цього типу:

`(a[2] + a[4]) / a[0]` і т. д.

Знавцям C/C++

Масив символів в `Java` не являється рядком, навіть якщо він закінчується нуль-символом '`\u0000`'.

2.12. Багатовимірні масиви

Елементами масиву у `Java` можуть бути знову масиви. Можна оголосити:

`char[][] c;`

що еквівалентно

`char[] c[];`

або

`char c[][];`

Потім визначаємо зовнішній масив:

`c = new char[3][];`

Становиться ясно, що `c` — масив, який складається з трьох елементів-масивів. Тепер визначимо його елементи-масиви:

`c[0] = new char[2];`

`c[1] = new char[4];`

`c[2] = new char[3];`

Після цих визначень змінна `c.length` рівна 3, `c[0].length` рівна 2,

`c[1].length` рівна 4 і `c[2].length` рівна 3.

Нарешті, задаємо початкові значення `c[0][0] = 'a', c[0][1] = 'r', c[1][0] = 'r', c[1][1] = 'a', c[1][2] = 'y'` і т.д.

Зауваження

Двомірний масив у [Java](#) не зобов'язаний бути прямокутним.

Описування можна скоротити:

```
int[][] d = new int[3][4];
```

А початкові значення задати так:

```
int[][] d = {{1, 2, 3}, {4, 5, 6}};
```

В лістингі 1.6 приведено приклад програми, яка обчислює перші 10 рядків трикутника Паскаля, заносить їх в трикутний масив і виводить його елементи на екран.

Лістинг 2.6. Трикутник Паскаля

```
class PascalTriangle{
    public static final int LINES = 10; // Так визначаються константи
    public static void main(String[] args) {
        int[][] p = new int[LINES][];
        p[0] = new int[1];
        System.out.println(p[0][0] = 1);
        p[1] = new int[2];
        p[1][0] = p[1][1] = 1;
        System.out.println(p[1][0] + " " + p[1][1]);
        for (int i = 2; i < LINES; i++) {
            p[i] = new int[i+1];
            System.out.print((p[i][0] = 1) + " ");
            for (int j = 1; j < i; j++)
                System.out.print(" " + (p[i][j] = p[i-1][j-1] + p[i-1][j]));
            System.out.println(p[i][i] = 1);
        }
    }
}
```

Заключення

Уф-ф-ф!! Ось ви і подолали базові конструкції мови. Раз ви добралися до цього місця, значить, умієте уже дуже багато. Ви можете написати програму на [Java](#), налаштувати її, усунувши помилки, і виконати. Ви здатні запрограмувати будь-який не занадто складний обчислювальний алгоритм, що опрацьовує числові дані. Тепер можна перейти до питань створення складних виробничих програм. Такі програми вимагають ретельного планування. Зробити це допомагає об'єктно-орієнтоване програмування, до якого ми зараз і переходимо.

Лабораторна робота 1. Базові поняття Java

1. Підготуйте простенькі програми для ілюстрації елементів мови [Java](#) розглянутих в пунктах

2.5.1
2.5.3
2.5.4
2.6
2.7.2
2.7.3
2.7.4
2.7.5.1
2.7.5.2
2.7.5.3
2.7.5.4
2.7.5.5
2.7.6
2.7.7
2.7.8
2.8
2.9
2.10.1
2.10.2
2.10.3
2.10.4.1
2.10.4.2
2.10.4.3
2.10.5
2.10.6
2.10.7
2.11
2.12

Пам'ятайте, завдання вважається виконаним, якщо програма спрацювала так, як ви того і хотіли. А те що ви хотіли від програми, повинно бути записане в коментарі на початку програми. Тут же вкажете і своє прізвище. Скопіюйте текст програми у [doc](#) файл і туди ж скопіюйте вікно [Командна строка](#) з результатом виконання програми. Розподіліть самостійно програми між собою. Назвіть файл відповідним пунктом, за яким повинно слідувати ваше прізвище, наприклад 21041_Porov. Ці файли здати старостам, які обєднають їх у папку [Java2Lab](#) і передадуть викладачу.

Деякі з цих програм ви поясніте всім на лекціях, деякі продемонструєте в лабораторії.

2. Ознайомтесь з методами класів [System](#) і [Character](#) по відповідним файлам. Звертайтесь до цих файлів кожен раз, коли треба зрозуміти, як працює той чи інший метод .

Програмування в Java

Урок 3. Об'єктно-орієнтоване програмування в Java

Реквізити:

- Текст лекції
- Файл `Object.html` з описом відповідного класу
- Файл `System.txt` з реалізацією відповідного класу

Зміст

- Парадигми програмування
- Принципи об'єктно-орієнтованого програмування
- Абстракція
- Ієрархія
- Відповіальність
- Модульність
- Принцип KISS
- Як описати клас і підклас
- Абстрактні методи і класи
- Остаточні члени і класи
- Клас `Object`
- Конструктори і класи
- Операція `new`
- Статичні члени класу
- Клас `Complex`
- Метод `main()`
- Де видимі змінні
- Заключення

Дещо, з написаного далі, ви вже знаєте або принаймні чули. Але не спішіть пропускати текст у пошуках чогось нового. Читайте все підряд і розмірковуйте. Пройде ще дуже багато часу доки ви все будете розуміти.

3.1. Парадигми програмування

Вся піввікова історія програмування комп'ютерів, а може бути, і історія всієї науки — це намагання зівладати зі складністю навколишнього світу. Завдання, які постають перед програмістами, становляться все більш громіздкими, інформація, яку треба опрацювати, росте як снігова куля. Ще недавно звичайними одиницями виміру інформації були кілобайти і мегабайти, а зараз уже говорять тільки про гігабайти і терабайти. Як тільки програмісти пропонують більш-менш задовільне розвязання поставлених задач, тут же виникають нові, ще більш складні задачі. Програмісти придумують нові методи, створюють нові мови. За піввіку з'явилося декілька сот мов, запропоновано багато методів і стилів. Деякі методи і стилі становляться загальноприйнятими і утворюють на деякий час так звану *парадигму програмування*.

Перші, навіть самі простіші програми, написані в машинних кодах, складали сотні рядків зовсім незрозумілого тексту. Для спрощення і прискорення програмування придумали мови високого рівня: FORTRAN, Algol і сотні інших, поклавши рутинні операції по створенню машинного коду на компілятор. Ті ж програми, переписані на мовах високого рівня, стали набагато зрозумілішими і коротшими. Але життя вимагало розвязання більш складних задач, і програми знову збільшилися у розмірах, стали неоглядними.

Виникла ідея: оформити програму у вигляді декількох, по можливості простих, процедур або функцій, кожна з яких вирішує свою конкретну задачу. Написати, скомпілювати і налаштувати невелику процедуру можна легко і швидко. Потім залишається лише зібрати всі процедури в потрібному порядку в одну програму. Крім того, один раз написані процедури можна потім використовувати в інших програмах як

будівельні цеглинки. [Процедурне програмування](#) швидко стало парадигмою. У всі мови всокого рівня включили засоби написання процедур і функцій. Появилось багато бібліотек процедур і функцій на всі випадки життя. Першою такою мовою програмування для вас була [QBasic](#).

Постало питання про те, як виявити структуру програми, розбити програму на процедури, яку частину коду виділити в окрему процедуру, як зробити алгоритм розвязку задачі простим і наочним, як зручніше звязати процедури між собою. Досвідчені програмісти запропонували свої рекомендації, названі [структурним програмуванням](#). Структурне програмування виявилось зручним і стало парадигмою. Появились мови програмування, наприклад [Pascal](#), на яких зручно писати структурні програми. Більше того, на них дуже важко написати неструктурні програми.

Складність поставших перед програмістами задач проявилась і тут: програми стали містити сотні процедур, і знову виявились неосяжними. "Цеглинки" стали занадто маленькими. Потребувався новий стиль програмування,

В той же час виявилося, що успішна або неуспішна структура вихідних даних може значно полегшити або ускладнити їх опрацювання. Одні вихідні дані зручніше обєднати в масив, для інших більше підходить структура дерева або стека. [Ніклаус Вірт](#) навіть назвав свою книгу ["Алгоритми + структури даних = програми"](#).

Виникла ідея обєднати вихідні дані і всі процедури їх опрацювання в один модуль. Ця ідея [модульного програмування](#) швидко завоювала умі і на деякий час стала парадигмою. Програми складалися із окремих модулів, що містили десяток-другий процедур і функцій. Ефективність таких програм тим вище, чим менше модулі залежать один від одного. Автономність модулів дозволяє створювати і бібліотеки модулів, щоб потім використовувати їх в якості будівельних блоків для програми. Модульна парадигма програмування дуже добре була видима в [Delphi](#), коли до вашої програми автоматично приєднувався десяток другий модулів.

Для того щоб забезпечити максимальну незалежність модулів один від одного, треба чітко відділити процедури, котрі будуть викликатися іншими модулями, — відкриті ([public](#)) процедури, від допоміжних, котрі опрацьовують дані, включені в цей модуль, — закритих ([private](#)) процедур — щось на зразок глобальних і локальних змінних. Перші перечисляються в окремій частині модуля — *інтерфейс* ([interface](#)), другі приймають участь тільки в *реалізації* ([implementation](#)) модуля. Дані, занесені в модуль, також діляться на відкриті, вказані в інтерфейсі і доступні для інших модулів, і закриті, доступні тільки для процедур того ж модуля. В різних мовах програмування цей розподіл здійснюється по-різному. В мові [Turbo Pascal](#) модуль спеціально ділиться на інтерфейс і реалізацію. В мові С інтерфейс виноситься в окремі "заголовкові" ([header](#)) файли. В мові C++, крім того, для описання інтерфейса можна скористатися абстрактними класами. В мові Java є спеціальна конструкція для описання інтерфейсів, яка так і називається — [interface](#), але можна написати і абстрактні класи.

Так виникла ідея укриття, *інкапсуляції* ([incapsulation](#)) даних і методів їх обробки. Подібні ідеї періодично виникають в дизайні побутової техніки. То телевізори щетиняться кнопками, ручками і движками на радість допитливому телеглядачу, панує "приладний" стиль, то все кудись пропадає, а на панелі залишаються тільки кнопка включення і ручка регулювання звуку. Допитливий телеглядач береться за відкрутку.

Інкапсуляція, звичайно, робиться не для того, щоб сховати від іншого модуля щось цікаве. Тут преслідуються дві основні цілі. Перша — забезпечити безпеку використання модуля, винести в інтерфейс, зробити загальнодоступними тільки ті методи обробки інформації, котрі не можуть зіпсувати або видалити вихідні дані. Друга мета — зменшити складність, сховавши від зовнішнього світу непотрібні деталі реалізації.

Знову виникає питання, яким чином розбити програму на модулі? Тут виявились корисними методи розвязання старої задачі програмування — моделювання дій штучних і природних об'єктів: роботів, станків з програмним забезпеченням, безпілотних літаків, людей, тварин, рослин, систем забезпечення життєдіяльності, систем управління технологічними процесами.

В самім ділі, кожний об'єкт — робот, автомобіль, людина — має певні характеристики. Ними можуть служити: вага, зріст, прізвище, максимальна швидкість, кут повороту, вантажопідемність. Об'єкт може виконувати якісі дії: зміщуватися в просторі, повертатися, підіймати, копати, рости або зменшуватися, їсти, пити, народжуватися і помирати, змінюючи свої початкові характеристики. Зручно змоделювати об'єкт у вигляді модуля. Його характеристики будуть даними, постійними або змінними, а дії — процедурами.

Те ж саме з програмою, її розбивають на модулі так, щоб вона претворилася у сукупність взаємодіючих об'єктів. Так виникло *об'єктно-орієнтоване програмування* (*object-oriented programming*), скорочено ООП (*OOP*) — сучасна парадигма програмування.

У вигляді об'єктів можна представити зовсім неочікувані поняття. Наприклад, вікно на екрані дисплея — це об'єкт, що має ширину *width* і висоту *height*, розташоване на екрані, описується координатами (*x*, *y*) лівого верхнього кута вікна, а також шрифт, яким у вікно виводиться текст, скажімо, *Times New Roman*, колір фону *color*, декілька кнопок, смуги прокрутки і інші характеристики. Вікно може зміщуватися по екрану методом *move()*, збільшуватися або зменшуватися в размірах методом *size()*, звертатися в ярлик методом *iconify()*, реагувати на дії миші і натискання клавіш. Це повноцінний об'єкт! Кнопки, смуги прокрутки і інші елементи вікна — це теж об'єкти зі своїми розмірами, шрифтами, зміщеннями.

Здається, вважати, що вікно само "уміє" виконувати дії, а ми тільки даємо йому доручення: "Звернись, розвернись, перемістись", — це дещо неочікуваний погляд на речі, але ж тепер можна подавати команди не тільки мишкою і клавішами, але й голосом!

Ідея об'єктно-орієнтованого програмування виявилась дуже плодотворною і стала активно розвиватися. Виявилось, що зручно ставити задачу зразу у вигляді сукупності діючих об'єктів — виник *об'єктно-орієнтований аналіз*, ООА. Вирішили проектувати складні системи у вигляді об'єктів — зявилось *об'єктно-орієнтоване проектування*, ООП (*OOD, object-oriented design*).

Розглянемо детальніше принципи об'єктно-орієнтованого програмування

3.2. Принципи об'єктно-орієнтованого програмування

Об'єктно-орієнтоване програмування розвивається уже більше двадцяти років. Єсть декілька шкіл, кожна з яких пропонує свій набір принципів роботи з об'єктами і по-своєму викладає ці принципи. Але єсть і декілька загальноприйнятих понять. Перечислимо їх.

3.2.1. Абстракція

Описуючи поведінку якого-небудь об'єкта, наприклад автомобіля, ми будуємо його модель. Модель, як правило, не може описати об'єкт повністю, реальні об'єкти досить складні. Приходиться відбирати тільки ті характеристики об'єкта, котрі важливі для розв'язання поставленої перед нами задачі. Для описання вантажоперевезень важливою характеристикою буде вантажопідемність автомобіля, а для описання автомобільних гонок вона не суттєва. Але для моделювання гонок обов'язково треба описати метод набору швидкості даним автомобілем, а для вантажоперевезення це не суть важливо.

Ми повинні *абстрагуватися* від деяких конкретних деталей об'єкта. Дуже важливо вибрати правильну ступінь абстракції. Занадто висока ступінь дасть тільки приблизне описание об'єкта, не дозволить правильно моделювати його поведінку. Занадто низька ступінь абстракції зробить модель дуже складною, перевантажену деталями, і тому непридатною.

Наприклад, можна цілком точно предбачити погоду на завтра в певному місці, але розрахунки по такій моделі триватимуть три доби навіть на самому потужному комп'ютері. Для чого потрібна модель, що запізнюються на два дні? Ну а точність моделі, якою користуються синоптики, ми всі знаємо самі. Зате розрахунки по цій моделі займають всього декілька годин.

Описання кожної моделі робиться у вигляді одного або декількох *класів* (*classes*). Клас можна вважати

проектом, зліпком, кресленням, по якому потім будуть створюватися конкретні обєкти. Клас містить описання змінних і констант, що характеризують об'єкт. Вони називаються *полями класу* (*class fields*). Процедури, які описують поведінку об'єкта, називаються *методами класу* (*class methods*). Всередині класу можна описати і *вкладені класи* (*nested classes*) і *вкладені інтерфейси*. Поля, методи і вкладені класи першого рівня являються *членами класу* (*class members*). Різні школи об'єктно-орієнтованого програмування пропонують різні терміни, ми використовуємо термінологію, прийняту в технології *Java*.

Ось зразок описання автомобіля:

```
class Automobile{
    int maxVelocity; // Поле, що містить найбільшу швидкість автомобіля
    int speed; // Поле, що містить поточну швидкість автомобіля
    int weight; // Поле, що містить вагу автомобіля
    // Інші поля...
    void moveTo(int x, int y){ // Метод, що моделює переміщення автомобіля. Параметри x
        i y - не поля
        int a = 1; // Локальна змінна - не поле
        // Тіло метода. Тут описується закон переміщення автомобіля в точку (x, y)
    }
    // Інші методи. . .
}
```

Знавцям Pascal

В Java немає вкладених процедур і функцій, в тілі метода не можна описати інший метод.

Після того як описання класу закінчено, можна створювати конкретні об'єкти, екземпляри (*instances*) описаного класу. Створення екземплярів відбувається в три етапи, подібно описанню масивів. Спочатку оголошуються посилання на об'єкти: записується ім'я класу, і через пробіл перечисляються екземпляри класу, точніше, посилання на них.

```
Automobile lada2110, fordScorpio, oka;
```

Потім операцією *new* визначаються самі об'єкти, під них виділяється оперативна пам'ять, посилання отримує адресу цієї частини пам'яті в якості свого значення.

```
lada2110 = new Automobile();
fordScorpio = new Automobile();
oka = new Automobile();
```

На третьому етапі відбувається ініціалізація об'єктів, задаються початкові значення. Цей етап, як правило, суміщається з другим, якраз для цього в операції *new* повторяється ім'я класа з дужками *Automobile ()*. Це так званий *конструктор* (*constructor*) класу, але про нього поговоримо пізніше.

Оскільки імена полів, методів і вкладених класів у всіх об'єктах одинакові, вони задані в описанні класу, їх треба уточнити іменем посилання на об'єкт:

```
lada2110.maxVelocity = 150;
fordScorpio.maxVelocity = 180;
oka.maxVelocity = 350;// Чому б і ні?
oka.moveTo(35, 120);
```

Нагадаємо, що текстовий рядок в лапках вважається в *Java* як об'єкт класу *String*. Тому можна написати

```
int strlen = "Це об'єкт класу String".length();
```

Об'єкт "рядок" виконує метод `length()`, один із методів свого класу `String`, що підраховує число символів у рядку. В результаті одержимо значення `strlen`, рівне 21. Подібний дивний запис зустрічається в `Java` програмах на кожному кроці.

В багатьох ситуаціях створюють декілька моделів з різним степенем деталізації. Скажімо, для конструктування пальто і шуби потрібна менш точна модель контурів людського тіла і його рухів, а для конструктування фрака або вечірнього плаття — уже набагато точніша. При цьому більш точна модель, з меншим степенем абстракції, буде використовувати уже наявні методи менш точної моделі.

Чи не здається вам, що клас `Automobile` сильно перевантажений? Дійсно, в світі випущені мільйони автомобілів різних марок і видів. Що між ними спільного, крім чотирьох коліс? Та і коліс може бути більше або менше. Чи не краще написати окремі класи для легкових і вантажних автомобілів, для гоночних автомобілів і всюдиходів? Як організувати всю цю множину класів? На це питання об'єктно-орієнтоване програмування відповідає так: треба організувати ієрархію класів.

3.2.2. Ієрархія

Ієрархія об'єктів давно використовується для їх класифікації. Особливо детально вона опрацьована в біології. Всі знайомі з сімействами, родами і видами. Ми можемо зробити описання своїх домашніх тварин (`pets`): кішок (`cats`), собак (`dogs`), корів (`cows`) і інших наступним чином:

```
class Pet{ // Тут описуємо спільні властивості всіх домашніх улюблениців
    Master person; // Хазяїн тварини
    int weight, age, eatTime; // Вага, вік, час годування
    int eat(int food, int drink, int time){ // Процес годування
        // Початкові дії...
        if (time == eatTime) person.getFood(food, drink);
        // Метод приймання їди
    }
    void voice(); // Звуки, що видають тварини
    // Інше...
}
```

Потім створюємо класи, які описують більш конкретні об'єкти, зв'язуючи іх із спільним класом:

```
class Cat extends Pet{ // Описуються властивості, властиві лише кішкам:
    int mouseCatched; // число спійманих мишей
    void toMouse(); // процес ловіння мишей
    // інші властивості
}

class Dog extends Pet{ // Властивості собак:
    void preserve(); // охороняти
}
```

Зверніть увагу, що ми не повторюємо спільні властивості, описані в класі `Pet`. Вони наслідуються автоматично. Ми можемо визначити об'єкт класа `Dog` і використовувати в ньому всі властивості класа `Pet` так, як ніби то вони описані в класі `Dog`:

```
Dog Tuzik = new Dog(), Sharik = new Dog();
```

Після такого визначення можна буде написати

```
Tuzik.age = 3;
int p = Sharik.eat (30, 10, 12);
```

А класифікацію продовжити так:

```
class Pointer extends Dog{ ... } // Властивості породи Пойнтер
class Setter extends Dog{ ... } // Властивості сеттерів
```

Зверніть увагу, що на кожному наступному рівні ієрархії в клас додаються нові властивості, але жодна властивість не пропадє. Тому і використовується слово **extends** — "розширює" і говорять, що клас **Dog** — **роздширення (extension)** класу **Pet**. З іншої сторони, кількість об'єктів при цьому зменшується: собак менше, ніж всіх домашніх тварин. Тому часто говорять, що клас **Dog** — **підклас (subclass)** класу **Pet**, а клас **Pet** — **суперклас (superclass)** або надклас класу **Dog**.

Часто використовують генеалогічну термінологію: батьківський клас, дочірній клас, клас-нащадок, клас-предок, виникають племінники і внуки, вся неспокійна сімейка вступає у відносини, гідні мексиканського серіала.

В цій термінології говорять про **наслідування (inheritance)** класів, в нашому прикладі клас **Dog** наслідує клас **Pet**.

Ми ще не визначили щасливого хазяїна нашого домашнього зоопарка. Опишемо його в класі **Master**. Зробимо прикладку:

```
class Master{ // Хазяїн тварини
String name; // Прізвище, ім'я
// Інші дані
void getFood(int food, int drink); // Годівля
// Інше
}
```

Хазяїн і його домашні тварини постійно контактиують. Їх взаємодія виражається дієсловами "гуляти", "годувати", "охороняти", "чистити", "лашитися", "проситися" і іншими. Для описання взаємодії об'єктів застосовується третій принцип об'єктно-орієнтованого програмування — обовязок або відповідальність.

3.2.3. Відповідальність

В нашему прикладі розглядається тільки взаємодія в процесі годівлі, яка описується методом **eat()**. В цьому методі тварина звертається до хазяїна, умоляючи його застосувати метод **getFood()**.

В англомовній літературі подібне звернення описується словом **message**. Це поняття невдало перекладено на українську мову ні до чого не зобовязуваним словом "повідомлення". Краще було б використати слово "послання", "доручення" або навіть "розпорядження". Але термін "повідомлення" устоявся і нам прийдеться його застосовувати. Чому ж не використовується словосполучення "виклик методу", адже говорять: "Виклик процедури"? Тому що між цими поняттями єсть, по крайній мірі, три відмінності.

- Повідомлення йде до конкретного об'єкта, що знає метод розв'язання задачі, в прикладі цей об'єкт — поточне значення змінної **person**. У кожного об'єкта є свій поточний стан, свої значення полів класу, і це може вплинути на виконання метода.
- Спосіб виконання доручення, що міститься в повідомленні, залежить від об'єкта, якому воно послане. Один хазяїн поставить миску з "**Chappi**", другий кине кістку, третій вижене собаку на вулицю. Цю цікаву властивість називають **поліморфізмом (polymorphism)**, її будемо обговорювати далі.
- Звернення до методу відбудеться лише на етапі виконання програми, компілятор нічого не знає про метод. Це називається "пізнім звязуванням" на противагу "ранньому звязуванню", при якому процедура приєднується до програми на етапі компонування.

Отже, об'єкт **Sharik**, виконуючи свій метод **eat()**, посилає повідомлення об'єкту, посилання на який міститься в змінній **person**, з просьбою видати йому певну кількість їди і питва. Повідомлення записане в рядку **person.getFood(food, drink)**.

Цим повідомленням заключається **контракт** (contract) між об'єктами, суть якого в тому, що об'єкт **Sharik** бере на себе **відповідальність** (responsibility) задати правильні параметри в повідомленні, а об'єкт — поточне значення **person** — бере на себе **відповідальність** застосувати метод годівлі **getFood()** , яким би він не був.

Для того щоб правильно реалізувати принцип відповідальності, застосовується четвертий принцип об'єктно-орієнтованого програмування — **модульність** (modularity).

3.2.4. Модульність

Цей принцип стверджує — кожний клас повинен складати окремий модуль. Члени класу, до яких не планується звертання зовні, повинні бути інкапсульовані. В мові **Java** інкапсуляція досягається додаванням модифікатора **private** до описання члена класу. Наприклад:

```
private int mouseCatched;
private String name;
private void preserve();
```

Ці члени класів становляться **закритими**, ними можуть користуватися тільки екземпляри того ж самого класу, наприклад, **Tuzik** може дати доручення

```
Sharik.preserve();
```

А якщо в класі **Master** ми напишемо

```
private void getFood(int food, int drink);
```

то метод **getFood()** не буде знайдено, і нещасний **sharik** не зможе отримати їжу.

В протилежність закритості ми можемо оголосити деякі члени класа **відкритими**, записавши замість слова **private** модифікатор **public**, наприклад:

```
public void getFood(int food, int drink);
```

До таких членів може звернутися любий об'єкт люального класа.

Знавцям C++

В мові **Java** словами **private**, **public** і **protected** відмічаються окремо кожний член класу.

Принцип модульності предбачає відкривати члени класу тільки у випадку необхідності. Згадайте напис: "Нормальне положення шлагбаума — закрите".

Якщо ж треба звернутися до поля класа, то рекомендується включити в клас спеціальні **методи доступу** (access methods), окремо для читання цього поля (**get method**) і для запису в це поле (**set method**). Імена методів доступу рекомендується починати зі слів **get** і **set**, додаючи до цих слів ім'я поля. Для **JavaBeans** ці рекомендації введені в ранг закону.

В нашому прикладі класу **Master** методи доступу до поля **Name** в самому простому випадку можуть виглядіти так:

```
public String getName() {
    return name;
}
public void setName(String newName)
```

```
{
name = newName;
}
```

В реальних ситуаціях доступ обмежується різними перевірками, особливо в **set**-методах, що змінюють значення полів. Можна перевіряти тип значення, що вводиться, задавати діапазон значень, зрівнювати зі списком допустимих значень.

Крім методів доступу рекомендується створювати провірочні **is**-методи, які повертають логічні значення **true** або **false**. Наприклад, в клас **Master** можна включити метод, перевіряючий, чи задано ім'я хазяїна:

```
public boolean isEmpty(){
return name == null ? true : false;
}
```

і використовувати цей метод для перевірки при доступі до поля **Name**, наприклад:

```
if (masterOl.isEmpty()) masterOl.setName("Іванов");
```

Таким чином, ми залишаємо відкритими лише методи, необхідні для взаємодії обєктів. При цьому зручно спланувати класи так, щоб залежність між ними була найменшою, як прийнято говорити в теорії ООП, було найменше **зачеплення** (**low coupling**) між класами. Тоді структура програми сильно спрощується. Крім того, такі класи зручно використовувати як будівельні блоки для побудови інших програм.

Навпаки, члени класа повинні активно взаємодіяти один з одним, як говорять, мати тісну функціональну **звязність** (**high cohesion**). Для цього в клас належить включати всі методи, що описують поведінку моделюючого обєкта, і тільки такі методи, нічого лишнього. Одно із правил досягнення сильної функціональної звязності, введене Карлом Ліберхером (**Karl J. Lieberherr**), отримало назву **закон Деметра**. Закон гласить: "в методі **m()** класу **A** належить використовувати тільки методи класу **A**, методи класів, до яких належать аргументи метода **m()**, і методи класів, екземпляри яких створюються всередині метода **m()**".

Обєкти, побудовні по цим правилам, схожі на кораблі, споряджені всім необхідним. Вони виходять в автономне плавання, готові виконати будь-яке доручення, на яке розрахована їх конструкція.

Чи будуть закриті члени класа доступні його наслідникам? Якщо в класі **Pet** написано

```
private Master person;
```

то чи можна використовувати **Sharik.person**? Розуміється, ні. Адже в протилежному випадку кожний, хто цікавиться закритими полями класу **A**, може розширити його класом **B**, і продивитися закриті поля класу **A** через екземпляри класу **B**.

Коли треба дозволити доступ наслідникам класу, але небажано вітківати його всьому світу, тоді в **Java** використовується **захищений** (**protected**) доступ, позначений модифікатором **protected**, наприклад, об'єкт **Sharik** може звернутися до поля **person** батьківського класу **Pet**, якщо в класі **Pet** це поле описано так:

```
protected Master person;
```

Треба зразу сказати, що на доступ до члена класу впливає ще і пакет, в якому знаходиться клас, але про це поговоримо в наступній главі.

Із цього загального схематичного описання принципів об'єктно-орієнтованого програмування видно, що мова **Java** дозволяє легко втілювати всі ці принципи. Ви уже зрозуміли, як записати клас, його поля і методи, як інкапсулювати члени класу, як зробити розширення класу і якими принципами належить при

цьому користуватися. Розберемо тепер детальніше правила запису класів і розглянемо додаткові їх можливості.

Але, говорячи про принципи ООП, я не можу утриматися від того, щоб не нагадати основний принцип всякої програмування.

3.2.5. Принцип KISS

Самий основний, базовий і самий великий принцип програмування — принцип **KISS** — не потребує розяснень: "Keep It Simple, Stupid!"

3.3. Як описати клас і підклас

Отже, описання класу починається зі слова **class**, після якого записується ім'я класу. "Code Conventions" рекомендує починати ім'я класу із заглавної літери.

Перед словом **class** можна записати модифікатори класа (**class modifiers**). Це одно із слів **public**, **abstract**, **final**, **strictfp**. Перед іменем вкладеного класу можна поставити, крім того, модифікатори **protected**, **private**, **static**. Модифікатори ми будемо вводити по мірі вивчення мови.

Тіло класу, в якому в будь-якому порядку перечисляються поля, методи, вкладені класи і інтерфейси, заключається в фігурні дужки.

При описанні поля вказується його тип, потім, через пробіл, ім'я і, може бути, початкове значення після знака рівності, яке можна записати константним виразом. Все це уже описано в уроці 2.

Описання поля може починатися з одного або декількох необовязкових модифікаторів **public**, **protected**, **private**, **static**, **final**, **transient**, **volatile**. Якщо треба поставити декілька модифікаторів, то перечисляти їх **JLS** рекомендує в указаному порядку, оскільки деякі компілятори вимагають певного порядку запису модифікаторів.

При описанні метода вказується тип значення, який він повертає або слово **void**, потім, через пробіл, ім'я метода, потім, в дужках, список параметрів. Після цього в фігурних дужках розписується виконуваний метод.

Описання метода може починатися з модифікаторів **public**, **protected**, **private**, **abstract**, **static**, **final**, **synchronized**, **native**, **strictfp**.

В списку параметрів через кому перечисляються тип і ім'я кожного параметра. Перед типом якого-небудь параметра може стояти модифікатор **final**. Такий параметр не можна змінювати всередині метода. Список параметрів може бути відсутнім, але дужки зберігаються.

Перед початком роботи метода для кожного параметра виділяється ячейка оперативної пам'яті, в яку копіюється значення параметра, задане при зверненні до метода. Такий спосіб називається передачею параметрів **по значенню**.

В лістингі 3.1 показано, як можна оформити метод ділення пополам для знаходження кореня нелінійного рівняння із лістинга 1.5.

Лістинг 3.1. Знаходжене кореня нелінійного рівняння методом бісекції

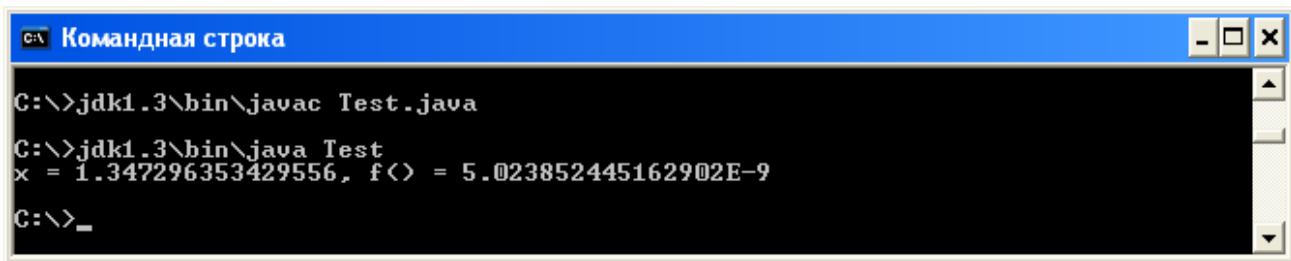
```
class Bisection2{
    private static double EPS = 1e-8; // Константа
    private double a = 0.0, b = 1.5, root; // Закриті поля
    public double getRoot(){return root;}
```

```

// Метод доступа
public double f(double x)
{
    return x*x*x - 3*x*x + 3; // Або щось інше
}
public void bisect() { // Параметрів немає – метод працює з полями екземпляра
    double y = 0.0; // Локальна змінна – не поле
    do{
        root = 0.5 *(a + b); y = f(root);
        if (Math.abs(y) < EPS) break;
        // Корінь знайдено. Виходимо з циклу.
        // Якщо на кінцях відрізка [a; root] функція має різні знаки:
        if (f(a) * y < 0.0) b = root;
        // значить, корінь тут. Переносимо точку b в точку root
        // В протилежному випадку:
    else a = root;
        // переносимо точку a в точку root
        // Продовжуємо, доки [a; b] не стане малим
    } while(Math.abs(b-a) >= EPS);
}
}

class Test{
public static void main(String[] args){
    Bisection2 b2 = new Bisection2();
    b2.bisect();
    System.out.println("x = " +
    b2.getRoot() + // Звертаємося до кореня через метод доступа
    ", f() = " +b2.f(b2.getRoot()));
}
}

```



```

Командная строка

C:\>jdk1.3\bin\javac Test.java
C:\>jdk1.3\bin\java Test
x = 1.347296353429556, f() = 5.023852445162902E-9
C:\>_

```

В описанні метода [f\(\)](#) збережений старий, процедурний стиль: метод одержує аргумент, опрацьовує його і повертає результат. Описання метода [bisect\(\)](#) виконано в дусі ООП: метод активний, він сам звертається до полів екземпляра [b2](#) і сам заносить результат в потрібне поле.

Ім'я метода, число і типи параметрів створюють *сигнатуру (signature)* метода. Компілятор розрізняє методи не по їх іменах, а по сигнатурах. Це дозволяє записувати різні методи з однаковими іменами, що відрізняються числом і/або типами параметрів.

Зауваження

Тип значення, що повертається, не входить в сигнатуру метода, значить, методи не можуть розрізнятися тільки типом результути та їх роботою.

Наприклад, в класі [Automobile](#) ми записали метод [moveTo\(int x, int y\)](#), позначивши пункт призначення його географічними координатами. Можна визначити ще метод [moveTo \(string destination\)](#) для вказівки географічної назви пункту призначення і звертатися до нього так:

```
oka.moveTo("Полтава") ;
```

Таке дублювання методів називається *перевантаженням* (overloading). Перевантаження методів дуже зручне у використанні. Згадайте, в уроці 2 ми виводили дані будь-якого типу на екран методом `println()` не турбуючись про те, дані якого іменно типу ми виводимо. На самім ділі ми використовували різні методи з одним і тим же іменем `println`, навіть не задумуючись про це. Звичайно, всі ці методи треба ретельно спланувати і заздалегідь описати в класі. Це і зроблено в класі `PrintStream`, де представлено близько двадцяти методів `print()` і `println()`. Ось методи `println()`.

void	<code>println()</code>	Terminate the current line by writing the line separator string.
void	<code>println(boolean x)</code>	Print a boolean and then terminate the line.
void	<code>println(char x)</code>	Print a character and then terminate the line.
void	<code>println(char[] x)</code>	Print an array of characters and then terminate the line.
void	<code>println(double x)</code>	Print a double and then terminate the line.
void	<code>println(float x)</code>	Print a float and then terminate the line.
void	<code>println(int x)</code>	Print an integer and then terminate the line.
void	<code>println(long x)</code>	Print a long and then terminate the line.
void	<code>println(Object x)</code>	Print an Object and then terminate the line.
void	<code>println(String x)</code>	Print a String and then terminate the line.

Якщо ж записати метод з тим же іменем в підкласі, наприклад:

```
class Truck extends Automobile{
void moveTo(int x, int y){
    // Якісь дії
}
// Ще щось
}
```

то він перекриє метод суперкласа. Визначивши екземпляр класу `Truck`, наприклад:

```
Truck gazel = new Truck();
```

і записавши `gazel.moveTo(25, 150)`, ми звернемося до метода класу `Truck`. Відбудеться *перевизначення* (overriding) метода.

При перевизначенні права доступу до метода можна тільки розширити. Відкритий метод `public` повинен залишатися відкритим, захищений `protected` може стати відкритим.

Чи можна всередині підкласу звернутися до метода суперкласу? Так, можна, якщо уточнити ім'я метода, словом `super`, наприклад, `super.moveTo(30, 40)`. Можна уточнити і ім'я метода, записаного в цьому ж класі, словом `this`, наприклад, `this.moveTo(50, 70)`, але в даному випадку це вже зайде. Таким же способом можна уточнити і співпадаючі імена полів, а не тільки методів.

Дані уточнення подібні тому, як ми говоримо про себе "я", а не "Іван Петрович", і говоримо "батько", а не "Петро Сидорович".

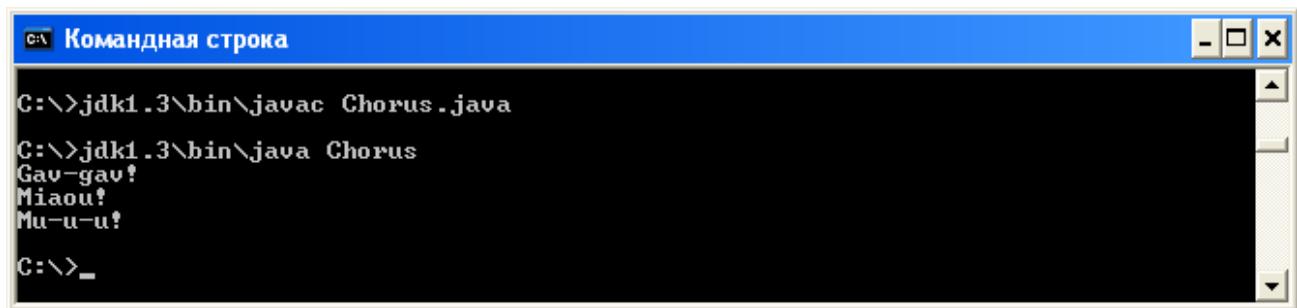
Перевизначення методів приводить до цікавих результатів. В класі `Pet` ми описали метод `voice()`. Перевизначимо його в підкласах і використаємо в класі `chorus`, як показано в лістингі 3.2.

Лістинг 3.2. Приклад поліморфного метода

```

abstract class Pet{
    abstract void voice();
}
class Dog extends Pet{
    int k = 10;
    void voice(){
        System.out.println("Gav-gav!");
    }
}
class Cat extends Pet{
    void voice () {
        System.out.println("Miaou!");
    }
}
class Cow extends Pet{
    void voice(){
        System.out.println("Mu-u-u!");
    }
}
public class Chorus{
    public static void main(String[] args){
        Pet[] singer = new Pet[3];
        singer[0] = new Dog();
        singer[1] = new Cat();
        singer[2] = new Cow();
        for (int i = 0; i < singer.length; i++)
            singer[i].voice();
    }
}

```



Вся справа тут у визначенні поля `singer[]`. Хоча масив `singer []` має тип `Pet`, кожний його елемент посилається на об'єкт свого типу `Dog`, `Cat`, `Cow`. При виконанні програми викликається метод конкретного об'єкта, а не метод класу, яким визначалося ім'я посилання. Так в `Java` реалізується поліморфізм.

Знавцям C++

В мові `Java` всі методи являються віртуальними функціями.

Уважний читач помітив у описанні класу `Pet` нове слово `abstract`. Клас `Pet` і метод `voice()` являються абстрактними.

3.4. Середовище програмування Jbuilder

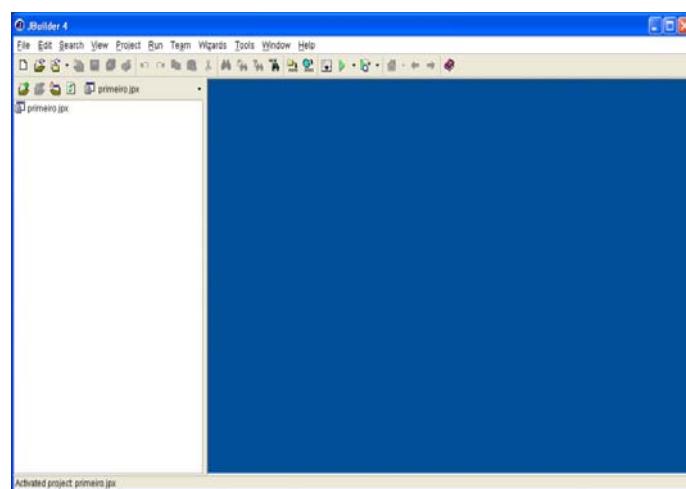
Компіляція і запуск **Java** аплікації з командного рядка не такі вже й складні, якщо в програмі немає помилок. Але таке може приснитися хіба-що уві сні. А на самім ділі, тільки виправиш одну помилку і перекомпілюєш файл, як зявляється повідомлення про іншу помилку. Скоріше напаштування **Java** програми робиться в інтегрованому середовищі програмування **JBuilder**. Правда, тут треба створювати проект з багатьма файлами, але, на щастя, це робиться майже автоматично, вам треба лише заповнити потрібні поля і натискати кнопки.

Отже, відкрийте середовище програмування **JBuilder**. В залежності від того, чи ви це робите перший раз, чи вже працювали раніше, перед вами можуть зявитися різні вікна. Це також може залежати і від версії **JBuilder**, але у вас уже достатньо досвіду, щоб внести в разі необхідності потрібні корективи в те, про що йтиме мова далі. Я користуюся версією **JBuilder4**. Ось яке вікно відкрилося у мене.



Як би ви добре знали англійську мову, то з цього вікна дізналися б багато цікавого, як працювати в середовищі програмування **JBuilder**. Але так як рівень вашої англійської мови не дозволяє це зробити, то вам залишається тільки закрити це вікно і понадіятись на мене. Багато я не встигну пояснити, та для початку і це буде добре. А далі все буде залежати лише від вас.

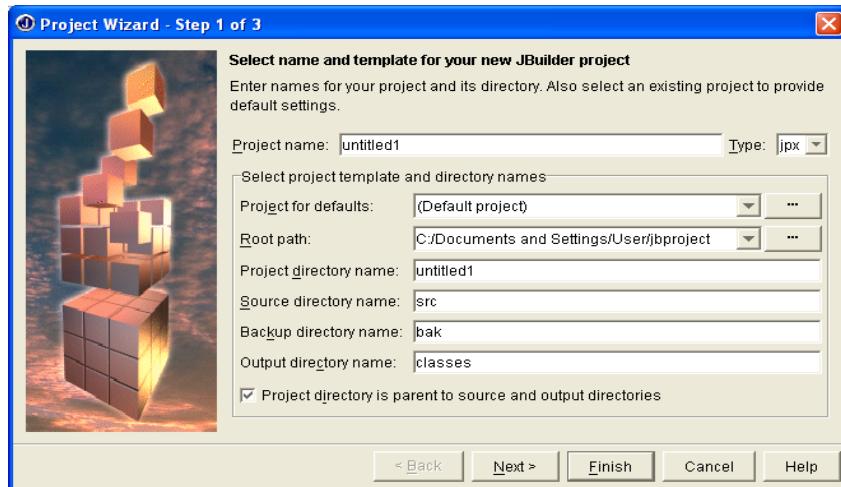
Перед цим я створив новий проект **primeiro.jpx**, але видалив з нього усі файли. Тому у мене зявилося таке вікно. У вас же може зявитися шаблонний проект **Wellcom.jpx** із запрошенням дізнатися як створюються **Java** проекти. Але знову вам на заваді стає англійська мова. Тому знову довіртесь мені.



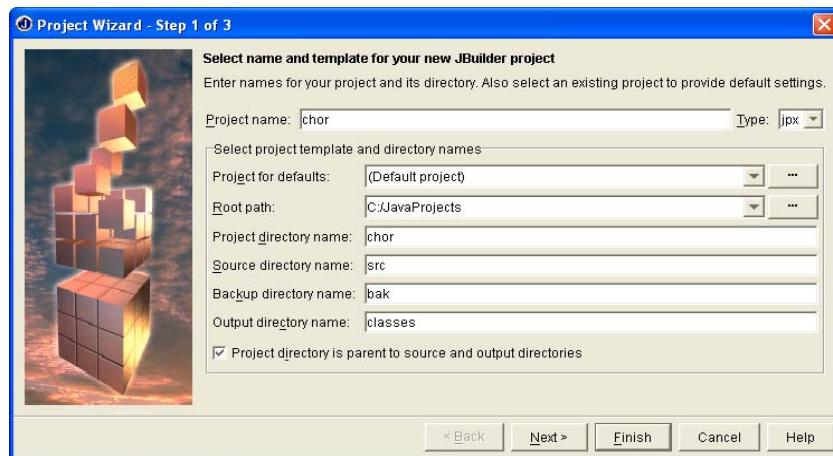
Закрійте всі файли, які ви знайдете у вікні проекту і не звертаючи уваги на те, що залишилось, тисніть в меню **File --> New Project**. Запускається програма **Project Wizard**, яка за три кроки створить нам новий

проект. Назвемо його **chor** - я хочу запустити аплікацію лістингу 3.2 з уроку 3 про котячо-собачий хор, тому їй дав проекту таку назву, а збережемо там де захочемо – я в папці <C:\JavaProjects>.

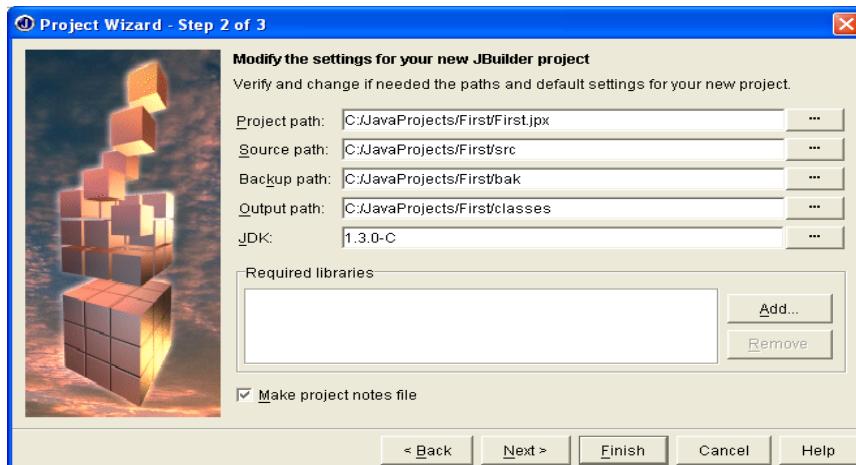
Ось який вигляд мало вікно **Step1** до вибору імені проекту і папки,



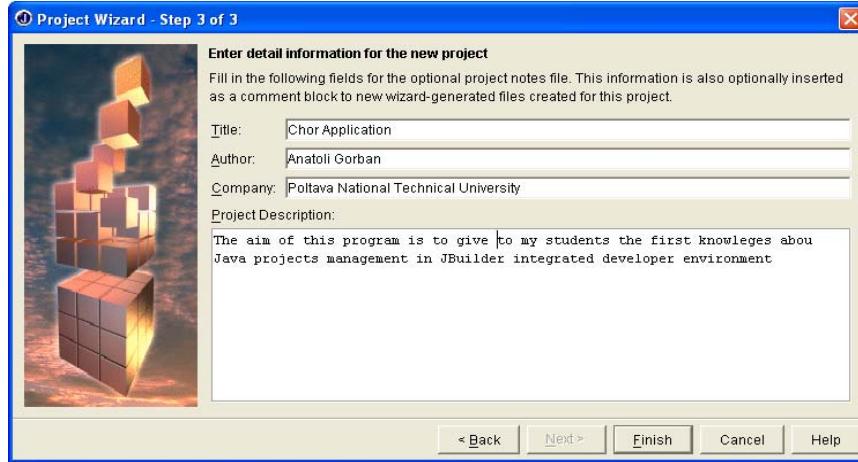
а такий після.



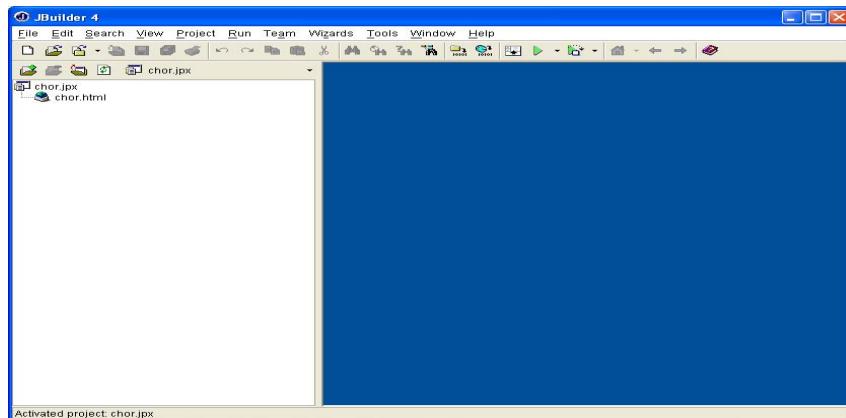
Решту полів програма заповнює сама. Тиснемо **Next** і переходимо до наступного кроку (**Step 2**)



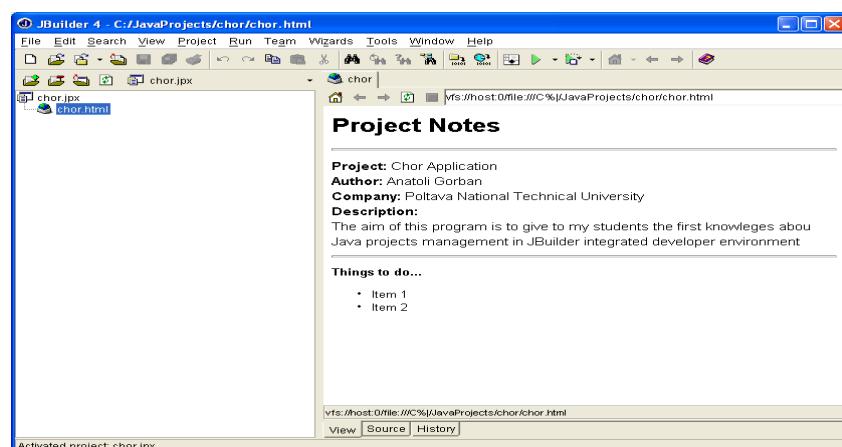
Залиште в цьому вікні все без зміни, воно в основному інформує вас про створені в проекті службові файли. Тисніть **Next**



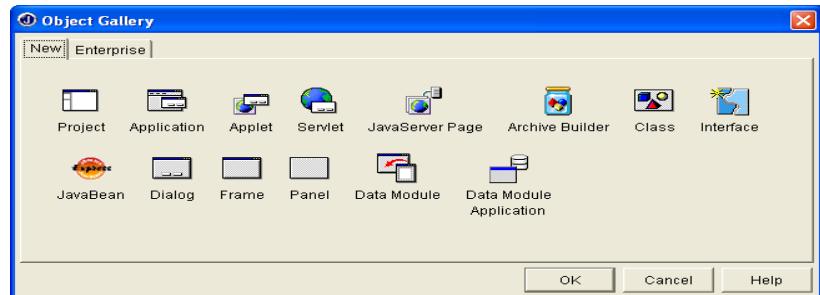
і впишіть у відповідні поля назву вашої аплікації, своє прізвище і установу, в якій працюєте. Опишіть коротко призначення аплікації. Все це зявиться пізніше в одному із файлів проекту. Тиснемо **Finish** і бачимо



Вікно інформує нас, що в проекті лише один файл [chor.html](#). Давайте відкриємо його



Знайома картина. Але куди ж скопіювати нашу програму з лістингу 3.2. Створимо необхідні для цього файли. Тиснемо **File -> New** і у вікні **Object Galery**



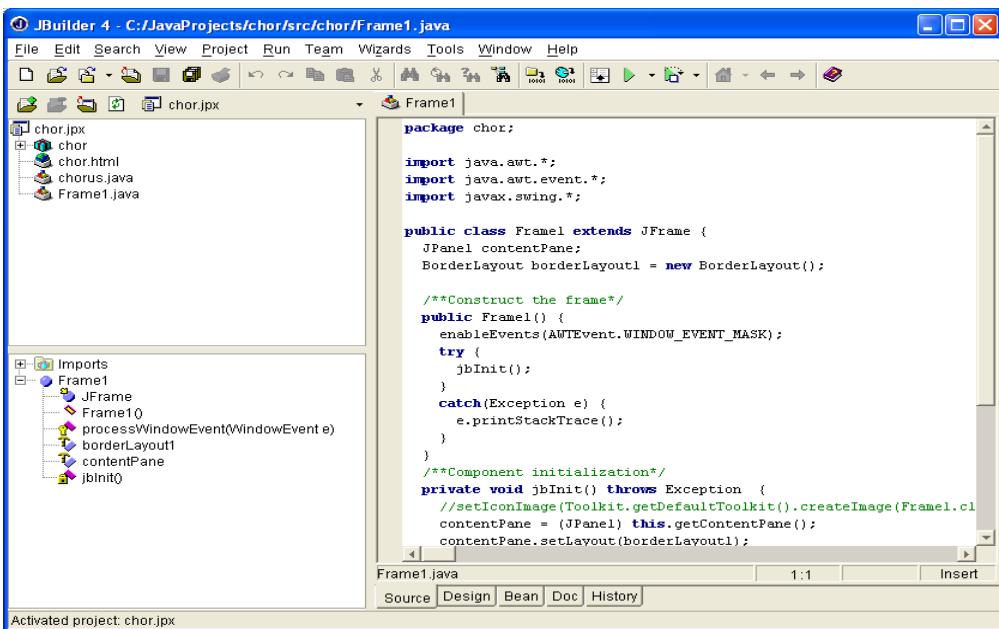
робимо **double click** на **Application**.



В поле **Class** я вписав ім'я класу **chorus**, який містить **main()** метод нашої аплікації. Тисну **Next** і бачу



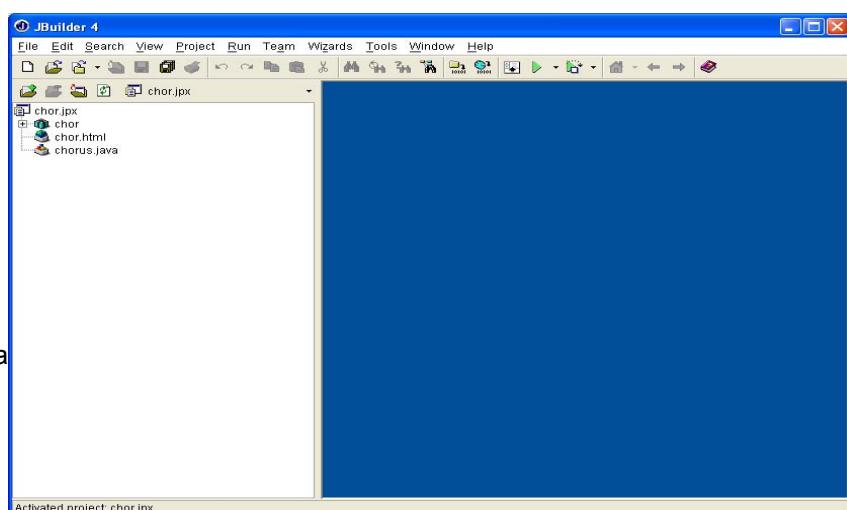
Залишаю це вікно без уваги. Воно служить для створення файлу форми, подібної до [Visual Basic](#) і [Delphi](#), але ми на перших порах не ю користуватися не будемо. Нам треба ще розібратися в конструкціях мови програмування [Java](#), а графічне середовище буде відволікати нашу увагу. Досить того, що ми в свій час награлися у [Visual Basic](#) і [Delphi](#). Однаке програма [Application Wizard](#) помимо нашої волі все ж таки створить такий файл. Більше того, вона створить шаблонний файл для програмного коду і в нього впише багато непотрібних для нас на перший випадок речей. Ось як ми будемо виходити із цієї ситуації. Тиснемо **Finish** і бачимо



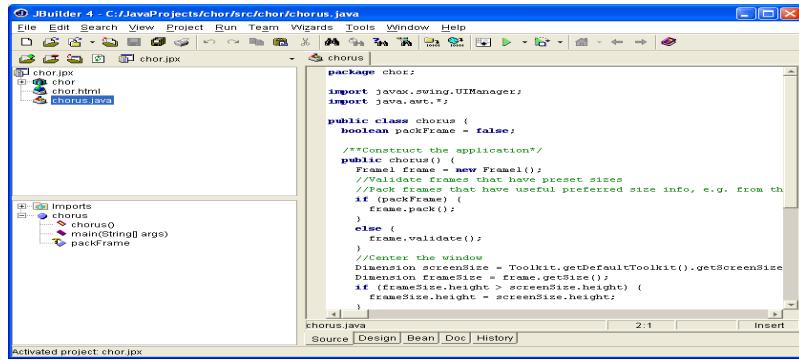
В правому вікні бачимо файл **Frame1**, про що свідчить іконка зверху. Цей файл, як уже говорилося, нам зараз непотрібний. Ми могли б на нього просто не звертати уваги, але

краще його взагалі видалити з проекту, щоб не заважав. У лівому верхньому вікні тиснемо правою кнопкою миші на **Frame1.java** і в контекстному меню вибираємо опцію **Delete**. І зразу на душі полегшало — нічого тобі лишнього.

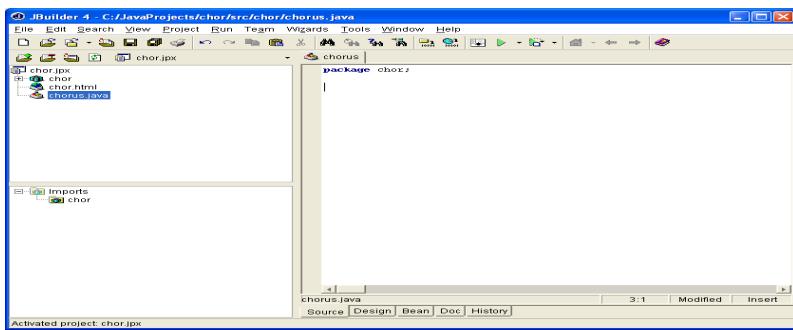
Навіть справа



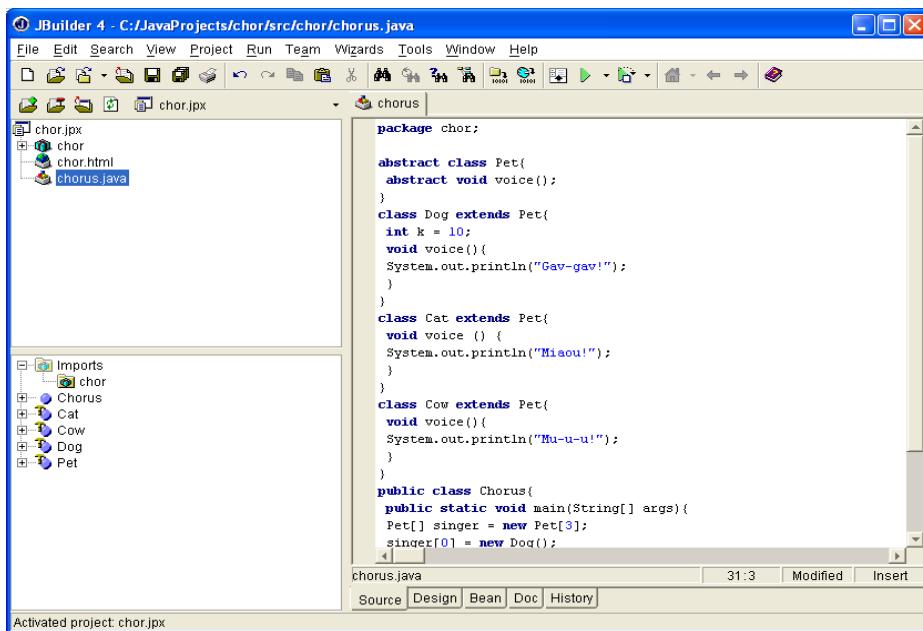
залишилося всього одне вікно. Відкриваємо через **double click** файл **chorus.java** і знову бачимо там багато незрозумілого, а головне непотрібного.



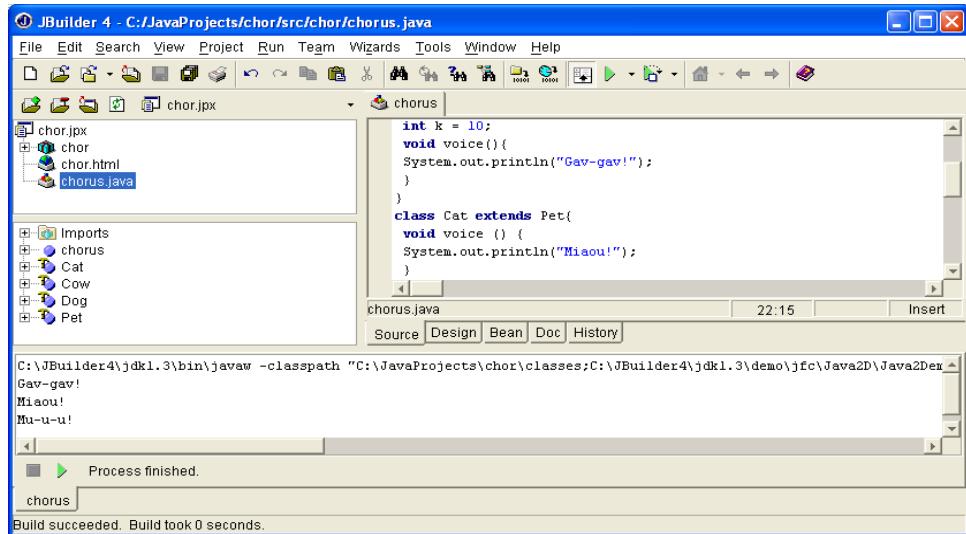
Видаляємо з цього файлу все, за виключенням першого рядка



і копіюємо в нього вміст лістингу 3.2. Зверніть увагу, в лівому нижньому вікні зявилася інформація про всі класи, що входять в аплікацію. Ви можете запустити її на виконання натиском зеленого трикутничка у **ToolBar** як в **Delphi** чи **Visual Basic**. Наші зусилля були варті того. Тепер ми можемо експериментувати з будь якими програмами, варто лише видалити в **chorus.java** один програмний код і скопіювати на його місце інший.



Ось що ми бачимо при запуску на виконання програми [chorus](#).



Результат зявився в нижньому вікні, яке має точно таку ж структуру, як і [Командна строка](#). Навчіться користуватися середовищем програмування **JBuilder** і це полегшить вам роботу з наступним матеріалом.

3.5. Абстрактні методи і класи

При описанні класу **Pet** ми не можемо задати в методі **voice ()** ніякого корисного алгоритму, оскільки у всіх тварин різні голоси. В таких випадках ми записуємо тільки заголовок метода і ставимо після закриваючої список параметрів дужки крапку з комою. Цей метод буде **абстрактним** (**abstract**), що необхідно вказати компілятору модифікатором **abstract**.

Якщо клас містить хоча один абстрактний метод, то створити його екземпляри, а тим більше використовувати їх не удасться. Такий клас становиться **абстрактним**, що обовязково треба вказати модифікатором **abstract**.

Як же використовувати абстрактні класи? Тільки породжуючи від них підкласи, в яких перевизначені абстрактні методи.

Для чого ж потрібні абстрактні класи? Чи не краще зараз же написати потрібні класи з повністю визначенними методами, а не наслідувати їх від абстрактного класа? Для відповіді знову звернемося до лістингу 3.2.

Хоча елементи масива **singer []** посилаються на підкласи **Dog**, **Cat**, **Cow**, але все-таки це змінні типу **Pet** і посилатися вони можуть тільки на поля і методи, описані в суперкласі **Pet**. Додаткові поля підкласу для них недоступні. Попробуйте звернутися, наприклад, до поля **k** класу **Dog**, написавши **singer[0].k**. Компілятор "скаже", що він не може реалізувати таку посилку. Тому метод, який реалізується в декількох підкласах, приходить виносити в суперклас, а якщо там його неможливо реалізувати, то оголосити абстрактним. Таким чином, абстрактні класи групуються на вершині ієрархії класів.

До речі, можна задати пусту реалізацію метода, просто поставивши пару фігурних дужок, нічого не написавши між ними, наприклад:

```
void voice(){}  
 
```

Получиться повноцінний метод. Але це штучне рішення, яке заплутає структуру класу. Замкнути ж ієрархію можна остаточними класами.

3.6. Остаточні члени і класи

Помітивши метод модифікатором `final`, можна заборонити його перевизначення в підкласах. Це зручно в цілях безпеки. Ви можете бути впевненими, що метод виконує ті дії, які ви задали. Якраз так визначені математичні функції `sin()`, `cos()` інші в класі `Math`. Ми впенені, що метод `Math.cos (x)` обчислює якраз косинус числа `x`. Розуміється, такий метод не може бути абстрактним.

Для повної безпеки, поля, що обробляються остаточними методами, належить зробити закритими (`private`).

Якщо ж помітить модифікатором `final` увесь клас, то його взагалі не можна буде розширювати. Так визначений, наприклад, клас `Math`:

```
public final class Math{ . . . }
```

Для змінних модифікатор `final` має зовсім інший зміст. Якщо помітити модифікатором `final` описання змінної, то її значення (а воно повинно бути обов'язково задано або тут же, або в блоці ініціалізації або в конструкторі) не можна змінити ні в підкласах, ні в самому класі. Змінна перетворюється в константу. Якраз так в мові `Java` визначаються константи:

```
public final int MIN_VALUE = -1, MAX_VALUE = 9999;
```

Згідно "Code Conventions" константи записуються прописними літерами, слова в них розділяються знаком підкреслювання.

На самій вершині ієрархії класів `Java` стоїть клас `Object`.

3.7. Клас `Object`

Якщо при описанні класу ми не вказуємо ніяке розширення, тобто не пишемо слово `extends` і ім'я класу за ним, як при описанні класу `Pet`, то `Java` вважає цей клас розширенням класу `Object`, і компілятор додисує це за нас:

```
class Pet extends Object{ . . . }
```

Можна записати це розширення і явно.

Сам же клас `Object` не являється нічим наслідником, від нього починається ієрархія любих класів `Java`. Зокрема, всі масиви — прямі наслідники класу `Object`.

Оскільки такий клас може містити тільки загальні властивості всіх класів, в нього включені лише декілька самих загальних методів, наприклад, метод `equals()`, що порівнює даний об'єкт на рівність з об'єктом, заданим в аргументі, і повертає логічне значення. Його можна використати так:

```
Object obj1 = new Dog(), obj2 = new Cat();
if (obj1.equals(obj2)) ...
```

Оцініть об'єктно-орієнтований дух цього запису: об'єкт `obj1` активний, він сам порівнює себе з другим об'єктом. Можна, звичайно, записати і `obj2.equals(obj1)`, зробивши активним об'єкт `obj2`, з тим же результатом.

Як указувалось в главі 1, посилання можна порівнювати на рівність і нерівність:

```
obj1 == obj2; obj1 != obj 2;
```

В цьому випадку співставляються адреси об'єктів, ми можемо узнати, чи не указують обидва посилання на один і той же об'єкт.

Метод `equals()` порівнює зміст об'єктів в їх поточному стані, фактично він реалізований у класі `Object` як тотожність: об'єкт рівний тільки самому собі. Тому його часто перевизначають в підкласах, більше того, правильно спроектовані, "добре виховані", класи повинні перевизначити методи класа `Object`, якщо їх не влаштовує стандартна реалізація.

Другий метод класу `Object`, який належить перевизначати в підкласах, — метод `toString()`. Цей метод без параметрів, який намагається зміст об'єкта претворити в рядок символів і повертає об'єкт класа `String`.

До цього методу виконуюча система `Java` звертається кожен раз, коли потрібно представити об'єкт у вигляді рядка, наприклад, в методі `printing`.

3.8. Конструктори класу

Ви уже звернули увагу на те, що в операції `new`, що визначає екземпляри класе, повторюється ім'я класе з дужками. Це схоже на звертання до методу, але що за "метод", ім'я котрого повністю співпадає з іменем класу?

Такий "метод" називається *конструктором класу* (`class constructor`). Його своєрідність заключається не тільки в імені. Перечислимо особливості конструктора.

- Конструктор єсть в будь-якому класі. Навіть якщо ви його не написали, компілятор Java сам створить *конструктор по замовчуванню* (`default constructor`), котрий, між іншим, пустий, він не робить нічого, крім виклику конструктора суперкласу.
- Конструктор виконується автоматично при створенні екземпляра класу, після розподілу пам'яті і обнулювання полів, але до початку використання створюваного об'єкта.
- Конструктор не повертає ніякого значення. Тому в його описанні не пишеться навіть слово `void`, але можна задати один із трьох модифікаторів `public`, `protected` або `private`.
- Конструктор не являється методом, він навіть не вважається членом класу. Тому його не можна наслідувати або перевизначати в підкласі.
- Тіло конструктора може починатися:
 - з виклику одного із конструкторів суперкласу, для цього записується слово `super()` з параметрами в дужках, якщо вони потрібні;
 - з виклику другого конструктора того ж класа, для цього записується слово `this()` з параметрами в дужках, якщо вони потрібні.
- Якщо ж `super()` на початку конструктора не указаній, то спочатку виконується конструктор суперкласа без аргументів, потім відбувається ініціалізація полів значеннями, указаними при їх оголошенні, а вже потім те, що записано в конструкторі.

У всьому останньому конструктор можна вважати звичайним методом, в ньому дозволяється записувати будь-які оператори, навіть оператор `return`, але тільки пустий, без всякого значення.

В класі може бути декілька конструкторів. Оскільки у них одне і те ж ім'я, що співпадає з іменем класу, то вони повинні відрізнятися типом і/або кількістю параметрів.

В наших прикладах ми ні разу не розглядали конструктори класів, тому що при створенні екземплярів наших класів викликався конструктор класу `Object`.

3.9. Операція `new`

Пора детальніше описати операцію з одним операндом, що позначається словом `new`. Вона застосовується для виділення пам'яті масивам і об'єктам.

В першому випадку в якості операнда указується тип елементів масиву і кількість його елементів у квадратних дужках, наприклад:

```
double a[] = new double[100];
```

В другому випадку операндом служить конструктор класу. Якщо конструктора в класі немає, то викликається конструктор по замовчуванню.

Числові поля класу одержують нульові значення, логічні поля — значення `false`, посилання — значення `null`.

Результатом операції `new` буде посилання на створений об'єкт. Це посилання може бути присвоєне змінній типу посилання на даний тип:

```
Dog k9 = new Dog();
```

але може використовуватися і безпосередньо

```
new Dog().voice();
```

Тут після створення безіменного об'єкта зразу виконується його метод `voice()`. Такий дивний запис зустрічається в програмах, написаних на `Java`, на кожному кроці

3.10. Статичні члени класу

Різні екземпляри одного класу мають повністю незалежні один від одного поля, що приймають різні значення. Зміна поля в одному екземплярі ніяк не впливає на те ж поле в другому екземплярі. В кожному екземплярі для таких полів виділяється своя комірка пам'яті. Тому такі поля називаються змінними екземпляра класа (*instance variables*) або змінними об'єкта.

Інколи треба визначити поле, спільне для всього класу, зміна котрого в одному екземплярі потягне зміну того ж поля у всіх екземплярах. Наприклад, ми хочемо в класі `Automobile` відзначити порядковий заводський номер автомобіля. Такі поля називаються змінними класу (*class variables*). Для змінних класу виділяється тільки одна комірка пам'яті, спільна для всіх екземплярів. Змінні класу створюються в `Java` модифікатором `static`. В лістингі 3.3 ми записуємо цей модифікатор при визначенні змінної `number`.

Лістинг 3.3. Статична змінна

```
class Automobile {
    private static int number;
    Automobile(){
        number++;
        System.out.println("From Automobile constructor:" + " number = " + number);
    }
}
public class AutomobileTest{
    public static void main(String[] args){
        Automobile lada2105 = new Automobile(),
        fordScorpio = new Automobile(),
        oka = new Automobile();
    }
}
```

Цікаво, що до статичних змінних можна звертатися з іменем класа, `Automobile.number`, а не тільки з іменем екземпляра, `lada2105.number`, причому це можна робити, навіть якщо не створено жодного екземпляра класу.

```
C:\>jdk1.3\bin\javac AutomobileTest.java
C:\>jdk1.3\bin\java AutomobileTest
From Automobile constructor: number = 1
From Automobile constructor: number = 2
From Automobile constructor: number = 3
C:\>
```

Для роботи з такими *статичними змінними* звичайно створюються *статичні методи*, помічені модифікатором **static**. Для методів слово **static** має зовсім інший зміст. Виконуюча система **Java** завжди створює в пам'яті тільки одну копію машинного коду метода, що розділяється всіма екземплярами, незалежно від того, статичний цей метод чи ні.

Основна особливість статичних методів — вони виконуються зразу у всіх екземплярах класу. Більше того, вони можуть виконуватися, навіть якщо не створений жоден екземпляр класа. Досить уточнити ім'я метода іменем класа (а не іменем об'єкта), щоб метод міг працювати. Якраз так ми користувалися методами класу **Math**, не створюючи його екземпляри, а просто записуючи **Math.abs(x)**, **Math.sqrt(x)**. Точні так же ми використовували метод **System.out.println()**. Та і методом **main()** ми користуємося, взагалі не створюючи ніяких об'єктів.

Тому статичні методи називаються *методами класу* (**class methods**), на відміну від нестатичних методів, що називаються *методами екземпляру* (**instance methods**).

Звідси витікають інші особливості статичних методів:

- в статичному методі не можна використовувати посилання **this** і **super**;
- в статичному методі не можна прямо, не створюючи екземплярів, посилатися на нестатичні поля і методи;
- статичні методи не можуть бути абстрактними;
- статичні методи перевизначаються в підкласах тільки як статичні.

Якраз тому в лістингі 1.5 ми помітили метод **f()** модифікатором **static**. Але в лістингі 3.1 ми працювали з екземпляром **b2** класу **Bisection2**, і нам не знадобилося оголошувати метод **f()** статичним.

Статичні змінні ініціалізуються ще до початку роботи конструктора, але при ініціалізації можна використовувати тільки константні вирази. Якщо ж ініціалізація вимагає складних обчислень, наприклад, циклів для задання значень елементам статичних масивів або звернення до методів, то ці обчислення заключають в блок, помічений словом **static**, який теж буде виконаний до запуску конструктора:

```
static int[] a = new int[10];
static {
    for(int k = 0; k < a.length; k++)
        a[k] = k * k;
}
```

Оператори, заключені в такий блок, виконуються лише один раз, при першому завантаженні класу, а не при створенні кожного екземпляра.

Тут уважний читач, напевно, піймав мене: "А говорив, що всі дії виконуються тільки за допомогою методів!" Каюсь: блоки статичної ініціалізації, і блоки ініціалізації екземпляра записуються зовні всіх методів і виконуються до початку виконання не то що метода, але навіть конструктора.

3.11. Клас Complex

Комплексні числа широко використовуються не лише в математиці. Вони часто застосовуються в графічних перетвореннях, в побудові фракталів, не говорячи уже про фізику і технічні дисципліни. Але клас, що описує комплексні числа, чомусь не включений у стандартну бібліотеку Java. Заповнимо цей пробіл.

Лістинг 2.4 довгий, але проглянути його треба уважно, при вивченні мови програмування дуже корисне читання програм на цій мові. Більш того, тільки програми і варто читати, пояснення автора лише заважають вникнути в зміст дій (жарт).

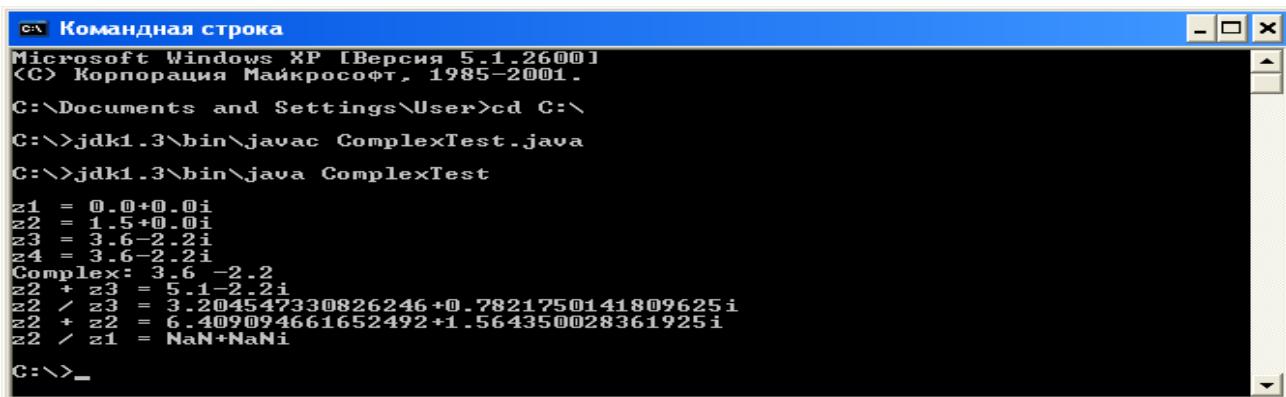
Лістинг 3.4. Клас Complex

```
class Complex {
    private static final double EPS = 1e-12; // Точність обчислень
    private double re, im; // Дійсна і уявна частини
    // Чотири конструктори
    Complex(double re, double im) { //Constructor 1
        this.re = re; this.im = im;
    }
    /* після виконання цього методу створиться об'єкт, який по формі нагадує масив або
    двовимірний вектор, координати якого будуть введені нами параметри re і im. Самі ж
    змінні класу re і im для нас недоступні – вони захищені директивою private.
    Схожість же вказаного об'єкта з масивом або вектором є тільки зовнішня, крім
    координат об'єкт має ще і методи, скриті в конструкції класу*/
    Complex(double re){this(re, 0.0); } //Constructor 2
    Complex(){this(0.0, 0.0); } //Constructor 3
    Complex(Complex z){this(z.re, z.im) ; } //Constructor 4
    // Методи доступу
    public double getRe(){return re;}
    public double getIm(){return im;}
    public Complex getZ(){return new Complex(re, im);}
    public void setRe(double re){this.re = re;}
    public void setIm(double im){this.im = im;}
    public void setZ(Complex z){re = z.re; im = z.im;}
    // Модуль і аргумент комплексного числа
    public double mod(){return Math.sqrt(re * re + im * im);}
    public double arg(){return Math.atan2(re, im);}
    // Перевірка: дійсне число?
    public boolean isReal(){return Math.abs(im) < EPS;}
    public void pr(){ // Виведення на екран
        System.out.println(re + (im < 0.0 ? "" : "+") + im + "i");
    }
    // Переизначення методів класа Object
    public boolean equals(Complex z){
        return Math.abs(re - z.re) < EPS &&
        Math.abs(im - z.im) < EPS;
    }
    public String toString(){
        return "Complex: " + re + " " + im;
    }
    // Методи, що реалізують операції +=, -=, *=, /=
    public void add(Complex z){re += z.re; im += z.im;}
    public void sub(Complex z){re -= z.re; im -= z.im;}
    public void mul(Complex z){
        double t = re * z.re - im * z.im;
        im = re * z.im + im * z.re;
        re = t;
    }
}
```

```

}
public void div(Complex z){
double m = z.mod();
double t = re * z.re - im * z.im;
im = (im * z.re - re * z.im) / m;
re = t / m;
}
// Методи, що реалізують операції +, -, *, /
public Complex plus(Complex z){
return new Complex(re + z.re, im + z.im);
}
public Complex minus(Complex z){
return new Complex(re - z.re, im - z.im);
}
public Complex asterisk(Complex z){
return new Complex(
re * z.re - im * z.im, re * z.im + im * z.re);
}
public Complex slash(Complex z){
double m = z.mod();
return new Complex(
(re * z.re - im * z.im) / m, (im * z.re - re * z.im) / m);
}
}
// Перевіримо роботу класу Complex
public class ComplexTest {
public static void main(String[] args){
Complex z1 = new Complex(), //Constructor 3, z1 = 0
z2 = new Complex(1.5), //Constructor 2, z2 = 1.5 - real number
z3 = new Complex(3.6, -2.2), //Constructor 1, z3 = 3.6 - 2.2i
z4 = new Complex(z3); //Constructor 4, z4 = z3
System.out.println(); // Залишаємо пустий рядок
System.out.print("z1 = "); z1.pr();      //z1 = 0.0 + 0.0i
System.out.print("z2 = "); z2.pr();      //z2 = 1.5 + 0.0i
System.out.print("z3 = "); z3.pr();      //z3 = 3.6 - 2.2i
System.out.print ("z4 = "); z4.pr();      //z4 = 3.6 - 2.2i
System.out.println(z4); // Працює метод toString() Complex: 3.6 -2.2
z2.add(z3); // z2 = z2 + z3
System.out.print("z2 + z3 = "); z2.pr(); //z2 + z3 = 5.1 - 2.2i
z2.div(z3); //z2 = z2/z3
System.out.print("z2 / z3 = "); z2.pr(); //z2/z3 = 1.303370787 + 0.185393258i
z2 = z2.plus(z2);
System.out.print("z2 + z2 = "); z2.pr(); // z2 + z2 = 2.606741574 + 0.370786516i
z3 = z2.slash(z1);
System.out.print("z2 / z1 = "); z3.pr(); /* на нуль не можна ділити. В будь який
іншій мові програмування ця програма б не спрацювала і компілятор вивів би
відповідне попередження. Java ж виконала всі дозволені дії, а про не дозволену дію
сигналізувала символом NaN - not a number */
}
}

```



```

С:\ Командная строка
Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.
C:\Documents and Settings\User>cd C:\
C:\>jdk1.3\bin\javac ComplexTest.java
C:\>jdk1.3\bin\java ComplexTest
z1 = 0.0+0.0i
z2 = 1.5+0.0i
z3 = 3.6-2.2i
z4 = 3.6-2.2i
Complex: 3.6 -2.2
z2 + z3 = 5.1-2.2i
z2 / z3 = 3.204547330826246+0.7821750141809625i
z2 + z2 = 6.409094661652492+1.564350028361925i
z2 / z1 = NaN+NaNi
C:\>_

```

Якщо ви уважно все прочитали і звірили результати уазані в моїх коментарях з виведеними на екран, то будете здивовані їх невідповідністю. Я запозичив цю програму з [Інтернет](#) і помітив, що два методи в класі **Complex** реалізовані невірно. [Справдилося мое передчуття, що програмісти не знають математики](#), а беручись за математичні задачі вводять в оману і інших. Таким чином, треба дуже критично відноситися до всього того, що знаходите в [Інтернет](#). А тепер попробуйте знайти і віправити помилку в двох методах класу **Complex**.

Клас комплекс можна реалізувати і по іншому, не вживаючи **this**. Одночасно я віправив помилки в реалізації двох помилкових методів. Чи ви це помітили?

```

class Complex {
    private static final double EPS = 1e-12; // Точність обчислень
    private double x, y; // Дійсна і уявна частини
    // Чотири конструктори
    Complex(double re, double im) { //Constructor 1
        x = re; y = im;
    }
    Complex(double re){x = re; y = 0.0; } //Constructor 2
    Complex(){x = 0.0; y = 0.0; } //Constructor 3
    Complex(Complex z){x =z.x; y = z.y ; } //Constructor 4
    // Методи доступу
    public double getRe(){return x;}
    public double getIm(){return y;}
    public Complex getZ(){return new Complex(x, y);}
    public void setRe(double re){x = re;}
    public void setIm(double im){y = im;}
    public void setZ(Complex z){x = z.x; y = z.y;}
    // Модуль і аргумент комплексного числа
    public double mod(){return Math.sqrt(x * x + y * y);}
    public double arg(){return Math.atan2(x, y);}
    // Перевірка: дійсне число?
    public boolean isReal(){return Math.abs(y) < EPS;}
    public void pr(){ // Виведення на екран
        System.out.println(x + (y < 0.0 ? "" : "+") + y + "i");
    }
    // Переизначення методів класа Object
    public boolean equals(Complex z){
        return Math.abs(x - z.x) < EPS && Math.abs(y - z.y) < EPS;
    }
    public String toString(){
        return "Complex: " + x + " " + y;
    }
    // Методи, що реалізують операції +=, -=, *=, /=
    public void add(Complex z){x += z.x; y += z.y;}
    public void sub(Complex z){x -= z.x; y -= z.y;}
}

```

```

public void mul(Complex z){
    double t = x * z.x - y * z.y;
    y = x * z.y + y * z.x;
    x = t;
}
public void div(Complex z){
    double m = z.mod()*z.mod();
    double t = x * z.x - y * z.y;
    y = (y * z.x - x * z.y) / m;
    x = t / m;
}
// Методи, що реалізують операції +, -, *, /
public Complex plus(Complex z){
    return new Complex(x + z.x, y + z.y);
}
public Complex minus(Complex z){
    return new Complex(x - z.x, y - z.y);
}
public Complex asterisk(Complex z){
    return new Complex(
        x * z.x - y * z.y, x * z.y + y * z.x);
}
public Complex slash(Complex z){
    double m = z.mod()*z.mod();
    return new Complex(
        (x * z.x - y * z.y) / m, (y * z.x - x * z.y) / m);
}
}

```

Щоб добре зрозуміти, як користуватися `this` випишемо всі методи, в яких він використовується і рядом альтернативний метод без `this`.

```

Complex(double re, double im) {
    this.re =; this.im = im;
}
/*this допомагає відрізити поля обєкта
re і im від переданих йому конструктором
параметрів re і im, позначених тими ж
символами*/
Complex(double re){this(re, 0.0); }
Complex(){this(0.0, 0.0); }
/*Фактично, this()(а не this) служить
для виклику попереднього конструктора, в
який передається нульова уявна частина.
Див. пункт 3.8*/
Complex(Complex z){this(z.re, z.im) ; }
/*Теж саме, змінюється лише спосіб
передачі параметрів у конструктор*/
public void setRe(double re){this.re = re;}
public void setIm(double im){this.im = im;}

```

```

Complex(double re, double im) {
    x = re; y = im;
}
/*Я позначив дійсну і уявну частину
комплексного числа як x і y, тому не
треба вживати this для створення
комплексного числа (обєкта) по переданим
в конструктор значенням re,im.*/

```

```

Complex(double re){x = re; y = 0.0; }
Complex(){x = 0.0; y = 0.0; }
/*А тут мені приходиться створювати
цілком нові конструктори*/
Complex(Complex z){x =z.x; y = z.y ; }

```

```

public void setRe(double re){x = re;}
public void setIm(double im){y = im;}
/*Тут же ми явно зекономили, завдяки
вдалим позначенням змінних класу і
параметрів конструкторів*/

```

Увага! `this` і `this()` - це різні речі.

3.12. Метод `main()`

Всяка програма, оформленена як **додаток** (`application`), повинна містити метод з іменем `main`. Він може бути один на всі додатки або міститися в деяких класах цього додатку, а може знаходитися і в кожному класі.

Метод `main()` записується як звичайний метод, може містити будь-які описання і дії, але він обовязково повинен бути відкритим (`public`), статичним (`static`), не повертати значення (`void`). Його аргументом обовязково повинен бути масив рядків (`String[]`). По традиції цей масив називають `args`, хоча ім'я може бути довільним.

Ці особливості виникають із-за того, що метод `main()` викликається автоматично виконуючою системою `Java` в самому початку виконання додатку. При виклику інтерпретатора `Java` вказується клас, де записаний метод `main()`, з якого треба почати виконання. Оскільки класів з методом `main()` може бути декілька, можна побудувати додаток з додатковими точками входу, починаючи виконання додатку в різних ситуаціях із різних класів.

Часто метод `main()` заносять в кожний клас з метою налагоджування. В цьому випадку в метод `main()` включають тести для перевірки роботи всіх методів класа.

При виклику інтерпретатора `Java` можна передати в метод `main()` декілька параметрів, які інтерпретатор заносить в масив рядків. Ці параметри перечисляються в рядку виклику `Java` через пробіл зразу після імені класу. Якщо ж параметр містить пробіли, треба взяти його в лапки. Лапки не будуть включені в параметр, це тільки обмежувачі.

Все це легко зрозуміти на прикладі лістинга 3.5, в якому записана програма, що виводить параметри, котрі передаються в метод `main()` при запуску.

Лістинг 3.5. Передача параметрів в метод `main()`

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

Як бачимо, ім'я класа не входить в число параметрів. Воно і так відоме у методі `main()`.

Знавцям С/С++

Оскільки в `Java` ім'я файла завжди співпадає з іменем класу, котрий містить метод `main()`, воно не заноситься в `args[0]`. Доступ до змінних середовища дозволений не завжди і здійснюється іншим способом. Деякі значення можна проглянути так: `System.getProperties().list(System.out);`.

Така властивість метода дозволяє передавати в програму значення, які потім можна присвоїти змінним

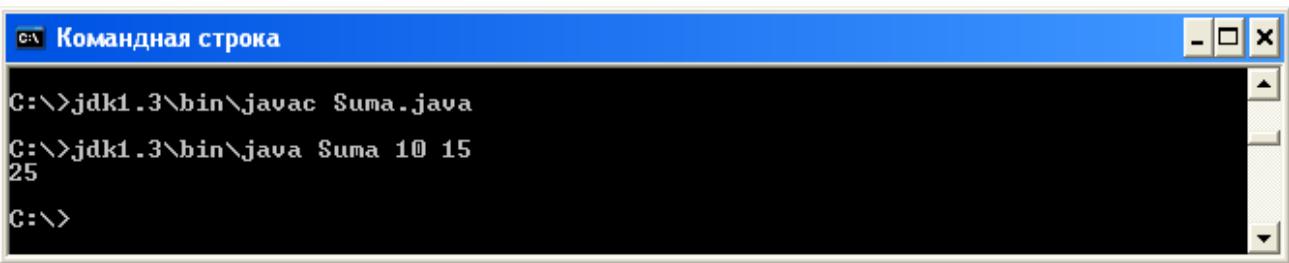
```
C:\>jdk1.3\bin\java Echo This words I have passed in main() function
args[0]=This
args[1]=words
args[2]=I
args[3]=have
args[4]=passed
args[5]=in
args[6]=main()
args[7]=function
C:\>
```

(подібно до `cin>>` в C++, `input` в Basic чи `read` в Pascal. Випробуйте наступні програми.

При запуску цієї програми введіть два числа, наприклад 10 і 15.

```
class Suma {
public static void main(String[] args) {
Integer x = new Integer(args[0]);
Integer y = new Integer(args[1]);
int x1 = x.intValue();
int y1 = y.intValue();
System.out.println(String.valueOf(x1+y1));
}
}
```

Не лякайтесь складності цієї програми. Вивчіть методи класів `Integer` і `String` і не матимете проблем з перетворюванням типів у `Java`.



```
C:\>jdk1.3\bin\javac Suma.java
C:\>jdk1.3\bin\java Suma 10 15
25
C:\>
```

3.13. Де змінні видимі

В мові `Java` нестатичні змінні можна оголошувати в будь-якому місці коду між операторами. Статичні змінні можуть бути тільки полями класу, а значить, не можуть оголошуватися всередині методів і блоків. Яка ж область видимості (scope) змінних? З яких методів ми можемо звернутися до тієї чи іншої змінної? В яких операторах використовувати? Розглянемо на прикладі лістинга 3.6 різні випадки оголошення змінних.

Лістинг 3.6. Видимість і ініціалізація змінних

```
class ManyVariables{
    static int x = 9, y; // Статичні змінні – поля класа
    // Вони видимі у всіх методах і блоках класа
    // Змінна у отримує значення 0
    static{ // Блок ініціалізації статичних змінних
        // Виконується один раз при перому завантаженні класа після
        // ініціалізації в оголошеннях змінних
        x = 99; // Оператор виконується зовні всякого метода!
    }
    int a = 1, p; // Нестатичні змінні – поля екземпляра
    // Відомі у всіх методах і блоках класа, в яких вони
    // не перекріти другими змінними з тим же іменем
    // Змінна p отримує значення 0
    { // Блок ініціалізації екземпляра
        // Виконується при створенні кожного екземпляра після
        // ініціалізації при оголошеннях змінних
        p = 999; // Оператор виконується зовні всякого метода!
    }
    static void f(int b){ // Параметр метода b – локальна
        // змінна, відома тільки всередині метода
    }
}
```

```

int a = 2; // Це друга змінна з тим же іменем "a"
// Вона відома тільки відома тільки в методі f()
// тут перекриве першу "a"
int c; // Локальна змінна, відома тільки в методі f()
//Не отримує ніякого початкового значення
//і повинна бути визначена перед застосуванням
{ int c = 555; // Помилка! Спроба повторного оголошення
int x = 333; // Локальна змінна, відома тільки в цьому блоці
}
// Тут змінна x уже невідома
for (int d = 0; d < 10; d++){
// Змінна цикла d відома тільки в циклі
int a = 4; // Помилка!
int e = 5; // Локальна змінна, відома тільки в циклі for
e++; // Ініціалізується при кожному виконанні цикла
System.out.println("e = " + e) ; // Виводиться завжди "e = 6"
}
// Ту змінні d і e невідомі
}
public static void main(String[] args){
int a = 9999; // Локальна змінна, відома тільки всередині метода main()
f (a);
}
}

```

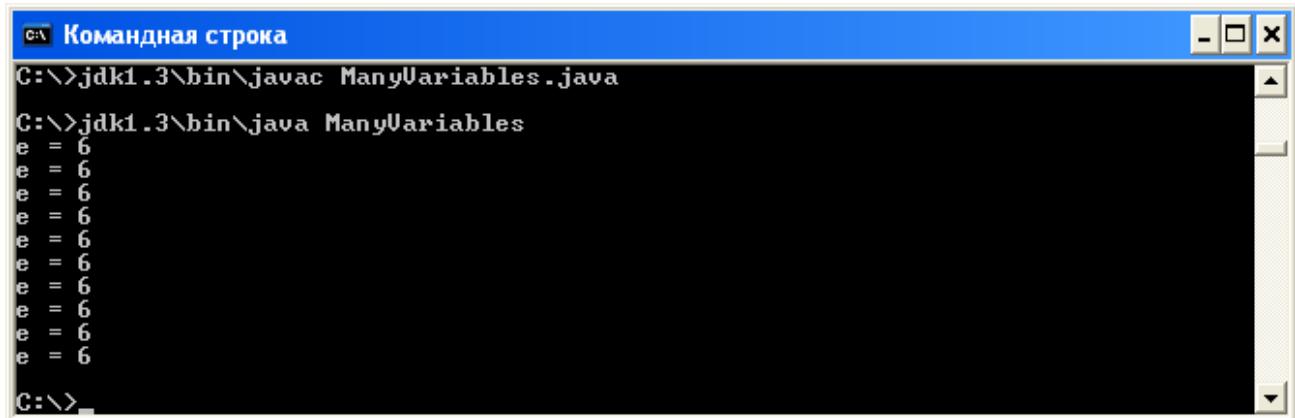
Зверніть увагу на те, що змінним класу і екземпляра неявно присвоюються нульові значення. Символи неявно одержують значення '\u0000', логічні змінні — значення false, посилання одержують неявно значення null.

Локальні ж змінні неявно не ініціюються. Їм повинні або явно присвоюватися значення, або вони повинні визначатися до першого використання. На щастя, компілятор помічає невизначені локальні змінні і повідомляє про них.

Увага

Поля класу при оголошенні обнулюються, локальні змінні автоматично не ініціалізуються.

В лістингі 3.6 звілася ще одна нова конструкція: блок ініціалізації екземпляра (instance initialization). Це просто блок операторів в фігурних дужках, але записується він зовні всякого метода, прямо в тілі класу. Цей блок виконується при створенні кожного екземпляра, після ініціалізації при оголошенні змінних, але до виконання конструктора. Він відіграє таку ж роль, як і static-блок для статичних змінних. Для чого ж він потрібний, адже весь його зміст можна написати на початку конструктора? В тих випадках, коли конструктор написати не можна, а саме, в безіменних внутрішніх класах.



```

C:\>jdk1.3\bin\javac ManyVariables.java
C:\>jdk1.3\bin\java ManyVariables
e = 6
e = 6
e = 6
e = 6
e = 6
e = 6
e = 6
e = 6
e = 6
e = 6
C:\>_

```

3.16. Заключення

Після прочитання цієї глави ви отримали уявлення про сучасну парадигму програмування — об'єктно-орієнтоване програмування і реалізації цієї парадигми в мові [Java](#).

Не біда, якщо ви не усвоїли зразу принципи ООП. Для вироблення "об'єктного" погляду на програмування потрібні час і практика. Подальші уроки якраз і дадуть вам цю практику. Але спочатку необхідно ознайомитися з важливими поняттями мови [Java](#) — пакетами і інтерфейсами.

Лабораторна робота 2. Створення класів

Куди б ви не звернулися з серйозним питанням, у вас попросять предявити паспорт, де зафіковане ваше прізвище, ім'я і по батькові, дата і місце народження, стать, місце проживання, номер паспорта та ким і коли він виданий. Для комп'ютерної обробки даних про людей в якійсь установі чи то в масштабі всієї країни можна створити запис, полями якого будуть всі перечислені вище елементи.

Ось як би ми це зробили

Basic	Pascal	C++
<pre>Type Homo Name(1 to 3) as String BirthDate(1 to 3) as Integer BirstPlace(1 to 4) as String Sex as String Address(1 to 4) as String PassNumber as String PassEmitter as String PassEmissionDate(1 to 3) as Integer End Type</pre>	<pre>type Homo = record Name: array [1..3] of String; BirthDate: array[1 .. 3] of Integer; BirstPlace: array [1..4] of String; Sex: String; Address: array [1..4] of String; PassNumber: String; PassEmitter: String; PassEmissionDate: array[1 .. 3] of Integer; End;</pre>	<pre>struct Homo { string[3] Name; int[3] BirthDate; string[4] BirstPlace; string Sex; string[4] Address; string PassNumber; string PassEmitter; int[3] PassEmissionDate; };</pre>

Примітка. У випадку C++ під [string](#) розуміємо не базовий тип, а вбудований клас.

А як це все робиться у [Java](#). Автори цієї мови, навчені піввіковим досвідом програмування, дивилися далі. Сучасні методи обробки даних не обмежуються наведеними вище даними про людину. Так, при сплаті певних платежів або при відкритті рахунку в банку вас попросять предявити ідентифікаційний код платника податків. На прийомі у лікаря його цікавить не ваша дата народження, а вік. Якщо ви будете записуватися на прийом до лікаря через комп'ютерну базу даних, ваш запис крім полів даних повинен містити і метод, який по поточній даті і по даті народження автоматично визначає ваш вік. Але тоді це вже не просто запис, а щось більше, ви вже здогадуєтесь — це клас. Поля класу повинні бути захищені від постороннього доступу з програми, частиною якої стає ваш клас при взаємодії з інформаційною системою. Тобто, треба заборонити цій програмі зчитувати дані з вашого класу, наприклад, так:

[Homo.Address](#),

адже таким самим способом з програми хтось може і змінити вашу адресу. А це створить вам серйозні проблеми — у вашому паспорті написано одне, а в інформаційній системі інше. Так можна і квартиру втратити.

Але якщо зовсім заборонити зчитувати дані, то як же інформаційна система буде їх обробляти? Вихід знайдено в доступі до даних через методи класу, написані таким чином, щоб дані можна було зчитувати, але не змінювати зовні.

Перед вами ставиться наступне завдання.

1. Створити клас **Homo** з перечисленими вище полями і методами доступу до них та методом обчислення віку по поточній даті і даті народження. Вік округляти до найближчого цілого.
2. Створити підклас **Student** з додатковими полями **University**, **Faculty**, **Speciality**, **Year** і методу підрахунку середнього балу по пяти дисциплінам : **Computer Science**, **Calculus**, **Discrete Mathematic**, **Algebra**, **Geometry**. Вважайте, що ваша стипендія пропорціональна цьому рейтингу – це ваш спосіб існування, ваш доход, яким може зацікавитися податкова інспекція.
3. Піднімемося на ступеньку вище і створимо ще два підкласи: **Doctor** і **Dealer**. Доход першого залежить від його досвіду і пропорціональний віку, а другого від кількості проданих автомобілів. Подумайте, чи не варто створити клас **Homo** абстрактним.

Візьміть за зразок класи **Pet**, **Dog**, **Cat** і **Cow** з цієї лекції і напишіть програму, яка б виводила доход кожного (студента, лікаря і торговця) з указанням професії і способу отримання доходу. Налаштуйте програму і випробуйте її.

Працюйте самостійно. Ваші програми будете пояснювати на лекціях і демонструвати в лабораторії. У файлі **System.txt** можете подивитися, як написаний клас **System** – дуже цікаво.

Програмування у Java

Урок 4. Класи-оболонки

- Числові класи
- Клас Boolean
- Клас Character
- Клас BigInteger
- Клас BigDecimal
- Клас Class

4.1. Класова природа Java

Java — повністю об'єктно-орієнтована мова. Це означає, що все, що тільки можна, в Java представлено об'єктами.

Вісім примітивних типів порушують це правило. Вони залишені в Java із-за багаторічної звички до чисел і символів. Та їх арифметичні дії зручніше і скоріше виконувати зі звичайними числами, а не з об'єктами класів.

Але і для цих типів у мові Java є відповідні класи — *класи-оболонки* (*wrapper*) примітивних типів. Звичайно, вони призначені не для обчислень, а для дій, типових при роботі з класами — створення об'єктів, перетворення об'єктів, одержання числових значень об'єктів у різних формах і передачі об'ємів у методи по посиланню.

На рис. 4.1 показана одна із віток ієрархії класів Java. Для кожного примітивного типу є відповідний клас. Числові класи мають спільного предка — абстрактний клас **Number**, в якому описані шість методів, які повертають числове значення, що міститься в класі, приведене до відповідного примітивного типу: `byteValue ()`, `doubleValue ()`, `floatValue ()`, `intValue ()`, `longValue ()`, `shortValue ()`. Прогляньте уважно файл `Number.html` у папці `java_lang` і ще раз продумайте механізм використання абстрактних класів.

Указані вище методи перевизначені у кожному із шести числових класів-оболонок.

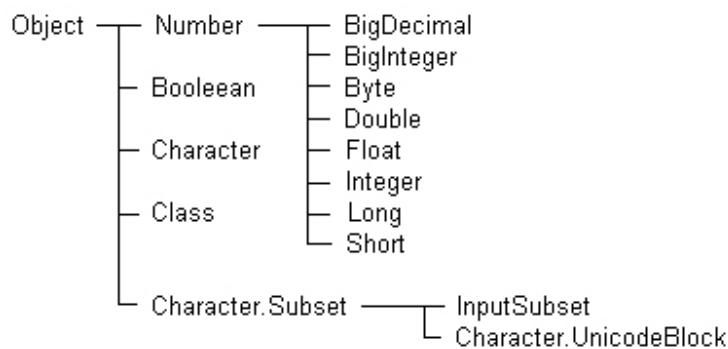


Рис. 4.1.

Помимо методу порівняння об'єктів `equals()`, перевизначеного із класу **Object**, всі описані в цій лекції класи, крім **Boolean** і **Class**, мають метод `compareTo ()`, який порівнює числове значення, що міститься в даному об'єкті, з числовим значенням об'єкту — аргументу методу `compareTo()`. В результаті роботи методу отримується ціле значення:

- 0, якщо значення рівні;
- від'ємне число (-1), якщо числове значення в даному об'єкті менше, ніж в об'єкті-аргументі;
- додатне число (+1), якщо числове значення в даному об'єкті більше числового значення, що

міститься в аргументі.

4.2. Числові класи

В кожному із шести числових класів-оболонок є статичні методи перетворення рядка символів типу `String`, котрий представляє число, у відповідний примітивний тип: `Byte.parseByte()`, `Double.parseDouble()`, `Float.parseFloat()`, `Integer.parseInt()`, `Long.parseLong()`, `Short.parseShort()`. Вихідний рядок типу `String`, як завжди у статичних методах, задається як аргумент методу. Ці методи корисні при введенні даних в поля введення, обробці параметрів командного рядка, т. е. тобто скрізь, де числа представляються рядками цифр зі знаками плюс або мінус і десятковою точкою.

В кожному із цих класів є статичні константи `MAX_VALUE` і `MIN_VALUE`, котрі показують діапазон числових значень відповідних примітивних типів. В класах `Double` і `Float` є ще константи `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `NaN`, про які йшла мова раніше, і логічні методи перевірки `isNaN()`, `isInfinite()`.

Якщо ви добре знаєте двійкове представлення дійсних чисел, то можете скористатися статичними методами `floatToIntBits()` і `doubleToLongBits()`, котрі перетворюють дійсне значення в ціле. Дійсне число задається як аргумент методу. Потім ви можете змінити окремі біти побітними операціями і перетворити змінене ціле число назад у дійсне значення методами `intBitsToFloat()` і `longBitsToDouble()`.

Статичними методами `toBinaryString()`, `toHexString()` і `toOctalString()` класів `Integer` і `Long` можна перетворити цілі значення типів `int` і `long`, задані як аргумент методу, в рядок символів, котрий показує двійкове, шістнадцятеричне або восьмеричне представлення числа.

Передивіться файли `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short` у папці `java.lang\Number` і научіться користуватися методами цих класів.

В лістингі 4.1 показано застосування цих методів, а на рис. 4.2 результат виконання програми

Лістинг 4.1. Методи числових класів

```
class NumberTest{
    public static void main(String[] args){
        int i = 0;
        short sh = 0;
        double d = 0;
        Integer k1 = new Integer(55);
        Integer k2 = new Integer(100);
        Double d1 = new Double(3.14);
        try{
            i = Integer.parseInt(args[0]);
            sh = Short.parseShort(args[0]);
            d = Double.parseDouble(args[1]);
            d1 = new Double(args[1]);
            k1 = new Integer(args[0]);
        }catch(Exception e){}
        double x = 1.0/0.0;
        System.out.println("i = " + i) ;
        System.out.println("sh - " + sh) ;
        System.out.println("d = " + d) ;
        System.out.println("k1.intValue() = " + k1.intValue());
        System.out.println("d1.intValue() = " + d1.intValue());
        System.out.println("k1 > k2? " + k1.compareTo(k2));
        System.out.println ("x = " + x);
        System.out.println("x isNaN? " + Double.isNaN(x));
        System.out.println("x isInfinite? " + Double.isInfinite(x));
    }
}
```

```

System.out.println("x == Infinity? " +
(x == Double.POSITIVE_INFINITY) );
System.out.println("d = " + Double.doubleToLongBits(d));
System.out.println("i = " + Integer.toBinaryString(i));
System.out.println("i = " + Integer.toHexString(i));
System.out.println("i = " + Integer.toOctalString(i));
}

```

```

Microsoft Windows XP [Версия 5.1.2600]
© Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\User>cd C:\

C:\>jdk1.3\bin\javac NumberTest.java

C:\>jdk1.3\bin\java NumberTest -20456 45.758
i = -20456
sh = -20456
d = 45.758
k1.intValue() = -20456
d1.intValue() = 45
k1 > k2? -1
x = Infinity
x isNaN? false
x isInfinite? true
x == Infinity? true
d = 4631636683301662491
i = 111111111111111011000000011000
i = fffffb018
i = 37777730030

C:\>_

```

Рис. 4.2

Методи [parseInt\(\)](#) і конструктори класів вимагають обробки виключень, тому в лістинг 4.1 вставлено блок [try{} catch\({}\)](#). Обробку виключчних ситуацій ми розберемо в лекції 16.

Передивіться файли [Byte](#), [Double](#), [Float](#), [Integer](#), [Long](#), [Short](#) у папці [java.lang](#) і научіться користуватися методами цих класів.

4.3. Клас Boolean

Це дуже невеликий клас, призначений головним чином для того, щоб передавати логічні значення в методи по посиланню.

Конструктор [Boolean \(String s\)](#) створює об'єкт, котрий містить значення [true](#), якщо рядок [s](#) рівний "true" в будь-якій комбінації регістрів літер, і значення [false](#) — для будь-якого іншого рядка.

Логічний метод [booleanValue\(\)](#) повертає логічне значення, що зберігається в об'єкті.

Передивіться файл [Boolean](#) у папці [java.lang](#) і научіться користуватися методами цього класу.

4.4. Клас Character

В цьому класі зібрані статичні константи і методи для роботи з окремими символами.

Статичний метод `digit(char ch, int radix)` переводить цифру `ch` системи числення з основою `radix` в її числове значення типу `int`.

Статичний метод `forDigit(int digit, int radix)` робить обернене перетворення цілого числа `digit` у відповідну цифру (тип `char`) у системі числення з основою `radix`.

Основа системи числення повинна знаходитися в діапазоні від `Character.MIN_RADIX` до `Character.MAX_RADIX`.

Метод `toString()` переводить символ, котрий міститься в класі, в рядок з тим же символом.

Статичні методи `toLowerCase()`, `toUpperCase()`, `toTitleCase()` повертають символ, що міститься в класі, у вказаному регістрі. Останній із цих методів слугує для правильного переводу у верхній регістр чотирьох кодів `Unicode`, котрі не виражаються одним символом.

Декілька статичних логічних методів перевіряють різні характеристики символу, переданого в якості аргументу методу:

- `isDefined()` — виясняє, чи визначений символ в кодуванні `Unicode`;
- `isDigit()` — перевіряє, чи являється символ цифрою `Unicode`;
- `isIdentifierIgnorable()` — виясняє, чи можна використовувати символ в ідентифікаторах;
- `isISOControl()` — визначає, чи є символ управлюючим;
- `isJavaIdentifierPart()` — виясняє, чи можна використовувати символ в ідентифікаторах;
- `isJavaIdentifierStart()` — виясняє, чи може символ бути на початку ідентифікатору;
- `isLetter()` — перевіряє, чи являється символ літерою `Java`;
- `isLetterOrDigit()` — перевіряє, чи являється символ літерою або цифрою `Unicode`;
- `isLowerCase()` — визначає, чи символ записаний в нижньому регістрі;
- `isSpaceChar()` — виясняє, чи є символ пробілом в `Unicode`;
- `isTitleCase()` — перевіряє, чи являється символ титульним;
- `isUnicodeIdentifierPart()` — виясняє, чи можна використовувати символ в іменах `Unicode`;
- `isUnicodeIdentifierStart()` — перевіряє, чи являється символ літерою `Unicode`;
- `isUpperCase()` — визначає, чи символ записаний у верхньому регістрі;
- `isWhiteSpace()` — виясняє, чи являється символ пробілом.

Точні діапазони управлюючих символів, поняття верхнього і нижнього регістра, титульного символу, пробільних символів, краще всього подивитися по документації [Java API](#).

Лістинг 4.2 демонструє використання цих методів, а на рис. 4.3 показаний результат цієї програми.

Лістинг 4.2. Методи класу Character в програмі CharacterTest

```
class CharacterTest{
    public static void main(String[] args) {
        char ch = '9';
        Character c = new Character(ch);
        System.out.println("ch = " + ch);
        System.out.println("c.charValue() = " + c.charValue());
        System.out.println("number of 'A' = " + Character.digit('A', 16));
        System.out.println("digit for 12 = " + Character.forDigit(12, 16));
        System.out.println("ch = " + c.toString());
        System.out.println("ch isDefined? " + Character.isDefined(ch));
        System.out.println("ch isDigit? " + Character.isDigit(ch));
```

```

System.out.println("ch isIdentifierIgnorable? " +
Character.isIdentifierIgnorable(ch));
System.out.println("ch isISOControl? " + Character.isISOControl(ch));
System.out.println("ch isJavaIdentifierPart? " +
Character.isJavaIdentifierPart(ch));
System.out.println("ch isJavaIdentifierStart? " +
Character.isJavaIdentifierStart(ch));
System.out.println("ch isLetter? " + Character.isLetter(ch));
System.out.println("ch isLetterOrDigit? " + Character.isLetterOrDigit(ch));
System.out.println("ch isLowerCase? " + Character.isLowerCase(ch));
System.out.println("ch isSpaceChar? " + Character.isSpaceChar(ch));
System.out.println("ch isTitleCase? " + Character.isTitleCase(ch));
System.out.println("ch isUnicodeIdentifierPart? " +
Character.isUnicodeIdentifierPart(ch));
System.out.println("ch isUnicodeIdentifierStart? " +
Character.isUnicodeIdentifierStart(ch));
System.out.println("ch isUpperCase? " + Character.isUpperCase(ch));
System.out.println("ch isWhitespace? " + Character.isWhitespace(ch)); }
}

```

Рис. 4.3

В клас [Character](#) вкладені класи [Subset](#) і [UnicodeBlock](#), причому клас [Unicode](#) і ще один клас, [inputSubset](#), являється розширенням класу [Subset](#), як це видно на рис. 4.1. Об'єкти цього класу містять підмножини [Unicode](#).

Передивіться файл [Character](#) у папці [java.lang](#) і визначте, чи всі методи цього класу були задіяні у попередній програмі.

Разом з класами-оболонками зручно розглянути два класи для роботи з як завгодно великими числами.

4.5. Клас BigInteger

Всі примітивні цілі тип мають обмежений діапазон значень. В цілочисельній арифметиці [Java](#) немає

переповнення, цілі числа приводяться по модулю, рівному діапазону значень. Для того щоб можна було виконувати цілочисельні обчислення з будь-якою розрядністю, у склад Java API введений клас `BigInteger`, котрий зберігається в пакеті `java.math`. Цей клас розширяє клас `Number`, значить, в ньому перевизначені методи `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`). Методи `byteValue()` і `shortValue()` не перевизначені, а прямо наслідуються від класу `Number`.

Дії з об'єктами класу `BigInteger` не приводять ні до переповнення, ні до приведення по модулю. Якщо результат операції великий, то число розрядів просто збільшується. Числа зберігаються в двійковій формі з додатковим кодом. Перед виконанням операції числа вирівнюються по довжині розповсюдженням знакового розряду.

Шість конструкторів класу створюють об'єкт класу `BigInteger` із рядка символів (знака числа і цифр) або із масиву байтів. Дві константи — `ZERO` і `ONE` — моделюють нуль і одиницю в операціях з об'єктами класу `BigInteger`. Метод `toByteArray()` перетворює об'єкт в масив байтів.

Більшість методів класу `BigInteger` моделюють цілочисельні операції і функції, повертуючи об'єкт класу `BigInteger`:

- `abs()` — повертає об'єкт, котрий містить абсолютну величину числа, яке зберігається в даному об'єкті `this`;
- `add(x)` — операція `this + x`;
- `and(x)` — операція `this & x`;
- `andNot(x)` — операція `this & (~x)`;
- `divide (x)` — операція `this / x`;
- `divideAndRemainder(x)` — повертає масив із двох об'єктів класу `BigInteger`, який містить частку і остачу від ділення `this` на `x`;
- `gcd(x)` — найбільший спільний дільник абсолютних значень об'єкту `this` і аргументу `x`;
- `max(x)` — найбільше із значень об'єкту `this` і аргумента `x`;
- `min(x)` — найменше із значень об'єкту `this` і аргументу `x`;
- `mod(x)` — остача від ділення об'єкту `this` на аргумент методу `x`;
- `modInverse(x)` — остача від ділення числа, оберненого об'єкту `this`, на аргумент `x`;
- `modPow(n, m)` — остача від ділення об'єкта `this`, піднесеного до степеня `n`, на `m`;
- `multiply (x)` — операція `this * x`;
- `negate()` — зміна знаку числа, що зберігається в об'єкті;
- `not()` — операція `~this`;
- `or(x)` — операція `this | x`;
- `pow(n)` — операція піднесення числа, яке зберігається в об'єкті, до степеня `n`;
- `remainder(x)` — операція `this % x`;
- `shiftLeft (n)` — операція `this « n`;
- `shiftRight (n)` — операція `this » n`;
- `signum()` — функція `sign (x)`;
- `subtract (x)` — операція `this - x`;
- `xor(x)` — операція `this ^ x`.

В лістингі 4.3 приведені приклади використання даних методів

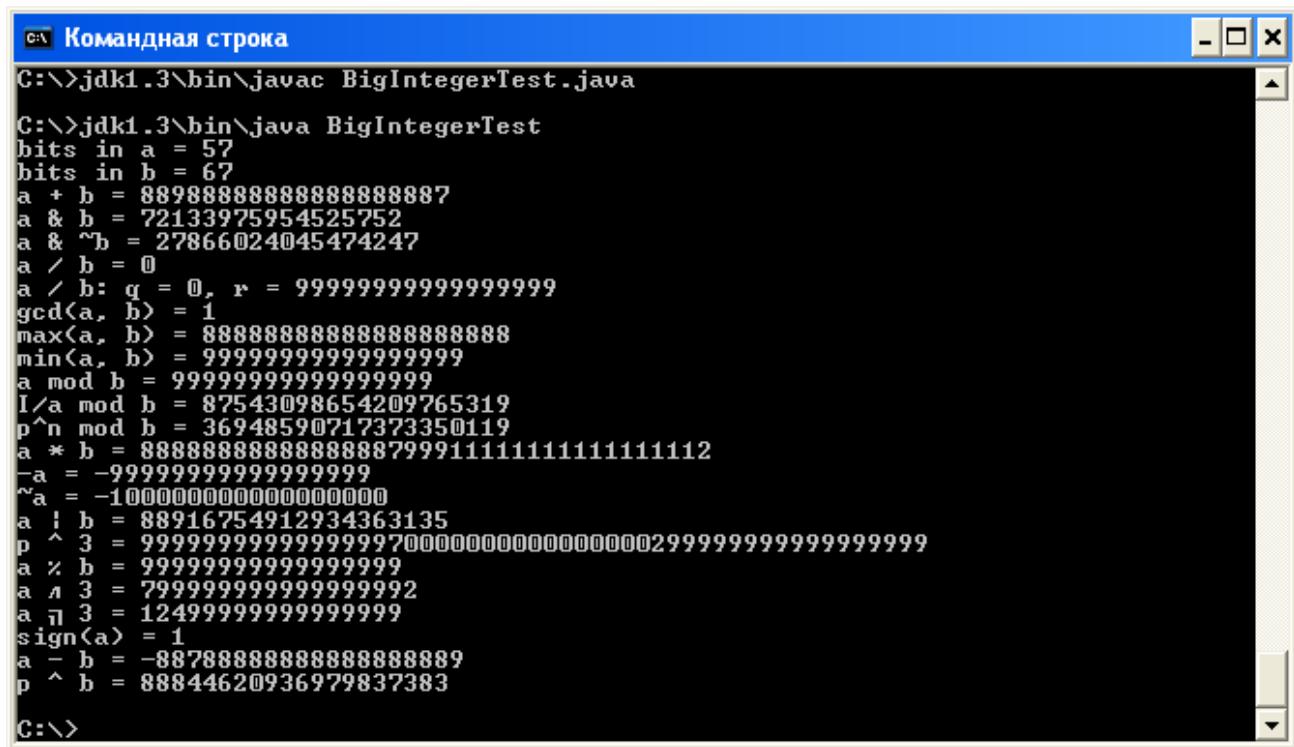
Лістинг 4.3. Методи класу `BigInteger` в програмі `BigIntegerTest`

```
import java.math.BigInteger;
class BigIntegerTest{
    public static void main(String[] args){
        BigInteger a = new BigInteger("9999999999999999");
        BigInteger b = new BigInteger("8888888888888888");
        System.out.println("bits in a = " + a.bitLength());
        System.out.println("bits in b = " + b.bitLength());
        System.out.println("a + b = " + a.add(b));
        System.out.println("a & b = " + a.and(b));
```

```

System.out.println("a & ~b = " + a.andNot(b));
System.out.println("a / b = " + a.divide(b));
BigInteger[] r = a.divideAndRemainder(b);
System.out.println("a / b: q = " + r[0] + ", r = " + r[1]);
System.out.println("gcd(a, b) = " + a.gcd(b));
System.out.println("max(a, b) = " + a.max(b));
System.out.println("min(a, b) = " + a.min(b));
System.out.println("a mod b = " + a.mod(b));
System.out.println("I/a mod b = " + a.modInverse(b));
System.out.println("a^n mod b = " + a.modPow(a, b));
System.out.println("a * b = " + a.multiply(b));
System.out.println("-a = " + a.negate());
System.out.println ("~a = " + a.not());
System.out.println("a | b = " + a.or(b));
System.out.println("a ^ 3 = " + a.pow(3));
System.out.println("a % b = " + a.remainder(b));
System.out.println("a << 3 = " + a.shiftLeft(3));
System.out.println("a >> 3 = " + a.shiftRight(3));
System.out.println("sign(a) = " + a.signum());
System.out.println("a - b = " + a.subtract(b));
System.out.println("a ^ b = " + a.xor(b));
}
}

```



```

C:\ Командная строка
C:\>jdk1.3\bin\javac BigIntegerTest.java
C:\>jdk1.3\bin\java BigIntegerTest
bits in a = 57
bits in b = 67
a + b = 88988888888888888887
a & b = 72133975954525752
a & ~b = 27866024045474247
a / b = 0
a / b: q = 0, r = 99999999999999999999
gcd(a, b) = 1
max(a, b) = 88888888888888888888
min(a, b) = 999999999999999999
a mod b = 9999999999999999
I/a mod b = 87543098654209765319
p^n mod b = 36948590717373350119
a * b = 888888888888888799911111111111111112
-a = -9999999999999999
~a = -10000000000000000000
a | b = 88916754912934363135
p ^ 3 = 9999999999999997000000000000000002999999999999999999
a << 3 = 9999999999999999
a >> 3 = 7999999999999992
a << 3 = 12499999999999999
sign(a) = 1
a - b = -8878888888888888889
p ^ b = 88844620936979837383
C:\>

```

Рис. 4.4

Зверніть увагу на те, що в програму лістингу 4.3 треба імпортувати пакет [java.math](#). Передивіться файл [BigInteger](#) у папці [java.math](#) і визначте, чи всі методи цього класу були задіяні у попередній програмі.

4.6. Клас Big Decimal

Клас [BigDecimal](#) розташований в пакеті [java.math](#).

Кожний об'єкт цього класу зберігає два цілочисельні значення: мантису дійсного числа у вигляді об'єкту класу [BigInteger](#), і невід'ємний десятковий порядок числа типу [int](#).

Наприклад, для числа 76.34862 буде зберігатися мантиса 7 634 862 в об'єкті класу [BigInteger](#), і порядок 5 як ціле число типу [int](#). Таким чином, мантиса може містити будь-яку кількість цифр, а порядок обмежений значенням константи [integer.MAX_VALUE](#). Результат операції над об'ємами класу [BigDecimal](#) округляється по одному із восьми правил, визначеними наступними статичними цілими константами:

- [ROUND_CEILING](#) — округлення в сторону більшого цілого;
- [ROUND_DOWN](#) — округлення до нуля, до меншого по модулю цілого значення;
- [ROUND_FLOOR](#) — округлення до меншого цілого;
- [ROUND_HALF_DOWN](#) — округлення до найближчого цілого, середнє значення округляється до меншого цілого;
- [ROUND_HALF_EVEN](#) — округлення до найближчого цілого, середнє значення округляється до парного числа;
- [ROUND_HALF_UP](#) — округлення до найближчого цілого, середнє значення округляється до більшого цілого;
- [ROUND_UNNECESSARY](#) — вважається, що результат буде цілим, і округлення не знадобиться;
- [ROUND_UP](#) — округлення від нуля, до більшого по модулю цілому значенню.

В класі [BigDecimal](#) чотири конструктори:

- [BigDecimal \(BigInteger bi\)](#) — об'єкт буде зберігати велике ціле [b](#) і порядок рівний нулю;
- [BigDecimal \(BigInteger mantissa, int scale\)](#) — задається мантиса [mantissa](#) и невід'ємний порядок [scale](#) об'єкту; якщо порядок [scale](#) відємний, виникає виключна ситуація;
- [BigDecimal \(double d\)](#) — об'єкт буде містити дійсне число подвійної точності [d](#); якщо значення [d](#) нескінчено або [NaN](#), то виникає виключна ситуація;
- [BigDecimal \(String val\)](#) — число задається рядком символів [val](#), який повинен містити запис числа по правилах мови [Java](#).

При використанні третього із перерахованих конструкторів виникає неприємна особливість, відмічена в документації. Оскільки дійсне число при переведі в двійкову форму представляється, як правило, нескінченим двійковим дробом, то при створенні об'єкту, наприклад, [BigDecimal\(0.1\)](#), мантиса, що зберігається в об'єкті, буде дуже великою. Вона показана на рис. 4.5. Але при створенні такого ж об'єкту четвертим конструктором, [BigDecimal \("0.1"\)](#), мантиса буде рівна просто 1.

В класі перевизначені методи [doubleValue\(\)](#), [floatValue\(\)](#), [intValue\(\)](#), [longValue\(\)](#).

Більшість методів цього класу моделюють операції з дійсними числами. Вони повертають об'єкт класу [BigDecimal](#). Тут буква [x](#) означає об'єкт класу [BigDecimal](#), буква [n](#) - ціле значення типу [int](#), буква [r](#) - спосіб округлення, одну із восьми перечислених вище констант:

- [abs\(\)](#) — абсолютне значення об'єкту [this](#);
- [add\(x\)](#) — операція [this + x](#);
- [divide\(x, r\)](#) — операція [this / x](#) з округленням по способу [r](#);
- [divide\(x, n, r\)](#) — операція [this / x](#) із зміною порядку і округленням по способу [r](#);
- [max\(x\)](#) — найбільше із [this](#) і [x](#);
- [min\(x\)](#) — найменше із [this](#) і [x](#);
- [movePointLeft\(n\)](#) — зсув вліво на [n](#) розрядів;
- [movePointRight\(n\)](#) — зсув вправо на [n](#) розрядів;

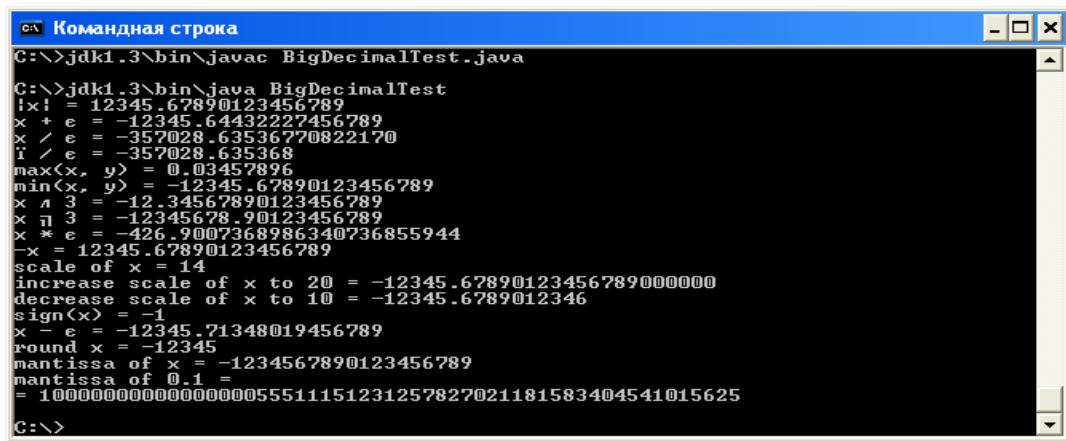
multiply(x) — операція **this * x**;
negate() — повертає об'єкт з протилежним знаком;
scale() — повертає порядок числа;
setScale(n) — установлює новий порядок **n** ;
setScale(n, r) — установлює новий порядок **n** і округляє число при необхідності по способу **r**;
signum() — знак числа, котре зберігається в об'єкті;
subtract(x) — операція **this - x**;
toBigInteger() — округлення числа, котре зберігається в об'єкті;
unscaledValue() — повертає мантису числа.

Лістинг 4.4. Методи класу **BigDecimal** в програмі **BigDecimalTest**

```

import java.math.*;
class BigDecimalTest{
  public static void main( String [] args) {
    BigDecimal x = new BigDecimal("-12345.67890123456789");
    BigDecimal y = new BigDecimal("345.7896e-4");
    BigDecimal z = new BigDecimal(new BigInteger("123456789"),8);
    System.out.println("x = " + x.abs());
    System.out.println("x + y = " + x.add(y));
    System.out.println("x / y = " + x.divide(y, BigDecimal.ROUND_DOWN));
    System.out.println("x / y = " + x.divide(y, 6, BigDecimal.ROUND_HALF_EVEN));
    System.out.println("max(x, y) = " + x.max(y));
    System.out.println("min(x, y) = " + x.min(y));
    System.out.println("x << 3 = " + x.movePointLeft(3));
    System.out.println("x >> 3 = " + x.movePointRight(3));
    System.out.println("x * y = " + x.multiply(y));
    System.out.println("-x = " + x.negate());
    System.out.println("scale of x = " + x.scale());
    System.out.println("increase scale of x to 20 = " + x.setScale(20));
    System.out.println("decrease scale of x to 10 = " +
    x.setScale (10, BigDecimal.ROUND_HALF_UP));
    System.out.println("sign(x) = " + x.signum());
    System.out.println("x - y = " + x.subtract(y));
    System.out.println("round x = " + x.toBigInteger());
    System.out.println("mantissa of x = " + x.unscaledValue());
    System.out.println("mantissa of 0.1 =\n= " +
    new BigDecimal(0.1).unscaledValue());
  }
}
  
```

Передивіться файл **BigDecimal** у папці **java.math** і визначте, чи всі методи цього класу були задіяні у попередній програмі.



```

Командная строка
C:\>jdk1.3\bin\javac BigDecimalTest.java
C:\>jdk1.3\bin\java BigDecimalTest
x = 12345.67890123456789
x + e = -12345.64432227456789
x / e = -357028.63536770822170
x / e = -357028.635368
max(x, y) = 0.03457896
min(x, y) = -12345.67890123456789
x << 3 = -12.34567890123456789
x << 3 = -12345678.90123456789
x * e = -426.9007368986340736855944
-x = 12345.67890123456789
scale of x = 14
increase scale of x to 20 = -12345.67890123456789000000
decrease scale of x to 10 = -12345.6789012346
sign(x) = -1
x - e = -12345.71348019456789
round x = -12345
mantissa of x = -1234567890123456789
mantissa of 0.1 =
= 100000000000000000000055511151231257827021181583404541015625
  
```

Рис. 4.5
 Приведено ще один приклад. Напишемо простенький калькулятор, який виконує

четири арифметичні дії з числами довільної величини. Він працює з командного рядка. Програма

представлена в лістингі 4.5, а приклади використання калькулятора — на рис. 4.6.

Лістинг 4.5. Найпростіший калькулятор

```
import java.math.*;
class Calc{
    public static void main(String[] args){
        if (args.length < 3){
            System.out.println("Usage: Java Calc operand operator operand");
            return;
        }
        BigDecimal a = new BigDecimal(args[0]);
        BigDecimal b = new BigDecimal(args[2]);
        switch (args[1].charAt(0)){
            case '+': System.out.println(a.add(b)); break;
            case '-': System.out.println(a.subtract(b)); break;
            case '*': System.out.println(a.multiply(b)); break;
            case '/': System.out.println(a.divide(b,
                BigDecimal.ROUND_HALF_EVEN)); break;
            default : System.out.println("Invalid operator");
        }
    }
}
```

```
Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\User>cd C:\
C:\>jdk1.3\bin\javac Calc.java
C:\>jdk1.3\bin\java Calc 235 + 138
373
C:\>jdk1.3\bin\java Calc 235 - 138
97
C:\>jdk1.3\bin\java Calc 235 "*" 138
32430
C:\>jdk1.3\bin\java Calc 235 / 138
2
C:\>_
```

Рис. 4.6

Чому символ множення - зірочка — заключений на рис. 4.6 в лапки? "Юніксоїдам" це зрозуміло, а для других дамо коротке пояснення. Це особливість операційної системи, а не мови **Java**. Введений з клавіатури рядок спочатку споглядається командна оболонка (**shell**) операційної системи, а зірочка для неї — вказівка підставити на це місце всі імена файлів із поточного каталогу. Оболонка зробить це, і інтерпретатор **Java** одержить від неї довгий рядок, в якому замість зірочки стоять імена файлів через пробіл. Зірочка в лапках розуміється командною оболонкою як звичайний символ. Командна оболонка знімає лапки і передає інтерпретатору **Java** те, що потрібно.

4.7. Клас Class

Клас **Object**, котрий стоїть на чолі ієрархії класів **Java**, представляє всі об'єкти, що діють в системі, являється їх спільною оболонкою. Всякий об'єкт можна вважати екземпляром класу **Object**.

Клас з іменем **Class** представляє характеристики класу, екземпляром якого являється об'єкт. Він зберігає інформацію про те, чи не являється об'єкт на самім ділі інтерфейсом, масивом або примітивним типом, який суперклас об'єкту, яке ім'я класу, які в ньому конструктори, поля, методи і вкладені класи.

В класі **Class** немає конструкторів, екземпляри цього класу створюються виконуючою системою **Java** під час завантаження класу і представляється методом **getClass()** класу **Object**, наприклад:

```
String s = "Це рядок";
Class c = s.getClass();
```

Статичний метод **forName(String class)** повертає об'єкт класу **Class** для класу, указаного в аргументі, наприклад:

```
Class cl = Class.forName("Java.lang.String");
```

Але цей спосіб створення об'єкту класу **Class** вважається застарілим (**deprecated**). В нових версіях **JDK** для цієї мети використовується спеціальна конструкція — до імені класу через точку додається слово **class**:

```
Class c2 = Java.lang.String.class;
```

Логічні методи **isArray()**, **isInterface()**, **isPrimitive()** дозволяють уточнити, чи не являється об'єкт масивом, інтерфейсом або примітивним типом.

Якщо об'єкт посилочного типу, то можна одержати дані про вкладені класи, конструктори, методи і поля методами **getDeclaredClasses()**, **getDeclaredConstructors()**, **getDeclaredMethods()**, **getDeclaredFields()**, у вигляді масиву класів, відповідно **Class**, **Constructor**, **Method**, **Field**. Останні три класи розташовані в пакеті **java.lang.reflect** і містять дані про конструктори, поля і методи аналогічно тому, як клас **Class** зберігає дані про класи.

Методи **getClasses()**, **getConstructors()**, **getInterfaces()**, **getMethods()**, **getFields()** повертають такі ж масиви, але не всіх, а тільки відкритих членів класу.

Метод **getSuperClass()** повертає суперклас об'єкту посилочного типу, **getPackage()** - пакет, **getModifiers()** - модифікатори класу в бітовій формі. Модифікатори можна потім розшифрувати методами класу **Modifier** із пакету **java.lang.reflect**.

Лістинг 4.6 показує застосування цих методів, а рис. 4.7 — виведення результатів

Лістинг 4.6. Методи класу **Class** в програмі **ClassTest**

```
import java.lang.reflect.*;
class ClassTest{
    public static void main(String[] args){
        Class c = null, c1 = null, c2 = null;
        Field[] fld = null;
        String s = "Some string";
        c = s.getClass();
        try{
            c1 = Class.forName("java.lang.String"); // Старий стиль
            c2 = java.lang.String.class; // Новий стиль
        }
```

```

if (!c1.isPrimitive())
fld = c1.getDeclaredFields(); // Всі поля класу String
} catch (Exception e) {}
System.out.println("Class c: " + c);
System.out.println("Class c1: " + c1);
System.out.println("Class c2: " + c2);
System.out.println("Superclass c: " + c.getSuperclass());
System.out.println("Package c: " + c.getPackage());
System.out.println("Modifiers c: " + c.getModifiers());
for (int i = 0; i < fld.length; i++)
System.out.println(fld[i]);
}
}
}

```

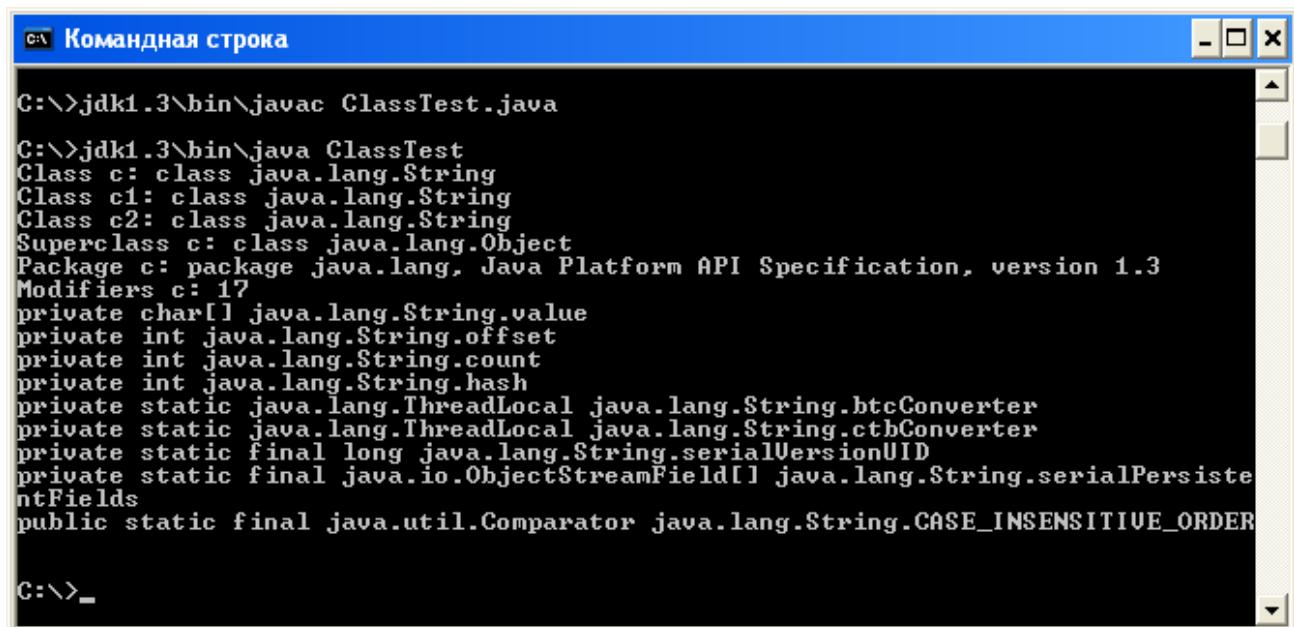


Рис. 4.7

Методи, які повертають властивості класів, спричиняють виключні ситуації, які вимагають обробки. Тому в програму введено блок `try{} catch()`. Розгляд обробки виключчих ситуацій ми відкладаємо до уроку 16.

Передивіться файл `Class` у папці `java.lang` і визначте, чи всі методи цього класу були задіяні у попередній програмі.

Лабораторна робота 3. Використання числових класів

Доповнити програму `Calculator` наступними функціями: обчислення тригонометричних функцій, включаючи `arc tg x`, показникової і степеневу, включаючи квадратний і кубічний корені, логарифми натуральний і десятковий.

Програмуванн в Java

УРОК 5. Робота з рядками

- Клас **String**
- Як створити рядок
- Зєднання рядків
- Маніпуляції з рядками
- Як узнати довжину рядка
- Як вибрати символи із рядка
- Як вибрати частину рядка (підрядок)
- Як порівняти рядки
- Як знайти символ в рядку
- Як знайти підрядок
- Як змінити регістр літер
- Як замінити окремий символ
- Як видалити пробіли на початку і в кінці рядка
- Як перетворити дані іншого типу в рядок
- Клас **StringBuffer**
- Конструктори
- Як додати підрядок
- Як вставити підрядок
- Як видалити підрядок
- Як видалити символ
- Як замінити підрядок
- Як перевернути рядок
- Синтаксичний разбір рядка
- Клас **StringTokenizer**
- Заключення

Дуже важливе місце в обробці інформації займає робота з текстами. Як і багато чого, текстові рядки в мові **Java** являються об'єктами. Вони представляються екземплярами класу **String** або класу **StringBuffer**. Спочатку це незвично і здається занадто громіздким, але, призвичаївшись, ви оціните зручність роботи з класами, а не з масивами символів. Звичайно, можливо занести текст в масив символів типу **char** або навіть в масив байтів типу **byte**, але тоді ви не зможете використати готові методи роботи з текстовими рядками.

Для чого в мову введені два класи для зберігання рядків? В об'єктах класу **String** зберігаються рядки-константи незмінної довжини і змісту, так би мовити, відлиті в бронзі. Це значно прискорює обробку рядків і дозволяє економити пам'ять, розділяючи рядок між об'єктами, які його використовують. Довжину рядків, що зберігаються в об'єктах класу **StringBuffer**, можна змінювати, вставляючи і додаючи рядки і символи, видаляючи підрядки або зчіплючи декілька рядків у один рядок. В багатьох випадках, коли треба змінити довжину рядка типу **String**, компілятор **Java** неявно перетворює його до типу **StringBuffer**, змінює довжину, потім перетворює назад в тип **String**. Наприклад, наступна дія

```
String s = "Це" + " один " + "рядок";
```

компілятор виконує так:

```
String s = new StringBuffer().append("Це").append(" один ").append("рядок").toString();
```

Буде створений об'єкт класу **StringBuffer**, в нього послідовно додані рядки "Це", " один ", "рядок", і одержаний об'єкт класу **StringBuffer** буде приведений до типу **String** методом **toString ()**. Нагадаємо, що символи в рядках зберігаються в кодировці **Unicode**, в якій кожен символ займає два байти. Тип кожного символу **char**.

5.1. Клас String

Перед роботою з рядком його треба створити. Це можна зробити різними способами.

5.1.1. Як створити рядок.

Найпростіший спосіб створити рядок — це організувати посилання типу `String` на рядок-константу:

```
String s1= "Це рядок.;"
```

Якщо константа довга, можна записати її в декількох рядках текстового редактору, зв'язуючи їх операцією зчеплення:

```
String s2 = "Це довгий рядок, " + "записаний в двох рядках вихідного тексту";
```

Зauważення

Не забувайте різницю між пустим рядком `String s = ""`, який не містить жодного символу, і пустою посилкою `String s = null`, котра не вказує ні на один рядок і не являється об'єктом.

Самий правильний спосіб створити об'єкт з точки зору ООП — це викликати його конструктор в операції `new`. Клас `String` надає у ваше розпорядження дев'ять конструкторів:

- `String()` — створюється об'єкт з пустим рядком;
- `String (String str)` — із одного об'єкта створюється інший, тому цей конструктор використовується рідко;
- `String (StringBuffer str)` — перетворена копія об'єкту класу `BufferString`;
- `String(byte[] byteArray)` — об'єкт створюється із масиву байтів `byteArray`;
- `String (char [] charArray)` — об'єкт створюється із масиву `charArray` символів `Unicode`;
- `String (byte [] byteArray, int offset, int count)` — об'єкт створюється із частини масиву байтів `byteArray`, починається з індексу `offset` і містить `count` байтів;
- `String (char [] charArray, int offset, int count)` — те ж саме, але масив складається із символів `Unicode`;
- `String(byte[] byteArray, String encoding)` — символи, записані в масиві байтів, задаються в `Unicode`-рядку з врахуванням кодування `encoding`;
- `String(byte[] byteArray, int offset, int count, String encoding)` — те ж саме, але тільки для частини масиву.

При неправильному заданні індексів `offset`, `count` або кодування `encoding` виникає виключна ситуація.

Конструктори, що використовують масив байтів `byteArray`, призначені для створення `Unicode`-рядка із масива байтових `ASCII`-кодувань символів. Така ситуація виникає при читанні `ASCII`-файлів, одержанні інформації із бази даних або при передачі інформації по мережі. В самому простому випадку компілятор для отримання двобайтових символів `Unicode` додасть до кожного байту старший нульовий байт. Получиться діапазон '`\u0000`' — '`\u00ff`' кодування `Unicode`, відповідний кодам `Latin 1`. Тексти на кирилиці будуть виведені неправильно.

Якщо ж на комп'ютері зроблені місцеві установки, як говорять на жаргоні "установлена локаль" (`locale`) (в `MS Windows` це виконується утилітою `Regional Options` у вікні `Control Panel`), то компілятор, прочитавши ці установки, створить символи `Unicode`, що відповідають місцевому кодуванню сторінки. В русифікованому варіанті `MS Windows`, яким ми в більшості випадків і користуємося, це звичайно кодова сторінка `CP1251`.

Якщо вихідний масив з кириличним `ASCII`-текстом був у кодуванні `CP1251`, то рядок `Java` буде

створений правильно. Кирилиця попаде в свій діапазон "\u0400"—"\u04FF" кодування [Unicode](#). Але у кирилиці є ще, по меншій мірі, чотири кодування.

- В [MS-DOS](#) застосовується кодування [CP866](#).
- В [UNIX](#) звичайно застосовується кодування [KOI8-R](#).
- На комп'ютерах [Apple Macintosh](#) використовується кодування [MacCyrillic](#).
- Єсть ще і міжнародне кодування кирилиці [ISO8859-5](#);

Наприклад, байт [11100011](#) (0xE3 в шістнадцятеричній системі) в кодуванні [CP1251](#) представляє кириличну літеру Г, в кодуванні [CP866](#) — літеру У, в кодуванні [KOI8-R](#) — літеру Ц, в [ISO8859-5](#) — літеру у, в [MacCyrillic](#) — літеру г.

Якщо вихідний кириличний [ASCII](#)-текст був в одному із цих кодувань, а місцеве кодування [CP1251](#), то [Unicode](#)-символи рядка [Java](#) не будуть відповідати кирилиці. В таких випадках використовуються останні два конструктори, в яких параметром [encoding](#) указується, яку кодову таблицю використовувати конструктору при створенні рядка.

Питання кирилізації ми ще будемо обговорювати в уроках 9 і 18, а поки що замітьте, що при створенні рядка із масиву байтів краще вказати те ж саме кириличне кодування, в якому записаний масив. Тоді ви одежите рядок [Java](#) з правильними символами [Unicode](#).

При виведенні ж рядка на консоль, у вікно, у файл або при передачі по мережі краще перетворити рядок [Java](#) з символами [Unicode](#) по правилам виведення в потрібне місце.

Ще один спосіб створити рядок — це використати два статичні методи

`copyValueOf(char[] charArray) і copyValueOf(char[] charArray, int offset, int length).`

Вони створюють рядок по заданому масиву символів і повертають його в якості результату своєї роботи. Наприклад, після виконання наступного фрагменту програми

```
char[] c = {'C', 'и', 'м', 'в', 'о', 'л', 'ъ', 'н', 'и', 'й'};
String s1 = String.copyValueOf(c);
String s2 = String.copyValueOf(c, 3, 7);
```

одержимо в обєкті [s1](#) рядок "Символьний", а в обєкті [s2](#) - рядок "вольний".

5.1.2. Зчеплення рядків

З рядками можна проводити операцію зчеплення рядків (concatenation), що позначається знаком плюс +. Ця операція створює новий рядок, просто складений із першого і другого рядка, як показано на початку даного уроку. Її можна застосувати і до констант, і до змінних. Наприклад:

```
String attention = "Увага: ";
String s = attention + "невідомий символ";
```

Друга операція - присвоювання += застосовується до змінних в лівій частині:

```
attention += s;
```

Оскільки операція + перевантажена з додавання чисел на зчеплення рядків, постає питання про

пріоритет цих операцій. У зчепленні рядків пріоритет вище, ніж у додавання, тому, записавши "2" + 2 + 2, одержимо рядок "222". Але, записавши 2 + 2 + "2", одержимо рядок "42", оскільки дії виконуються зліва направо. Якщо ж запишемо "2" + (2 + 2), то одержимо "24".

5.1.3. Маніпуляції рядками

В класі String є багато методів для роботи з рядками. Подивимось, що вони дозволяють робити.

5.1.3.1. Як узнати довжину рядка

Для того щоб узнати довжину рядка, тобто кількість символів в ній, треба звернутися до методу length():

```
String s = "Write once, run anywhere.";
int len = s.length();
```

або ще простіше

```
int len = "Write once, run anywhere.".length();
```

оскільки рядок-константа — повноцінний об'єкт класу String. Замітьте, що рядок — це не масив, у нього немає поля length.

Уважний читач, що вивчив рис. 4.7, готовий зі мною не погодитися. Ну, що ж, дійсно, символи зберігаються в масиві, але він закритий, як і всі поля класу String.

5.1.3.2. Як вибрати символи із рядка

Вибрати символ з індексом ind (індекс першого символу рівний нулю) можна методом charAt(int ind). Якщо індекс ind відємний або не менший за довжину рядка, виникає виключна ситуація. Наприклад, після ізначення

```
char ch = s.charAt(3);
```

змінна ch буде мати значення 't'

Всі символи рядка у вигляді масиву символів можна отримати методом

toCharArray(), що повертає масив символів.

Якщо ж потрібно включити в масив символів dst, починаючи з індексу ind масиву підрядок від індексу begin включно до індексу end виключно, то користуйтеся методом getChars(int begin, int end, char[] dst, int ind) типу void. В масив буде записано end - begin символів, які займуть елементи масиву, починаючи з індексу ind до індексу ind + (end - begin) - 1. Цей метод створює виключну ситуацію в наступних випадках:

- посилка dst = null;
- індекс begin відємний;
- індекс begin більше індексу end;
- індекс end більше довжини рядка;
- індекс ind відємний;
- ind + (end - begin) > dst.length.

Наприклад, після виконання

```
char[] ch = {'К', 'о', 'р', 'о', 'л', 'ъ', ' ', 'л', 'е', 'т', 'а'};
```

```
"Пароль легко найти".getChars(2, 8, ch, 2);
```

результат буде такий:

```
ch = {'К', 'о', 'р', 'о', 'л', 'ъ', ' ', 'л', 'е', 'т', 'а'};
```

Якщо треба одержати масив байтів, що містить всеі символи рядка в байтовому кодуванні **ASCII**, то використовуйте метод `getBytes()`. Цей метод при переведі символів із **Unicode** в **ASCII** використовує локальну кодову таблицю. Якщо ж треба одержати масив байтів не в локальному кодуванні, а в якомусь іншому, використовуйте метод `getBytes(String encoding)`.

5.1.3.4. Як вибрати підрядок

Метод `substring(int begin, int end)` виділяє підрядок від символу з індексом `begin` включно до символу з індексом `end` виключно. Довжина підрядка буде рівною `end - begin`.

Метод `substring (int begin)` виділяє підрядок від індекса `begin` включно до кінця рядка.

Якщо індекси відємні, індекс `end` більше довжини рядка або `begin` більше ніж `end`, то виникає виключна ситуація. Наприклад, після виконання

```
String s = "Write once, run anywhere.;"
```

```
String sub1 = s.substring(6, 10);
```

```
String sub2 = s.substring(16);
```

одержимо в рядку `sub1` значення "once", а в `sub2` - значення "anywhere".

5.1.3.4. Як порівняти рядки

Операція порівняння `==` співставляє лише посилання на рядки. Вона виясняє, чи вказують посилання на один і той же рядок. Наприклад, для рядків

```
String s1 = "Якийсь рядок";
```

```
String s2 = "Інший рядок";
```

порівняння `s1 == s2` дає в результаті `false`.

Значення `true` одержимо лише у випадку, коли обидві посилки вказують на один і той же рядок, наприклад, після присвоєння `s1 = s2`.

Логічний метод `equals (object obj)`, перевизначений із класу **Object**, повертає `true`, якщо аргумент `obj` не ріній `null`, являється об'єктом класу **String**, і рядок, що міститься в ньому, повністю ідентичний даному рядку аж до співпадання реєстру літер. В решті випадків повертається значення `false`.

Логічний метод `equalsIgnoreCase(object obj)` працює так же, але однакові літери, записані в різних реєстрах, вважаються співпадаючими.

Наприклад, `s2.equals("інший рядок")` дасть в результаті `false`, а `s2.equalsIgnoreCase("інший рядок")` поверне `true`.

Метод `compareTo(string str)` повертає ціле число типу `int`, обчислене за наступними правилами:

1. Порівнюються символи даного рядка `this` і рядка `str` з однаковими індексами, доки не зустрінуться різні символи з індексом, допустимо `k`, або доки один з рядків не закінчиться.
2. В першому випадку повертається значення `this.charAt(k) - str.charAt(k)`, тобто різниця кодів Unicode перших неспівпадаючих символів.
3. В другому випадку повертається значення `this.length() - str.length()`, тобто різниця довжин рядків.
4. Якщо рядки співпадають, повертається `0`.

Якщо значення `str` рівно `null`, виникає виключна ситуація. Нуль повертається в тій же ситуації, в якій метод `equals()` повертає `true`.

Метод `compareTolgnoreCase(string str)` робить порівняння без врахування регістру літер, точніше кажучи, виконується метод

`this.toUpperCase().toLowerCase().compareTo`

`str.toUpperCase().toLowerCase());`

Ще один метод - `compareTo(Object obj)` створює виключну ситуацію, якщо `obj` не являється рядком. В решті випадків він працює як метод `compareTo(String str)`. Ці методи не враховують алфавітний порядок символів в локальному кодуванні.

Порівняти підрядок даного рядка `this` з підрядком тієї ж довжини `len` іншого рядка `str` можна логічним методом

`regionMatches(int ind1, String str, int ind2, int len)`

Тут `ind1` — індекс початку підрядка даного рядка `this`, `ind2` - індекс початку підрядка іншого рядка `str`. Результат `false` отримаємо в наступних випадках:

- хоч би один із індексів `ind1` або `ind2` відємний;
- хоч би одно із `ind1 + len` або `ind2 + len` більше довжини відповідного рядка;
- хоч би одна пара символів не співпадає.

Цей метод розрізняє символи, записані в різних регістрах. Якщо треба порівняти підрядки без врахування регістрів літер, то використовуйте логічний метод:

`regionMatches(boolean flag, int ind1, String str, int ind2, int len)`

Якщо перший параметр `flag` рівний `true`, то регістр літер при порівнянні підрядків не враховується, якщо `false` - враховується.

5.1.3.5. Як знайти символ в рядку

Пошук завжди ведеться з врахуванням регістру літер. Першу поява символу `ch` в даному рядку `this` можна виявити методом `indexOf(int ch)`, що повертає індекс цього символу в рядку або `-1`, якщо символу `ch` в рядку `this` немає.

Наприклад, `"Молоко".indexOf('o')` видасть в результаті `1`.

Другу і наступні появи символу `ch` в даному рядку `this` можна відслідкувати методом `indexOf(int ch, int ind)`. Цей метод починє пошук символу `ch` з індекса `ind`. Якщо `ind < 0`, то пошук іде з початку рядка, якщо `ind` більше за довжину рядка, то символ не шукається, тобто повертається `-1`.

Наприклад, "молоко".indexof('o', indexof ('o') + 1) дасть в результаті 3.

Остання поява символу `ch` в даному рядку `this` відслідковується методом `lastIndexOf (int ch)`. Він проглядає рядок в зворотному порядку. Якщо символ `ch` не знайдено, повертається -1.

Наприклад, "Молоко".lastIndexOf('o') дасть в результаті 5.

Передостанню і попередню появу символу `ch` в даному рядку `this` можна відслідкувати методом `lastIndexOf(int ch, int ind)`, якийй проглядає рядок у зворотному порядку, починаючи з індексу `ind`. Якщо `ind` більше за довжину рядка, то пошук іде від кінця рядка, якщо `ind < 0`, то повертається -1.

5.1.3.6. Як знайти підрядок

Пошук завжди ведеться з врахуванням регістру літер. Перше входження підрядка `sub` в даний рядок `this` відшукується методом `indexOf (String sub)`. Він повертає індекс першого символу першого входження підрядка `sub` в рядок або -1, якщо підрядок `sub` не входить в рядок `this`. Наприклад, " Молоко".indexof ("ок") дасть в результаті 3.

Якщо ви хочете почати пошук не з початку рядка, а з якогось індексу `ind`, використовуйте метод `indexOf (String sub, int ind)`. Якщо `ind < 0`, то пошук іде з початку рядка, якщо ж `ind` більше довжини рядка, то символ не шукається, тобто повертається -1.

Останнє входження підрядка `sub` в даний рядок `this` можна відшукати методом `lastIndexOf (string sub)`, що повертає індекс першого символу останнього входження підрядка `sub` в рядок `this` або -1, якщо підрядок `sub` не входить в рядок `this`.

Останнє входження підрядка `sub` не у весь рядок `this`, а тільки в його початок до індексу `ind` можна відшукати методом `lastIndexOf(String stf, int ind)`. Якщо `ind` більше довжини рядка, то пошук іде від кінця рядка, якщо `ind < 0`, то повертається -1.

Для того щоб перевірити, чи не починається даний рядок `this` з підрядка `sub`, використовуйте логічний метод `startsWith(string sub)`, який повертає `true`, якщо даний рядок `this` починається з підрядка `sub`, або співпадає з ним, або підрядок `sub` пустий.

Можна перевірити і появу підрядка `sub` в даному рядку `this`, починаючи з деякого індексу `ind` логічним методом `startsWith(String sub),int ind)`. Якщо індекс `ind` відємний або більший за довжину рядка, повертається `false`.

Для того щоб перевірити, чи не закінчується даний рядок `this` підрядком `sub`, використовуйте логічний метод `endsWith(String sub)`. Врахуйте, що він повертає `true`, якщо підрядок `sub` співпадає з усім рядком або підрядок `sub` пустий.

Наприклад, `if (fileName.endsWith(". Java"))` відслідкує імена файлів з вихідними текстами `Java`.

Перечислені вище методи створюють виключну ситуацію, якщо

`sub == null`.

Якщо ви хочете здійснити пошук, який не враховує регістр літер, поміняйте перед цим регістр всіх символів рядка.

5.1.3.7. Як змінити регістр літер

Метод `toLowerCase ()` повертає новий рядок, в якому всі літери переведеі в нижній регістр, тобто зроблені прописними.

Метод `toUpperCase ()` повертає новий рядок, в якому всі літери переведеі у верхній регістр, тобто зроблені заглавними. При цьому використовується локальна кодова таблиця по замовчуванню. Якщо потрібна інша локаль, то застосовуються методи `toLowerCase(Locale loc)` і `toUpperCase(Locale loc)`.

5.1.3.8. Як замінити окремий символ

Метод `replace (int old, int new)` повертає новий рядок, в якому всі входження символу `old` замінені символом `new`. Якщо символа `old` в рядку немає, то повертається посилка на вихідний рядок. Наприклад, після виконання "Рука в руку сует хлеб", `replace ('у', 'е')` одержимо рядок "Река в реке сеет хлеб".

Регістр літер при заміні враховується.

5.1.3.9. Як видалити пробіли на початку і в кінці рядка

Метод `trim()` повертає новий рядок, в якому видалені початкові і кінцеві символи з кодами, що не превищують `\u0020`.

5.1.3.10. Як перетворити дані іншого типу в рядок

В мові Java прийнято, що кожний клас відповідає за перетворення інших типів в тип цього класу і повинен містити потрібні для цього методи. Клас `String` містить вісім статичних методів `valueOf (type elem)` перетворення в рядок примітивних типів `boolean, char, int, long, float, double`, масиву `char[]`, і просто обєкту типу `Object`. Дев'ятий метод `valueOf(char[] ch, int offset, int len)` перетворює в рядок підмасив масиву `ch`, який починається з індексу `offset` і має `len` елементів.

Крім того, в кожному класі є метод `toString ()`, перевизначений або просто унаслідуваній від класу `Object`. Він перетворює обєкти класу в рядок. Фактично, метод `valueOf()` викликає метод `toString ()` відповідного класу. Тому результат перетворення залежить від того, як реалізований метод `toString ()`.

Ще один простий спосіб - зчепити значення `elem` якогось типу з пустим рядком: `"" + elem`. При цьому неявно викликається метод `elem. toString ()`.

5.2. Клас StringBuffer

Обєкти класу `StringBuffer` - це рядки змінної довжини. Тільки що створений обєкт має буфер ви значеної ємності (`capacity`), по замовчуванню достатній для зберігання 16 символів. Ємність можна задати в конструкторі обєкта. Як тільки буфер починає переповнюватися, його ємність автоматично збільшується, щоб вмістити нові символи. Також ємність буфера можна збільшити, звернувшись до методу `ensureCapacity(int minCapacity)`. Цей метод змінить ємність тільки якщо `minCapacity` буде більшим за довжину рядка, що зберігається в обєкті. Ємність буде збільшена по наступному правилу. Нехай ємність буфера рівна `N`. Тоді нова ємність буде рівна

`Max(2 * N + 2, minCapacity)`

Таким чином, ємність буфера не можна збільшити менше ніж вдвічі.

Методом `setLength(int newLength)` можна установити довільну довжину рядка. Якщо вона виявиться більше поточної довжини, то додаткові символи будуть рівні `'\u0000'`. Якщо вона буде менше поточної довжини, то рядок буде обрізаний, останні символи втратяться, точніше, будуть замінені символом `'\u0000'`. Ємність при цьому не зміниться. Якщо число `newLength` виявиться від'ємним, виникні виключна ситуація.

Порада

Будьте обережні, установлюючи нову довжину обєкту. Кількість символів у рядку можна узнати, як і для

обєкту класу `String`, методом `length ()`, а ємність — методом `capacity ()`. Створити об'єкт класу `StringBuffer` можна тільки конструкторами.

5.2.1. Конструктори

В класі `StringBuffer` три конструктори:

`StringBuffer ()` - створює пустий об'єкт з ємністю 16 символів;

`StringBuffer.(int capacity)` - створює пустий об'єкт заданої ємності `capacity`;

`StringBuffer (String str)` - створює об'єкт ємністю `str.length() + 16`, що містить рядок `str`.

5.2.2. Як додати підрядок

В класі `StringBuffer` єсть десять методів `append ()`, що додають підрядок в кінець рядка. Вони не створюють новий екземпляр рядка, а повертають посилання на той же самий, але змінений рядок.

Основний метод `append (String str)` приєднує рядок `str` в кінець даного рядка. Якщо посилка `str == null`, то дододається рядок `"null"`.

Шість методів `append (type elem)` додають примітивні типи `boolean, char, int, long, float, double`, перетворені в рядок.

Два методи приєднують до рядка масив `str` і підмасив `sub` символів, перетворені в рядок: `append (char [] str)` і `append (char [], sub, int offset, int len)`.

Десятий метод додає просто об'єкт `append (Object obj)`. Перед цим об'єкт `obj` перетворюється в рядок своїм методом `toString ()`.

5.2.3. Як вставити підрядок

Десять методів `insert ()` призначені для вставки рядка, вказаного параметром методу, в даний рядок. Місце вставки задається першим параметром методу `ind`. Це індекс елементу рядка, перед яким буде зроблена вставка. Він повинен бути невід'ємним і менше довжини рядка, інакше виникне виключна ситуація. Рядок роздвигається, ємність буфера при необхідності збільшується. Методи повертають посилку на той же перетворений рядок.

Основний метод `insert (int ind, String str)` вставляє рядок `str` в даний рядок перед його символом з індексом `ind`. Якщо посилка `str == null` вставляється рядок `"null"`.

Наприклад, після виконання

```
String s = new StringBuffer("Це великий рядок").insert(3, "не").toString();
```

одержимо `s == "Це невеликий рядок"`.

Метод `sb.insert(sb.length o, "xxx")` буде працювати так же, як і метод

```
sb.append("xxx").
```

Шість методів `insert (int ind, type elem)` вставляють примітивні типи `boolean, char, int, long, float, double`, перетворені в рядок. Два методи вставляють масив `str` і підмасив `sub` символів, перетворені в рядок:

```
insert(int ind, char[] str)
```

```
insert(int ind, char[] sub, int offset, int len)
```

Десятий метод вставляє просто об'єкт:

```
insert(int ind, Object obj)
```

Об'єкт `obj` перед додаванням перетворюється в рядок своїм методом `toString()`.

5.2.4. Як видалити підрядок

Метод `delete(int begin, int end)` видаляє із рядка символи, починаючи з індексу `begin` включно до індексу `end` виключно, якщо `end` більше за довжину рядка, то до кінця рядка.

Наприклад, після виконання

```
String s = new StringBuffer("Це невеликий рядок").
```

```
delete(3, 5).toString();
```

одержимо `s == "Це великий рядок"`.

Якщо `begin` відемне, більше за довжину рядка або більше `end`, виникає виключна ситуація.

Якщо `begin == end`, видалення не відбувається.

5.2.5. Як видалити символ

Метод `deleteCharAt (int ind)` видаляє символ з указаним індексом `ind`. Довжина рядка зменшується на одиницю. Якщо індекс `ind` відемний або більший за довжину рядка, виникає виключна ситуація.

5.2.6. Як замінити підрядок

Метод `replace (int begin, int end, String str)` видаляє символи із рядка, починаючи з індексу `begin` включно до індексу `end` виключно, якщо `end` більше за довжину рядка, то до кінця рядка, і вставляє замість них рядок `str`. Якщо `begin` відемне, більше за довжину рядка або більше `end`, виникає виключна ситуація.

5.2.7. Як перевернути рядок

Метод `reverse()` змінює порядок розташування символів у рядку на зворотний порядок.

Наприклад, після виконання

```
StringBuffer s = new StringBuffer("Це невеликий рядок"),
```

```
reverse().toString();
```

одержимо `s == "кодяр йикилевен еЦ"`.

5.2.8. Синтаксичний розбір рядка

Задача розбору введеного тексту (`parsing`) — вічна задача програмування, наряду із сортуванням і

пошуком. Написана маса програм-парсерів ([parser](#)), що розбирають текст по різним ознакам. Але задача залишається. І ось черговий програміст, зневірившись знайти що-небудь підходяще, береться за розробку власної програми розбору.

В пакет [java.util](#) входить простий клас [StringTokenizer](#), що полегшує розбір рядків. Клас [StringTokenizer](#) невеликий, в ньому три конструктори і шість методів.

Перший конструктор [StringTokenizer \(String str\)](#) створює об'єкт, готовий розбити рядок [str](#) на слова, розділені так званими сепараторами - пробілами, символами табуляції '\t', переводу рядка '\n' і повернення каретки '\r'. Сепаратори не включаються в число слів.

Другий конструктор [StringTokenizer \(String str, String delimiters\)](#) задає сепаратори другим параметром [delimiters](#), наприклад:

```
StringTokenizer("Казнить,нельзя:пробелов-нет", " \t\n\r,:-");
```

Тут перший сепаратор - пробіл. Потім ідути символ табуляції, символ переводу рядка, символ повернення каретки, кома, двокрапка, дефіс. Порядок розташування сепараторів у рядку [delimiters](#) не має значення. Сепаратори не включаються в число слів.

Третій конструктор дозволяє включити сепаратори в число слів:

```
StringTokenizer(String str, String delimiters, boolean flag);
```

Якщо параметр [flag](#) рівний [true](#), то сепаратори включаються в число слів, якщо [false](#) — ні. Наприклад:

```
StringTokenizer("a - (b + c) / b * c", " \t\n\r+*-/, true);
```

В разборі рядка на слова активно приймають участь два методи:

метод [nextToken\(\)](#) повертає у вигляді рядка наступне слово;

логічний метод [hasMoreTokens\(\)](#) повертає [true](#), якщо в рядку ще єсть слова, і [false](#), якщо слів більше немає.

Третій метод [countTokens\(\)](#) повертає число залишившихся слів.

Четвертий метод [nextToken\(string newDelimeters\)](#) дозволяє "на ходу" міняти сепаратори. Наступне слово буде виділено по новим сепараторам [newDelimeters](#); нові сепаратори діють далі замість старих, визначених в конструкторі або в попередньому методі [nextToken\(\)](#).

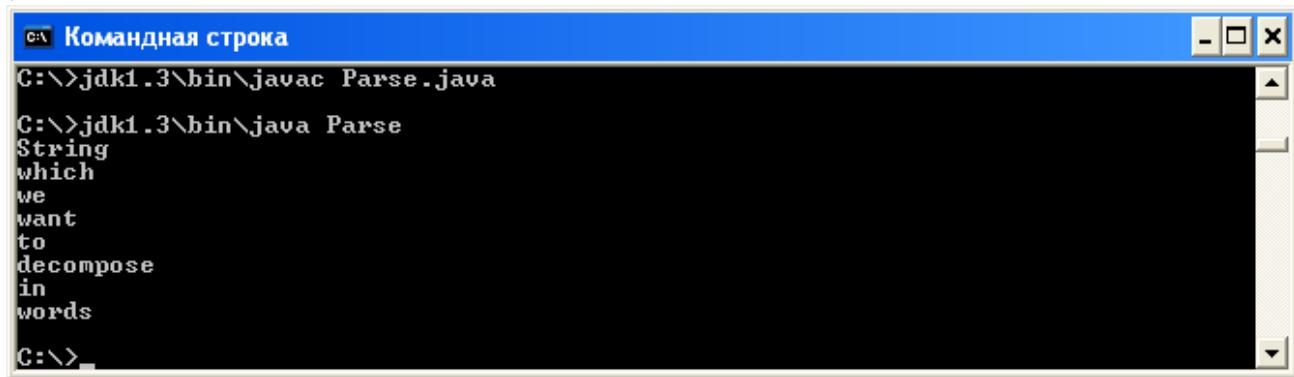
Останні два методи [nextElement\(\)](#) і [hasMoreElements\(\)](#) реалізують інтерфейс [Enumeration](#). Вони просто звертаються до методів [nextToken\(\)](#) і [hasMoreTokens\(\)](#).

Схема дуже проста (лістинг 5.1).

Лістинг 5.1. Розбиття рядка на слова :

```
import java.util.*;
class Parse{
public static void main(String[] args){
String s = "String, which we want to decompose in words ";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,. ");
while(st.hasMoreTokens()) {
```

```
// Одержано слово і що-небудь робимо з ним, наприклад, просто виводимо на екран
System.out.println(st.nextToken());
}
}
```



Одержані слова звичайно заносяться в який-небудь клас-коллекцію: [Vector](#), [Stack](#) або інший, найбільш зручний для подальшої обробки тексту контейнер. Класи-коллекції ми розглянемо в наступному уроці.

Передивітесь файл [StringTokenizer](#) у папці [java.util](#) і визначте, чи всі методи цього класу були задіяні у попередній програмі.

Заключення

Всі методи представлених в цьому уроці класів написані мовою **Java**. Їх вихідні тексти можна подивитися, вони входять у склад **JDK**. Це дуже корисне заняття. Проглянувши вихідний текст, ви отримаєте повну уяву про те, як працює метод. В останніх версіях **JDK** вихідні тексти зберігаються в упакованому архіватором **jar** файлі **src.jar**, що знаходиться в кореневом каталозі **JDK**.

Лабораторна робота 4. Класи String, StringBuffer і StringTokenizer.

Над наведеними нижче завданнями ви повинні працювати дома, а на лабораторній роботі представите налаштовані програми викладачу. При цьому програми 1 – 11 повинен знати кожний і на вибір викладача запустити її на протязі 15 хвилин. Сподіваюся, що програми 12 – 15 виберуть і напишуть кращі студенти.

1. Напишіть програму, яка підраховує число слів у тексті. Розгляньте випадки, коли текст визначається в самій програмі і вводиться з командного рядка при запуску програми на виконання.
 2. Напишіть програму, що записує текст у зворотному порядку. Розгляньте випадки, коли текст визначається в самій програмі і вводиться з командного рядка при запуску програми на виконання.
 3. Напишіть програму, яка в рядку “This is a small string” замінить слово “small” на “very large”.
 4. Напишіть програму, яка в рядку “This is a small string” видалить артикль “a” .
 5. Напишіть програму, яка в рядку “This is a small string” видалить слово “small” .
 6. Напишіть програму, яка в рядок “This is a small string” додасть слово “very” перед “small”.

7. Напишіть програму, яка в кінець рядка “[This is a small string](#)” додасть рядок “[, that we use to illustrate the methods of class StringBuffer](#)” не створюючи нового рядка.
8. Напишіть програму, яка в рядку “[This is a small string](#)” замінить всі літери “[ї](#)” на літеру “[о](#)”.
9. Напишіть програму, яка підраховує число символів у тексті. Розгляньте випадки, коли текст визначається в самій програмі і вводиться з командного рядка при запуску програми на виконання.
10. Напишіть програму, яка перевіряє, чи певне слово міститься в даному тексті. Розгляньте випадки, коли текст визначається в самій програмі і вводиться з командного рядка при запуску програми на виконання. Програма повинна розпізнавати слово незалежно від регістру, в якому воно записане.
11. Напишіть програму, яка перевіряє, чи певна літера міститься в даному тексті. Розгляньте випадки, коли текст визначається в самій програмі і вводиться з командного рядка при запуску програми на виконання. Програма повинна розпізнавати літеру незалежно від регістру, в якому вона записана.
12. Напишіть програму, яка підрахує скільки разів певне слово входить в даний текст незалежно від регістру, в якому воно написане.
13. Напишіть програму, яка підрахує скільки разів певна літера входить в даний текст незалежно від регістру, в якому вона написана.
14. Напишіть програму, яка обчислює відносну частоту появи кожного символу в даному тексті, включаючи знаки пунктуації і пробіли.
15. Напишіть програму, яка обчислює відносну частоту появи певного слова в даному тексті, незалежно від регістру.

Програмування у Java

Урок 6. Класи-колекції

- Клас **Vector**
- Як створити вектор
- Як додати елемент у вектор
- Як замінити елемент
- Як узнати розмір вектора
- Як звернутися до елементу вектора
- Як узнати, чи єсть елемент у векторі
- Як узнати індекс елементу
- Як видалити елементи
- Клас **Stack**
- Клас **Hashtable**
- Як створити таблицю
- Як заповнити таблицю
- Як одержати значення по ключу
- Як узнати наявність ключа або значення
- Як одержати всі елементи таблиці
- Як видалити елементи
- Клас **Properties**
- Інтерфейс **Collection**
- Інтерфейс **List**
- Інтерфейс **Set**
- Інтерфейс **SortedSet**
- Інтерфейс **Map**
- Інтерфейс **Map.Entry**
- Інтерфейс **SortedMap**
- Абстрактні класи-колекції
- Інтерфейс **Iterator**
- Інтерфейс **ListIterator**
- Класи, що створюють списки
- Двонаправлений список
- Класи, що створюють відображення
- Упорядковані відображення
- Порівняння елементів колекцій
- Класи, що створюють множини
- Упорядковані множини
- Дії з колекціями
- Методи класу **Collections**
- Заключення

В лістингі 5.1ми розібрали рядок на слова. Як їх зберегти для подальшої обробки? До сих пір ми користувалися масивами. Вони зручні, якщо треба швидко обробити однотипні елементи, наприклад, просумувати числа, знайти найбільше і найменше значення, посортувати елементи. Але вже для пошуку потрібних даних у великому обємі інформації масиви незручні. Для цього краще використовувати бінарні дерева пошуку. Крім того, масиви завжди мають постійну, наперед задану довжину, в масив незручно додавати елементи. При видаленні елемента із масиву решту елементів треба перенумерувати. При вирішенні задач, в яких кількість елементів заздалегідь невідома, елементи треба часто видаляти і додавати, треба шукати інші способи зберігання. В мові **Java** з самих перших версій єсть клас **Vector**, призначений для зберігання змінного числа елементів самого загального типу **Object**.

6.1. Клас **Vector**

В класі **Vector** із пакету **java.util** зберігаються елементи типу **Object**, а значить, довільного типу. Кількість елементів може бути довільним і наперед не визначається. Елементи одержують індекси 0, 1, 2, До кожного елемента вектора можна звернутися по індексу, як і до елемента масиву. Крім кількості елементів, яку називають **розміром** (**size**) вектора, є ще що розмір буфера — **ємність** (**capacity**) вектора. Звичайно ємність співпадає з розміром вектора, але можна її збільшити методом **ensureCapacity(int minCapacity)** або порівняти з розміром вектора методом **trimToSize()**. В Java 2 клас **Vector** перероблений, щоб включити його в ієрархію класів-колекцій. Тому багато дій можна робити старими і новими методами. Рекомендується використовувати нові методи, оскільки старі можуть бути виключені із наступних версій **Java**.

6.1.1. Як створити вектор

В класі чотири конструктори:

- **Vector ()** - створює пустий об'єкт нульової довжини;
- **Vector (int capacity)** - створює пустий об'єкт указаної ємності **capacity**;
- **Vector (int capacity, int increment)** - створює пустий об'єкт указаної ємності **capacity** і задає число **increment**, на яке збільшується ємність при необхідності;
- **Vector (Collection c)** - вектор створюється по указаній колекції. Якщо **capacity** відємне, створюється виключна ситуація. Після створення вектору його можна заповнити елементами.

6.1.2. Як додати елемент у вектор

- Метод **add (Object element)** дозволяє додати елемент в кінець вектора (те ж саме робить старий метод **addElement (Object element)**).
- Методом **add (int index, Object element)** або старим методом **insertElementAt (Object element, int index)** можна вставити елемент у вказане місце **index**. Елемент, що знаходиться на цьому місці, і всі наступні елементи здвигуються, їх індекси збільшуються на одиницю.
- Метод **addAll (Collection coll)** дозволяє додати в кінець вектора всі елементи колекції **coll**.
- Методом **addAll(int index, Collection coll)** можна вставити в позицію **index** всі елементи колекції **coll**.

6.1.3. Як замінити елемент вектора

Метод **set (int index, object element)** заміняє елемент, що стоїть у векторі в позиції **index**, на елемент **element** (те ж саме дозволяє виконувати старий метод **setElementAt (Object element, int index)**)

6.1.4. Як узнати розмір вектора

Кількість елементів у векторі завжди можна віднайти методом **size()**. Метод **capacity()** повертає ємність вектора. Логічний метод **isEmpty()** повертає **true**, якщо у векторі немає жодного елемента.

6.1.5. Як звернутися до елемента вектора

Звернутися до першого елементу вектора можна методом **firstElement()**, до останнього - методом **lastElement()**, до довільного елементу - методом **get (int index)** або старим методом **elementAt (int index)**. Ці

методи повертають об'єкт класу `Object`. Перед використанням його треба привести до потрібного типу. Одержані всі елементи вектора у вигляді масиву типу `Object[]` можна методами `toArray()` і `toArray (Object [] a)`. Другий метод заносить всі елементи вектора в масив `a`, якщо в ньому достатньо місця.

6.1.6. Як узнати, чи є елемент у векторі

- Логічний метод `contains (Object element)` повертає `true`, якщо елемент `element` знаходиться у векторі.
- Логічний метод `containsAll (Collection c)` повертає `true`, якщо вектор містить всі елементи указаної колекції.

6.1.7. Як узнати індекс елемента

Чотири методи дозволяють знайти позицію указаного елемента `element`:

- `indexof (Object element)` - повертає індекс першої появи елемента у векторі;
- `indexOf (Object element, int begin)` - веде пошук, починаючи з індекса `begin` включно;
- `lastIndexof (Object element)` - повертає індекс останньої появи елемента у векторі;
- `lastIndexOf (Object element, int start)` - веде пошук від індекса `start` включно до початку вектора.

Якщо елемент не знайдено, повертається `-1`.

6.1.8. Як видалити елементи вектора

- Логічний метод `remove (Object element)` видаляє із вектора перше входження указаного елемента `element`. Метод повертає `true`, якщо елемент знайдено і видалення виконано.
- Метод `remove (int index)` видаляє елемент із позиції `index` і повертає його в якості свого результату типу `Object`. Аналогічні дії дозволяють виконати старі методи типу `void`: `removeElement (Object element)` і `removeElementAt (int index)`, що не повертають результату.
- Видалити діапазон елементів можна методом `removeRange(int begin, int end)`. Видаляються елементи від позиції `begin` включно до позиції `end` виключно.
- Видалити із даного вектора всі елементи колекції `coll` можна логічним методом `removeAll(Collection coll)`. Видалити останні елементи можна, просто урізавши вектор методом `setSize(int newSize)`.
- Видалити всі елементи, крім тих, що входять у вказану колекцію `coll`, дозволяє логічний метод `retainAll(Collection coll)`. Видалити всі елементи вектора можна методом `clear()` або старим методом `removeAllElements()` або обнуливши розмір вектора методом `setSize(0)`.

Лістинг 6.1 розширяє лістинг 5.1, обробляючи виділені із рядка слова за допомогою вектора.

Лістинг 6.1. Робота з вектором

```
import java.util.*;
class Parse{
public static void main(String[] args){
Vector v = new Vector();
String s = "String, that we want to decompose in words.";

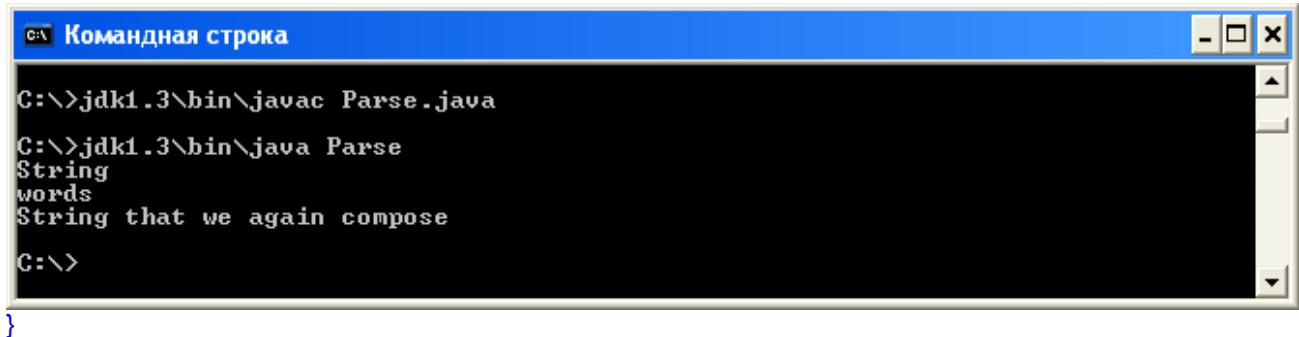
StringTokenizer st = new StringTokenizer(s, " \t\n\r,. ");

while (st.hasMoreTokens()){
// Одержано слово і заносимо у вектор
v.add(st.nextToken()); // Додаємо в кінець вектора
}
System.out.println(v.firstElement()); // Перший елемент
System.out.println(v.lastElement()); // Останній елемент
v.setSize(4); // Зменшуємо число елементів
v.add("compose"); // Додаємо в кінець вкороченого вектора
v.set(3, "again"); // Ставимо в позицію 3
}
```

```

for (int i = 0; i < v.size(); i++)// Перебираємо весь вектор
System.out.print(v.get(i) + " ");
System.out.println();
}
}

```



```

C:\>jdk1.3\bin\javac Parse.java
C:\>jdk1.3\bin\java Parse
String
words
String that we again compose
C:\>
}

```

Клас [Vector](#) являється прикладом того, як можна обєкти класу [Object](#), а значить, будь-які обєкти, об'єднувати в колекцію. Цей тип колекції упорядковує і навіть нумерує елементи. У векторі єсть перший елемент, єсть останній елемент. До кожного елементу звертаються безпосередньо по індексу. При додаванні і видаленні елементів решта елементів автоматично перенумеровуються.

Передивіться файл [Vector](#) у папці [java.util](#) і визначте, чи всі методи цього класу були задіяні у попередній програмі.

Другий приклад колекції — клас [Stack](#) — розширяє клас [Vector](#).

6.2. Клас Stack

Клас [Stack](#) із пакету [java.util](#) об'єднує елементи в стек. *Стек (stack)* реалізує порядок роботи з елементами подібно магазину гвинтівки — першим вистрілить патрон, покладений в магазин останнім, — або подібно залізничному тутику — першим із тутика вийде вагон, загнаний туди останнім. Такий порядок обробки називається [LIFO \(Last In — First Out\)](#).

Перед роботою створюється пустий стек конструктором [Stack \(\)](#). Потім на стек кладутся і знімаються елементи, причому доступний тільки "верхній" елемент, той, що покладений на стек останнім. Додатково до методів класу [Vector](#) клас [Stack](#) містить п'ять методів, що дозволяють працювати з колекцією як зі стеком:

- [push \(Object item\)](#) — поміщає елемент [item](#) в стек;
- [pop \(\)](#) — дістає верхній елемент зі стека;
- [peek \(\)](#) — читає верхній елемент, не дістаючи його зі стека;
- [empty \(\)](#) — перевіряє, чи не пустий стек;
- [search \(object item\)](#) — знаходить позицію елемента [item](#) в стеку. Верхній елемент має позицію 1, під ним елемент 2 і т. д. Якщо елемент не знайдено, повертається - 1.

Лістинг 6.2 показує, як можна використовувати стек для перевірки парності дужок.

Лістинг 6.2. Перевірка парності дужок

```

import java.util.*;
class StackTest{
static boolean checkParity(String expression,
String open, String close){
Stack stack = new Stack ();
StringTokenizer st = new StringTokenizer(expression, " \t\n\r+*/-(){}", true);
while (st.hasMoreTokens ()) {
}
}
}

```

```

String tmp = st.nextToken();
if (tmp.equals(open)) stack.push(open);
if (tmp.equals(close)) stack.pop();
}
if (stack.isEmpty ()) return true;
else
return false;
}
public static void main(String[] args){
System.out.println(
checkParity("a - (b - (c - a) / (b + c) - 2" , "(", ")"));
}
}

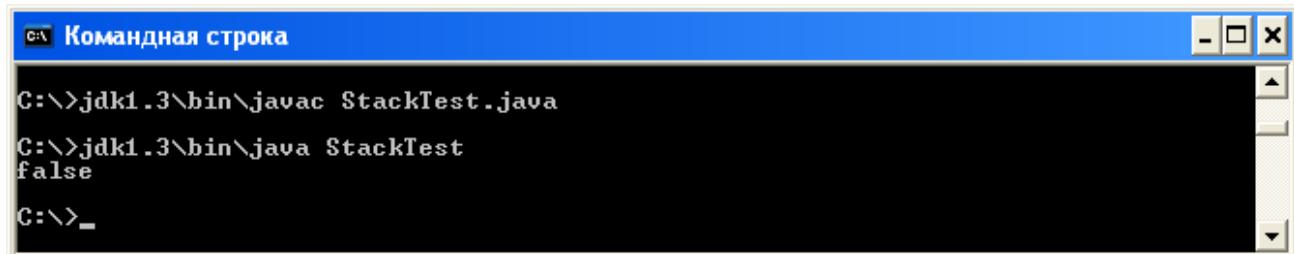
```

Передивіться файл [Stack](#) у папці `java.util` і визначте, чи всі методи цього класу були задіяні у попередній програмі.

Як бачимо, колекції значно полегшують обробку наборів даних. Ще один приклад колекції зовсім іншого роду - таблиці — представляє клас [Hashtable](#).

6.3. Клас [Hashtable](#)

Клас [Hashtable](#) розширює абстрактний клас [Dictionary](#). В об'єктах цього класу зберігаються пари "ключ - значення". Із таких пар "Прізвище І. Б. — номер" складається, наприклад, телефонний довідник. Ще один приклад - анкета. Її можна представити як сукупність пар "Прізвище - Іванов", "Ім'я — Петро", "По батькові - Сидорович", "Рік народження — 1975" і т. д. Подібних прикладів можна привести багато. Кожний об'єкт класу [Hashtable](#) крім [розміру \(size\)](#) - кількості пар, має ще дві характеристики: [ємність \(capacity\)](#) - розмір буфера, і [показник завантаженості \(load factor\)](#) — процент заповнення буфера, по досягненні якого збільшується його розмір.



```

Командная строка

C:\>jdk1.3\bin\javac StackTest.java
C:\>jdk1.3\bin\java StackTest
false
C:\>_

```

6.3.1. Як створити таблицю

Для створення об'єктів клас [Hashtable](#) має чотири конструктори:

- [Hashtable \(\)](#) - створює пустий об'єкт з початковою ємністю в 101 елемент і показником завантаженості 0,75;
- [Hashtable \(int capacity\)](#) - створює пустий об'єкт з початковою ємністю `capacity` і показником завантаженості 0,75;
- [Hashtable\(int capacity, float loadFactor\)](#) - створює пустий об'єкт з початковою ємністю `capacity` і показником завантаженості `loadFactor`;
- [Hashtable \(Map f\)](#) - створює об'єкт класу [Hashtable](#), що містить всі елементи відображення `f`, з ємністю, рівною подвоєному числу елементів відображення `f`, але не менше 11, і показником завантаженості 0,75.

6.3.2. Як заповнити таблицю

Для заповнення об'єкту класу [Hashtable](#) використовуються два методи:

- [Object put\(Object key, Object value\)](#) — додає пару "key— value", якщо ключа `key` не було в таблиці, і змінює значення `value` ключа `key`, якщо він уже єсть в таблиці. Повертає старе значення ключа

- або `null`, якщо його не було. Якщо хоч би один параметр рівний `null`, виникає виключна ситуація;
- метод `void putAll(Map f)` — додає всі елементи відображення `f`. В обєктах-ключах `key` повинні бути реалізовані методи `hashCode()` і `equals()`.

6.3.3. Як одержати значення по ключу

Метод `get (Object key)` повертає значення елемента з ключем `key` у вигляді обєкта класу `object`. Для подальшої роботи його треба перетворити в конкретний тип.

6.3.4. Як узнати наявність ключа або значення

- Логічний метод `containsKey(object key)` повертає `true`, якщо в таблиці є ключ `key`.
- Логічний метод `containsValue (Object value)` або старий метод `contains (Object value)` повертає `true`, якщо в таблиці є ключі із значенням `value`.
- Логічний метод `isEmpty()` повертає `true`, якщо в таблиці немає елементів.

6.3.5. Як одержати всі елементи таблиці

- Метод `values()` представляє всі значення `value` таблиці у вигляді інтерфейсу `Collection`. Всі модифікації в обєкті `Collection` змінюють таблицю, і навпаки.
- Метод `keyset()` представляє всі ключі `key` таблиці у вигляді інтерфейсу `Set`. Всі зміни в обєкті `Set` коректують таблицю, і навпаки.
- Метод `entrySet()` представляє всі пари "key— value" таблиці у вигляді інтерфейсу `Set`. Всі зміни в обєкті `Set` коректують таблицю, і навпаки.
- Метод `toString()` повертає рядок, що містить всі пари.
- Старі методи `elements()` і `keys()` повертають значення і ключі у вигляді інтерфейсу `Enumeration`.

6.3.6. Як видалити елементи

- Метод `remove (Object key)` видаляє пару з ключем `key`, повертаючи значення цього ключа, якщо воно єсть, і `null`, якщо пара з ключем `key` не знайдена.
- Метод `clear()` видаляє всі елементи, очищаючи таблицю.

В лістингі 6.3 показано, як можна використати клас `Hashtable` для створення телефонного довідника, а на рис. 6.1 — виведення цієї програми.

Лістинг 6.3. Телефонний довідник

```
import java.util.*;
class PhoneBook{
public static void main(String[] args){
Hashtable yp = new Hashtable();
String name = null;
yp.put("John", "123-45-67");
yp.put ("Lemon", "567-34-12");
yp.put("Bill", "342-65-87");
yp.put("Gates", "423-83-49");
yp.put("Batman", "532-25-08");
try{
name = args[0];
}
catch(Exception e){
System.out.println("Usage: Java PhoneBook Name");
return;
}
if (yp.containsKey(name))
System.out.println(name + "'s phone = " + yp.get(name));
else
System.out.println("Sorry, no such name");
```

```

}
}

```

Передивіться файл [Hashtable](#) у папці [java.util](#) і визначте, чи всі методи цього класу були задіяні у

```

Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\User>cd C:\

C:\>jdk1.3\bin\javac PhoneBook.java

C:\>jdk1.3\bin\java PhoneBook Lemon
Lemon's phone = 567-34-12

C:\>-

```

попередній програмі.

Рис. 6.1. Робота з телефонною книгою

6.4. Клас Properties

Клас [Properties](#) розширяє клас [Hashtable](#). Він призначений в основному для введення і виведення пар властивостей системи і їх значень. Пари зберігаються у вигляді рядків типу [String](#). В класі [Properties](#) два конструктори:

- [Properties \(\)](#) - створює пустий об'єкт;
- [Properties \(Properties default\)](#) - створює об'єкт із заданими параметрами властивостей [default](#).

Крім унаслідуваних від класу [Hashtable](#) методів в класі [Properties](#) є ще наступні методи. Два методи, що повертають значення ключа-рядка у вигляді рядка:

- [String getProperty \(string key\)](#) - повертає значення по ключу [key](#);
- [String getProperty\(String key, String defaultValue\)](#) - повертає значення по ключу [key](#); якщо такого ключа немає, повертається [defaultValue](#).
- Метод [setProperty\(String key, String value\)](#) додає нову пару, якщо ключа [key](#) немає, і змінює значення, якщо ключ [key](#) єсть.
- Метод [load\(InputStream in\)](#) завантажує властивості із входного потоку [in](#).
- Методи [list\(PrintStream out\)](#) і [list \(PrintWriter out\)](#) виводять властивості у вихідний потік [out](#).
- Метод [store \(OutputStream out, String header\)](#) виводить властивості у вихідний потік [out](#) із заголовком [header](#).

Дуже простий лістинг 6.4 і рис. 6.2 демонструють виведення всіх системних властивостей [Java](#).

Лістинг 6.4. Виведення системних властивостей

```

class Prop{
    public static void main(String[] args){
        System.getProperties().list(System.out);
    }
}

```

Передивіться файл **Properties** у папці **java.util** і визначте, як ще можна використати цей клас.

Рис. 6.2. Системні властивості

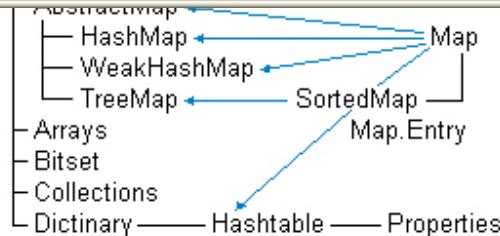
Приклади класів **Vector**, **Stack**, **Hashtable**, **Properties** показують зручності класів-колекцій. Тому в **Java 2** розроблена ціла ієрархія колекцій. Вона показана на рис. 6.3. Курсивом записані імена інтерфейсів. Пунктирні лінії вказують класи, що реалізують ці інтерфейси. Всі колекції розбиті на три групи, описані в інтерфейсах **List**, **Set** і **Map**.

```
С:\>jdk1.3\bin\javac Prop.java
С:\>jdk1.3\bin\java Prop
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=C:\jdk1.3\jre\bin
java.vm.version=1.3.0-C
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
path.separator=;
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
java.vm.specification.name=Java Virtual Machine Specification
user.dir=C:\
java.runtime.version=1.3.0-C
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
os.arch=x86
java.io.tmpdir=C:\DOCUME~1\User\LOCALS~1\Temp\
line.separator=
java.vm.specification.vendor=Sun Microsystems Inc.
java.awt.fonts=
os.name=Windows 2000
java.library.path=C:\jdk1.3\bin;.;C:\WINDOWS\system32;C...
java.specification.name=Java Platform API Specification
java.class.version=47.0
os.version=5.1
user.home=C:\Documents and Settings\User
user.timezone=
java.awt.printerjob=sun.awt.windows.WPrinterJob
file.encoding=Cp1251
java.specification.version=1.3
user.name=User
java.class.path=
java.vm.specification.version=1.0
java.home=C:\jdk1.3\jre
user.language=uk
java.specification.vendor=Sun Microsystems Inc.
awt.toolkit=sun.awt.windows.WToolkit
java.vm.info=mixed mode
java.version=1.3.0
java.ext.dirs=C:\jdk1.3\jre\lib\ext
sun.boot.class.path=C:\jdk1.3\jre\lib\rt.jar;C:\jdk1.3\jre...
java.vendor=Sun Microsystems Inc.
file.separator\
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
sun.cpu.endian=little
sun.io.unicode.encoding=UnicodeLittle
user.region=UA
sun.cpu.isalist=pentium i486 i386
C:\>
```

Рис. 6.3.
Ієрапхія класів в інтерфейсів-колекцій

Приладо
М реалізації
інтерфейсів

у **List** може служити клас **Vector**, прикладом реалізації інтерфейсу **Map** — клас **Hashtable**. Колекції **List** і **Set** мають багато спільного, тому їх спільні методи об'єднані і



винесені в суперінтерфейс [Collection](#). Подивимось, що, на думку розробників [Java API](#), повинно міститися в цих колекціях.

6.5. Інтерфейс Collection

Інтерфейс [Collection](#) із пакету [java.util](#) описує загальні властивості колекцій [List](#) і [Set](#). Він містить методи додавання і видалення елементів, перевірки і перетворення елементів:

- [boolean add\(Object obj\)](#) - додає елемент [obj](#) в кінець колекції; повертає [false](#), якщо такий елемент в колекції уже єсть, а колекція не допускає повторних елементів; повертає [true](#), якщо додавання пройшло успішно;
- [boolean addAll\(Collection coll\)](#) - додає всі елементи колекції [coll](#) в кінець даної колекції;
- [void clear\(\)](#) - видаляє всі елементи колекції;
- [boolean contains\(Object obj\)](#) - перевіряє наявність елементу [obj](#) в колекції;
- [boolean containsAll\(Collection coll\)](#) - перевіряє наявність всіх елементів колекції [coll](#) в даній колекції;
- [boolean isEmpty\(\)](#) - перевіряє, чи колекція пуста ;
- [Iterator iterator\(\)](#) - повертає ітератор даної колекції;
- [boolean remove\(object obj\)](#) — видаляє указаний елемент із колекції; повертає [false](#), якщо елемент не знайдено, [true](#), якщо видалення пройшло успішно;
- [boolean removeAll \(Collection coll\)](#) — видаляє елементи указаної колекції, що належать даній колекції;
- [boolean retainAll\(Collection coll\)](#) - видаляє всі елементи даної колекції, крім елементів колекції [coll](#);
- [int size\(\)](#) - повертає кількість елементів в колекції;
- [Object \[\] toArray \(\)](#) - повертає всі елементи колекції у вигляді масиву;
- [Object\[\] toArray<object>\[\] a](#) - записує всі елементи колекції в массив [a](#), якщо в ньому достатньо місця.

Передивіться файл [java.util\Interfaces\Collection](#).

6.6. Інтерфейс List

Інтерфейс [List](#) із пакету [java.util](#), що розширює інтерфейс [Collection](#), описує методи роботи з упорядкованими колекціями. Інколи їх називають *послідовностями (sequence)*. Елементи такої колекції перенумеровані, починаючи з нуля, до них можна звернутися по індексу. На відміну від колекції [Set](#) елементи колекції [List](#) можуть повторятися. Клас [Vector](#) — одна із реалізацій інтерфейсу [List](#). Інтерфейс [List](#) додає до методів інтерфейсу [Collection](#) методи, що використовують індекс [index](#) елемента:

- [void add\(int index, object obj\)](#) - вставляє елемент [obj](#) в позицію [index](#); старі елементи, починаючи з позиції [index](#), зсуванняться, їх індекси збільшуються на одиницю;
- [boolean addAll\(int index, Collection coll\)](#) - вставляє всі елементи колекції [coll](#);
- [object get\(int index\)](#) - повертає елемент, що знаходиться в позиції [index](#);
- [int indexOf\(Object obj\)](#) - повертає індекс першої появи елемента [obj](#) в колекції;
- [int lastIndexOf \(object obj\)](#) - повертає індекс останньої появи елемента [obj](#) в колекції;
- [ListIterator listIterator \(\)](#) - повертає ітератор колекції;
- [ListIterator listIterator \(int index\)](#) — повертає ітератор кінця колекції від позиції [index](#);
- [object set \(int index, object obj\)](#) — заміняє елемент, що знаходиться в позиції [index](#), елементом [obj](#);
- [List subList\(int from, int to\)](#) — повертає частину колекції від позиції [from](#) включно до позиції [to](#) виключно.

Передивіться файл [java.util\Interfaces>List](#).

6.7. Інтерфейс Set

Інтерфейс [Set](#) із пакету [java.util](#) розширює інтерфейс [Collection](#), описуючи неупорядковану колекцію, яка не містить повторюваних елементів. Це відповідає математичному поняттю *множини (set)*. Такі колекції зручні для перевірки наявності або відсутності у елемента властивості, що визначає множину. Нові методи в інтерфейс [Set](#) не додані, просто метод [add \(\)](#) не стане додавати ще одну копію елемента, якщо такий елемент уже єсть в множині. Цей інтерфейс розширяється інтерфейсом [SortedSet](#).

Передивіться файл [java.util\Interfaces\Set](#).

6.8. Інтерфейс SortedSet

Інтерфейс [SortedSet](#) із пакету [java.util](#), розширює інтерфейс [Set](#) і описує упорядковану множину, відсортовану по природному порядку зростання його елементів або по порядку, заданому реалізацією інтерфейсу [comparator](#). Елементи не нумеруються, але єсть поняття першого, останнього, більшого і меншого елемента. Додаткові методи інтерфейсу відображають ці поняття:

- [comparator comparator\(\)](#) - повертає спосіб упорядкування колекції; [Object first \(\)](#) - повертає перший, менший елемент колекції;
- [SortedSet headSet\(Object toElement\)](#) — повертає початкові, менші елементи до елементу [toElement](#) виключно;
- [Object last\(\)](#) - повертає останній, більший елемент колекції;
- [SortedSet subSet\(Object fromElement, Object toElement\)](#) - повертає підмножину колекції від елемента [fromElement](#) включно до елемента [toElement](#) виключно;
- [SortedSet tailSet\(Object fromElement\)](#) - повертає останні, більші елементи колекції від елемента [fromElement](#) включно.

Передивіться файл [java.util\Interfaces\SortedSet](#).

6.9. Інтерфейс Map

Інтерфейс [Map](#) із пакету [java.util](#) описує колекцію, що складається із пар "ключ — значення". У кожного ключа тільки одне значення, що відповідає математичному поняттю однозначної функції або відображення ([map](#)). Таку колекцію часто називають ще [словником](#) ([dictionary](#)) або [асоціативним масивом](#) ([associative array](#)). Звичайний масив — найпростіший приклад словника із заздалегідь заданим числом елементів. Це відображення множини перших невідемних цілих чисел на множину елементів масиву, множина пар "індекс масиву - елемент масиву". Клас [HashTable](#) - одна із реалізацій інтерфейсу [Map](#). Інтерфейс [Map](#) містить методи, що працюють з ключами і значеннями:

- [boolean containsKey\(Object key\)](#) - перевіряє наявність ключа [key](#);
- [boolean containsValue\(Object value\)](#) - перевіряє наявність значення [value](#);
- [Set entrySet\(\)](#) — представляє колекцію у вигляді множини, кожний елемент якої — пара із даного відображення, з якою можна працювати методами вкладеного інтерфейсу [Map.Entry](#);
- [Object get\(Object key\)](#) - повертає значення, відповідне ключу [key](#);
- [Set keySet\(\)](#) - представляє ключі колекції у вигляді множини;
- [Object put\(Object key, Object value\)](#) — додає пару "key—value", якщо такої пари не було, і замінює значення ключа [key](#), якщо такий ключ уже єсть в колекції;
- [void putAll\(Map m\)](#) - додає до колекції всі пари із відображення [m](#);
- [Collection values\(\)](#) - представляє всі значення у вигляді колекції.

В інтерфейс [Map](#) вкладений інтерфейс [Map.Entry](#), який містить методи роботи з окремою парою.

Передивіться файл [java.util\Interfaces\Map](#).

6.10. Вкладений інтерфейс Map.Entry

Цей інтерфейс описує методи роботи зарами, отриманими методом [entrySet\(\)](#):

- методи [getKey\(\)](#) і [getValue\(\)](#) дозволяють отримати ключ і значення пари;
- метод [setValue\(Object value\)](#) змінює значення в даній парі.

Передивіться файл [java.util\Interfaces\Map.Entry](#)

6.11. Інтерфейс SortedMap

Інтерфейс [SortedMap](#), розширює інтерфейс [Map](#) і описує впорядковану по ключах колекцію [Map](#). Сортування відбувається або в природному порядку зростання ключів, або, в порядку, описаному в інтерфейсі [Comparator](#). Елементи не нумеруються, але єсть поняття більшого і меншого із двох елементов, першого, самого маленького, і останнього, самого більшого елемента колекції. Ці поняття описуються наступними методами:

- [Comparator comparator\(\)](#) - повертає спосіб упорядочення колекції;
- [Object firstKey\(\)](#) - повертає перший, найменший елемент колекції;
- [SortedMap headMap\(Object toKey\)](#) - повертає початок колекції до елемента з ключем [toKey](#) виключно;
- [Object lastKey\(\)](#) - повертає останній, найбільший ключ колекції;
- [SortedMap subMap \(Object fromKey, Object toKey\)](#) - повертає частину колекції від елемента з ключем [fromKey](#) включно до елемента з ключем [toKey](#) виключно;
- [SortedMap tailMap\(Object fromKey\)](#) - повертає останчу колекції від елементу [fromKey](#) включно.

Ви можете створювати свої колекції, реалізувавши розглянуті інтерфейси. Це справа важка, оскільки в інтерфейсах багато методів. Щоб полегшити цю задачу, в Java API введені частинні реалізації інтерфейсів — абстрактні класи-колекції.

Передивіться файл [java.util\Interfaces\SortedMap](#).

6.12. Абстрактні класи-колекції

Ці класи лежать в пакеті [java.util](#).

- Абстрактний клас [AbstractCollection](#) реалізує інтерфейс [Collection](#), але залишає нереалізованими методи [iterator\(\)](#), [size\(\)](#).
- Абстрактний клас [AbstractList](#) реалізує інтерфейс [List](#), але залишає нереалізованим метод [get\(\)](#) і наслідуваний метод [size\(\)](#). Цей клас дозволяє реалізувати колекцію з прямим доступом до елементів, подібно масиву
- Абстрактний клас [AbstractSequentialList](#) реалізує інтерфейс [List](#), але залишає нереалізованим метод [listIterator\(int index\)](#) і наслідуваний метод [size\(\)](#). Даний клас дозволяє реалізувати колекцію з послідовним доступом до елементів за допомогою ітератора [ListIterator](#).
- Абстрактний клас [AbstractSet](#) реалізує інтерфейс [Set](#), але залишає нереалізованими методи, наслідувані від [AbstractCollection](#).
- Абстрактний клас [AbstractMap](#) реалізує інтерфейс [Map](#), але залишає нереалізованим метод [entrySet\(\)](#).

Нарешті, в складі Java API єсть повністю реалізовані класи-колекції помимо уже розглянутих класів [Vector](#), [Stack](#), [Hashtable](#) і [Properties](#). Це класи [ArrayList](#), [LinkedList](#), [HashSet](#), [TreeSet](#), [HashMap](#), [TreeMap](#), [WeakHashMap](#). Для роботи з цими класами розроблені інтерфейси [Iterator](#), [ListIterator](#), [Comparator](#) і класи [Arrays](#) та [Collections](#). Перед тим як розглянути використання даних класів, обговоримо поняття ітератора.

6.13. Інтерфейс [Iterator](#)

В 70—80-х роках минулого століття, після того як була усвідомлена важливість правильної організації даних у певну структуру, велика увага приділялась вивченню і побудові різних структур даних: звязаних списків, черг, деків, стеків, дерев, мереж. Разом з розвитком структур даних розвивались і алгоритми роботи з ними: сортування, пошук, обхід, хешування. Цим питанням присвячена обширна література. В 90-х роках було вирішено заносити дані в певну колекцію, заховавши її внутрішню структуру, а для роботи з даними використовувати методи цієї колекції. Зокрема, завдання обходу поклали на саму колекцію. В Java API введений інтерфейс [Iterator](#), який описує спосіб обходу всіх елементів колекції. В кожній колекції єсть метод [Iterator\(\)](#), який повертає реалізацію інтерфейса [Iterator](#) для вказаної колекції. Отримавши цю реалізацію, можна обходити колекцію в деякому порядку, визначеним даним ітератором, за допомогою методів, описаних в інтерфейсі [Iterator](#) і реалізованих в цьому ітераторі. Подібна техніка використана в класі [StringTokenizer](#). В інтерфейсі [Iterator](#) описані всього три методи:

- логічний метод [hasNext\(\)](#) повертає [true](#), якщо обхід ще не завершено;
- метод [next\(\)](#) робить поточним наступний елемент колекції і повертає його у вигляді обєкта класу

- `Object`;
- метод `remove()` видаляє поточний елемент колекції.

Можна представити собі справу так, що ітератор — це покажчик на елемент колекції. При створенні ітератора покажчик установлюється перед першим елементом, метод `next()` переміщує показчик на перший елемент і показує його. Наступне застосування методу `next()` переміщає показчик на другий елемент колекції і показує його. Останнє застосування методу `next()` виводить показчик за останній елемент колекції. Метод `remove()`, здається, лишній, він уже не відноситься до завдання обходу колекції, але дозволяє при перегляді колекції видалити із нього непотрібні елементи.

В лістингі 6.5 до тексту лістинга 6.1 додана робота з ітератором.

Лістинг 6.5. Використання ітератора вектора

```
import java.util.*;
class Vect {
public static void main(String[] args) {
Vector v = new Vector();
String s = "String, that we want to decompose in words.";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,. ");
while (st.hasMoreTokens()) {
    // Одержано слово і заносимо у вектор.
    v.add(st.nextToken()); // Додаємо в кінець вектора }
System.out.println(v.firstElement()); // Перший елемент
System.out.println(v.lastElement()); // Останній елемент
v.setSize(4); // Зменшуємо число елементів
    v.add("compose."); // Додаємо в кінці скороченого вектора
v.set(3, "again"); // Ставимо в позицію 3
for (int i = 0; i < v.size(); i++) // Перебираємо увесь вектор
System.out.print(v.get(i) + ".");
System.out.println();
Iterator it = v.iterator(); // Одержано ітератор вектора
try{
    while(it.hasNext())// Доки у векторі єсть елементи,
    System.out.println(it.next()); // виводимо поточний елемент
} catch(Exception e){}
}
}
}
```

Запустіть програму на виконання і ретельно проаналізуйте результат.

6.14. Інтерфейс ListIterator

Інтерфейс `ListIterator` розширяє інтерфейс `Iterator`, забезпечуючи переміщення по колекції як в прямому, так і в зворотному напрямі. Він може бути реалізований лише в тих колекціях, в яких єсть поняття наступного і попереднього елемента і де елементи пронумеровані. В інтерфейс `ListIterator` додані наступні методи:

- `void add(Object element)` - додає елемент `element` перед поточним елементом;
- `boolean hasPrevious()` - повертає `true`, якщо в колекції єсть елементи, що стоять перед поточним елементом;
- `int nextIndex()` - повертає індекс поточного елемента; якщо поточним являється останній елемент колекції, повертає розмір колекції;
- `Object previous()` - повертає попередній елемент і робить його поточним;
- `int previousIndex()` - повертає індекс попереднього елемента;
- `void set(Object element)` - замінює поточний елемент елементом `element` - виконується зразу після `next()` або `previous()`.

Як бачимо, ітератори можуть змінювати колекцію, в якій вони працюють, додаючи, видаляючи і заміняючи елементи. Щоб це не приводило до конфліктів, передбачена виключна ситуація, яка виникає при намаганні використовувати ітератори паралельно "рідним" методам колекції. Якраз тому в лістингі 6.5 дії з ітератором заключені в блок `try{}catch({})`.

Змінимо закінчення лістингу 6.5 з використанням ітератора `ListIterator`.

```
// Текст лістинга 6.1...
// ...
ListIterator lit = v.listIterator(); // Отримуємо ітератор вектора
// Показчик зараз знаходиться перед початком вектора
try{
    while(lit.hasNext()) // Доки у вектора єсть елементи
        System.out.println(lit.next()); //Переходимо до наступного елемента і виводимо його
    // Тепер показчик за кінцем вектора. Переїдемо в початок
    while (lit. hasPrevious ())
        System.out.println(lit. previous ());
} catch (Exception e) {}
```

Цікаво, що повторне використання методів `next()` і `previous()` один за другим буде видавати один і той же поточний елемент. Подивимось тепер, які можливості представляють класи-колекції **Java 2**.

6.15. Класи, що створюють списки

Клас `ArrayList` повністю реалізує інтерфейс `List` і ітератор типу `iterator`. Клас `ArrayList` дуже схожий на клас `Vector`, має той же набір методів і може використовуватися в тих же ситуаціях. В класі `ArrayList` три конструктори:

- `ArrayList ()` - створює пустий об'єкт;
- `ArrayList (Collection coll)` - створює об'єкт, який містить всі елементи колекції `coll`;
- `ArrayList (int initCapacity)` - створює пустий об'єкт ємності `initCapacity`.

Єдина відмінність класу `ArrayList` від класу `Vector` заключається в тому, що клас `ArrayList` не синхронізований. Це означає, що одночасна зміна экземпляру цього класу декількома підпроцесами приведе до непередбачуваних результатів. Ці питання розглянемо в уроці 17.

6.16. Двонаправлений список

Клас `LinkedList` повністю реалізує інтерфейс `List` і містить додаткові методи, що перетворюють його в двонаправлений список. Він реалізує ітератори типу `iterator` і `bisIterator`. Цей клас можна використовувати для обробки елементів в стеку, деку або двонаправленому списку. В класі `LinkedList` два конструктори:

- `LinkedList` - створює пустий об'єкт
- `LinkedList (Collection coll)` — створює об'єкт, що містить всі елементи колекції `coll`.

6.17. Класи, що створюють відображення

Клас `HashMap` повністю реалізує інтерфейс `Map`, а також ітератор типу `iterator`. Клас `HashMap` дуже схожий на клас `Hashtable` і може використовуватися в тих же ситуаціях. Він має той же набір функцій і такі ж конструктори:

- `HashMap()` - створює пустий об'єкт з показником завантаженості 0,75;
- `HashMap(int capacity)` - створює пустий об'єкт з початковою ємністю `capacity` і показником завантаженості 0,75;
- `HashMap(int capacity, float loadFactor)` - створює пустий об'єкт з початковою ємністю `capacity` і показником завантаженості `loadFactor`;
- `HashMap(Map f)` - створює об'єкт класу `HashMap`, що містить всі елементи відображення `f`, з ємністю, рівною подвоєному числу елементів відображення `f`, але не менше 11, і показником завантаженості 0,75.

- Клас [WeakHashMap](#) відрізняється від класу [HashMap](#) тільки тим, що в його обєктах не використовувані елементи, на які ніхто не посилається, автоматично виключаються із обєкта.

6.18. Упорядковані відображення

Клас [TreeMap](#) повністю реалізує інтерфейс [sortedMap](#). Він реалізований як бінарне дерево пошуку, значить його елементи зберігаються в упорядкованому вигляді. Це значно прискорює пошук потрібного елемента. Порядок задається або природним слідуванням елементів, або об'єктом, реалізуючим інтерфейс няння [Comparator](#). В цьому класі чотири конструктори:

- [TreeMap \(\)](#) — створює пустий об'єкт з природним порядком елементів;
- [TreeMap \(Comparator c\)](#) — створює пустий об'єкт, в якому порядком елементів задається об'єктом порівняння [c](#);
- [TreeMap \(Map f\)](#) — створює об'єкт, що містить всі елементи відображення [f](#), з природним порядком його елементів;
- [TreeMap \(SortedMap sf\)](#) — створює об'єкт, що містить всі елементи відображення [sf](#) в тому ж порядку.

Тут треба пояснити, яким способом можна задати упорядкованість елементів колекції

6.19. Порівняння елементів колекцій

Інтерфейс [Comparator](#) описує два методи порівняння:

- [int compare\(Object obj1, Object obj2\)](#) - повертає відємне число, якщо [obj1](#) в певному змісті менше [obj2](#); нуль, якщо вони вважаються рівними; додатне число, якщо [obj1](#) більше [obj2](#). Для студентів, знайомих з теорією множин, скажемо, що цей метод порівняння має властивості рефлексивності, антисиметричності і транзитивності;
- [boolean equals\(Object obj\)](#) - порівнює даний об'єкт з об'єктом [obj](#), повертає [true](#), якщо об'єкти співпадають в певному змісті, заданому цим методом.

Для кожної колекції можна реалізувати ці два методи, задавши конкретний спосіб порівняння елементів, і визначити об'єкт класу [SortedMap](#) другим конструктором. Елементи колекції будуть автоматично відсортовані в заданому порядку. Лістинг 6.6 показує один з можливих способів упорядкованості комплексних чисел — об'єктів класу [Complex](#) із лістингу 2.4. Тут описується клас [ComplexCompare](#), реалізуючий інтерфейс [Comparator](#). В лістингі 6.7 він застосовується для упорядкованого зберігання множини комплексних чисел.

Лістинг 6.6. Порівняння комплексних чисел

```
import java.util.*;
class ComplexCompare implements Comparator{
public int compare(Object obj1, Object obj2){
Complex z1 = (Complex)obj1, z2 = (Complex)obj2;
double re1 = z1.getRe(), im1 = z1.getIm();
double re2 = z2.getRe(), im2 = z2.getIm();
if (re1 != re2) return (int)(re1 - re2);
else if (im1 != im2) return (int)(im1 - im2);
else return 0;
}
public boolean equals(Object z) {
return compare(this, z) == 0;
}
}
```

6.20. Класи, що створюють множини

Клас [HashSet](#) повністю реалізує інтерфейс [set](#) і ітератор типу [iterator](#). Клас [HashSet](#) використовується в тих випадках, коли треба зберігати тільки одну копію кожного елемента. В класі [HashSet](#) чотири

конструктори:

- `HashSet ()` - створює пустий об'єкт з показником завантаженості 0,75;
- `HashSet (int capacity)` - створює пустий об'єкт з початковою ємністю `capacity` і показником завантаженості 0,75;
- `HashSet (int capacity, float loadFactor)` — створює пустий об'єкт з початковою ємністю `capacity` і показником завантаженості `loadFactor`;
- `HashSet (Collection coll)` — створює об'єкт класу `HashMap`, що містить всі елементи відображення `coll`, з ємністю, рівною подвоєному числу елементів відображення `f`, але не менше 11, і показником завантаженості 0,75.

6.21. Упорядковані множини

Клас `TreeSet` повністю реалізує інтерфейс `sortedSet` і ітератор типу `iterator`. Клас `TreeSet` реалізований як бінарне дерево пошуку, значить, його елементи зберігаються в упорядкованому вигляді. Це значно прискорює пошук потрібного елемента. Порядок задається або природним слідуванням елементів, або об'єктом, реалізуючим інтерфейс порівняння `Comparator`. Цей клас зручний при пошуку елемента в множині, наприклад, для перевірки, чи володіє якийсь елемент властивістю, що визначає множину. В класі `TreeSet` чотири конструктори:

- `TreeSet()` - створює пустий об'єкт з природним порядком елементів;
- `TreeSet (Comparator c)` - створює пустий об'єкт, в якому порядок задається об'єктом порівняння `c`;
- `TreeSet (Collection coll)` - створює об'єкт, що містить всі елементи колекції `coll`, з природним порядком їх елементів;
- `TreeSet (SortedMap sf)` - створює об'єкт, що містить всі елементи відображення `sf`, в тому ж порядку.

В лістингі 6.7 показано, як можна зберігати комплексні числа в упорядкованому вигляді. Порядок задається об'єктом класу `ComplexCompare`, визначеного в лістингі 6.6.

Лістинг 6.7. Зберігання комплексних чисел в упорядкованому вигляді

```
TreeSet ts = new TreeSet (new ComplexCompare ());
ts.add(new Complex(1.2, 3.4));
ts.add(new Complex (-1.25, 33.4));
ts.add(new Complex(1.23, -3.45));
ts.add(new Complex(16.2, 23.4));
Iterator it = ts.iterator();
while(it.hasNext()) , ((Complex)it.next()).pr();
```

6.22. Дії з колекціями

Колекції призначені для зберігання елементів у зручному для подальшої обробки вигляді. Дуже часто обробка заключається в сортуванні елементів і пошуку потрібного елемента. Ці і інші методи обробки зібрані в класі `Collections`.

6.23. Методи класу Collections

Всі методи класу `Collections` статичні, ними можна користуватися, не створюючи екземпляри класу `Collections`. Як звичайно в статичих методах, колекція, з якою працює метод, задається його аргументом. Сортування може бути зроблена лише в упорядкованій колекції, реалізуючій інтерфейс `List`. Для сортування в класі `Collections` є два методи:

- `static void sort (List coll)` - сортує в природному порядку зростання колекцію `coll`, реалізуючу інтерфейс `List`;
- `static void sort (List coll, Comparator c)` - сортує колекцію `coll` в порядку, заданому об'єктом `c`. Після сортування можна здійснювати бінарний пошук в колекції.

- `static int binarySearch(List coll, Object element)` - відшукує елемент `element` у відсортованій в природному порядку зростання колекції `coll` і повертає індекс елемента або відємне число, якщо елемент не знайдено; відємне число показує індекс, з яким елемент `element` був би вставлений в колекцію, з протилежним знаком;
- `static int binarySearch(List coll, Object element, Comparator c)` - те ж, але колекція відсортована в порядку, визначеному об'єктом `c`.

Чотири методи знаходять найбільший і найменший елементи в упорядкованій колекції:

- `static Object max(Collection coll)` - повертає найбільший в природному порядку елемент колекції `coll`;
- `static Object max(Collection coll, Comparator c)` - те ж в порядку, заданому об'єктом `c`;
- `static Object min(Collection coll)` - повертає найменший в природному порядку елемент колекції `coll`;
- `static Object min(Collection coll, Comparator c)` - те ж в порядку, заданому об'єктом `c`.

Два методи "перемішують" елементи колекції випадковим способом:

- `static void shuffle (List coll)` - випадкові числа задаються по замовчуванню;
- `static void shuffle (List coll, Random r)` - випадкові числа визначаються об'єктом `r`.
- Метод `reverse (List coll)` змінює порядок розташування елементів на зворотний.
- Метод `copy (List from, List to)` копіює колекцію `from` в колекцію `to`.
- Метод `fill (List coll, Object element)` замінює всі елементи даної колекції `coll` елементом `element`.

З рештою методів познайомимося по мірі необхідності.

6.24. Заключення

Таким чином, в даному уроці ми вияснили, що мова `Java` представляє багато засобів для роботи з великими об'ємами інформації. В більшості випадків достатньо додати в програму три-п'ять операторів, щоб можна було зробити нетривіальну обробку інформації.

В наступному уроці ми розглянемо аналогічні засоби для роботи з масивами, датами, для одержання випадкових чисел і інших необхідних засобів програмування.

Лабораторна робота 5. Використання класів `Vector`, `Stack` і `HashTable` для створення баз даних.

1. Користуючись класом `Vector` створіть базу даних, елементами якої є об'єкти класу `Student`, створені на попередній роботі. Передбачити необхідні методи управління базою даних і проілюструвати їх на працючій програмі.
2. Переробіть попередню програму з використанням класу `Stack`. Проілюструйте в програмі дію методів, відсутніх у класі `Vector`.
3. Переробіть програму пункту 1 з використанням класу `HashTable` використовуючи пари (Прізвище студента, об'єкт класу `Student`).

Урок 7

Класи-утиліти

- Робота с масивами
- Локальні установки
- Робота з датами і часом
- Часовий пояс і літній час
- Клас Calendar
- Підклас GregorianCalendar
- Представлення дати і часу
- Одержання випадкових чисел
- Копіювання масивів
- Взаємодія з системою

В цьому уроці описані засоби, корисні для створення програм: робота с масивами, датами, випадковими числами.

7.1. Робота з масивами

В класі `Arrays` із пакету `java.util` зібрано багато методів для роботи з масивами. Їх можна розділити на чотири групи.

Вісімнадцять статичних методів сортують масиви з різними типами числових елементів у порядку зростання чисел або просто обєкти в їх природному порядку. Вісім із них мають простий вигляд

`static void sort(type[] a)`

де `type` може бути один із семи примітивних типів `byte`, `short`, `int`, `long`, `char`, `float`, `double` або тип `Object`. Вісім методів з тими ж типами сортують частину масиву від індексу `from` включно до індексу `to` виключно:

`static void sort(type[] a, int from, int to)`

Останні два методи сортування упорядковують масив або його частину з елементами типу `Object` по правилу, заданому об'єктом `c`, реалізуючим інтерфейс `Comparator`:

`static void sort(Object[] a, Comparator c)`
`static void sort(Object[] a, int from, int to, Comparator c)`

Після сортування можна організувати бінарний пошук в масиві одним із дев'яти статичних методів пошуку. Вісім методів мають вигляд

`static int binarySearch(type[] a, type element)`

де `type` - один із тих же восьми типів. Дев'ятий метод пошуку має вигляд

`static int binarySearch(Object[] a, Object element, Comparator c)`.

Він відшукує елемент `element` в масиві, відсортованому в порядку, заданому об'єктом `c`. Методи пошуку повертають індекс знайденого елемента масиву. Якщо елемент не знайдено, то повертається відємне число, що означає індекс, з яким елемент був би вставлений в масив у заданому порядку, з протилежним знаком.

Вісімнадцять статичних методів заповнюють масив або частину масиву указаним значенням `value`:

`static void fill(type[], type value)`
`static void fill(type[], int from, int to, type value)`

де `type` - один із восьми примітивних типів або тип `Object`. Нарешті, дев'ять статичних логічних методів

порівнюють масиви:

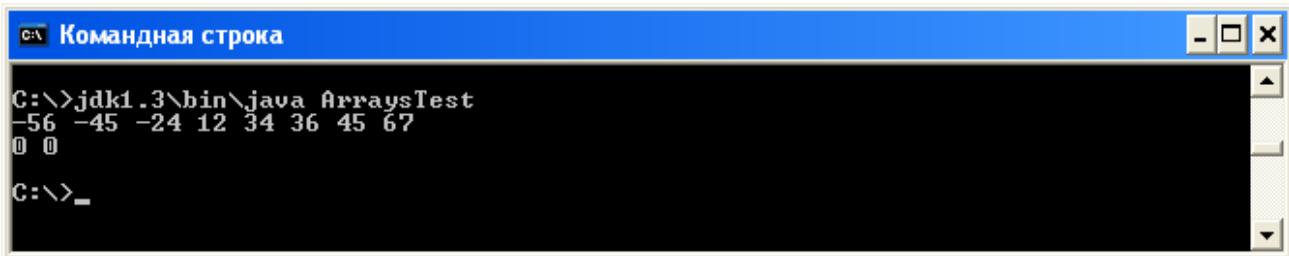
```
static boolean equals(type[] a1, type[] a2)
```

де `type` - один із восьми примітивних типів або тип `Object`.

Масиви вважаються рівними, і повертається `true`, якщо вони мають однакову довжину і рівні елементи масивів з однаковими індексами. В лістингі 7.1 приведений простий приклад роботи з цими методами.

Лістинг 7.1. Застосування методів класу `Arrays`

```
import java.util.*;
class ArraysTest{
public static void main(String[] args){
int[] a = {34, -45, 12, 67, -24, 45, 36, -56};
Arrays.sort(a) ;
for (int i = 0; i < a.length; i++)
System.out.print (a[i] + " ");
System.out.println();
Arrays.fill(a, Arrays.binarySearch(a, 12), a.length, 0);
for (int i = 6; i < a.length; i++)
System.out.print(a[i] + " ");
System.out.println();
}
}
```



7.2. Локальні установки

Деякі дані — дати, час — традиційно представляються в різних місцевостях по-різному. Наприклад, дата в Росії виводиться в форматі число, місяць, рік через точку: 27.06.01. В США прийнято запис місяць/число/рік через похилу риску: 06/27/01. Сукупність таких форматів для даної місцевості, як говорять на жаргоні "локаль", зберігається в обєкті класу `Locale` із пакету `java.util`. Для створення такого обєкта достатньо знати мову `language` і місцевість `country`. Інколи потрібна третя характеристика — варіант `variant`, що визначає програмний продукт, наприклад, `"WIN"`, `"MAC"`, `"POSIX"`. По замовчуванню місцеві установки визначаються операційною системою і читаються із системних властивостей. Подивіться на рядки :

```
user.language = ua // Мова — українська
user.region = UA // Місцевість — Україна
file.encoding = Cp1251 // Байтове кодування — CP1251
```

Вони визначають українську локаль і локальне кодування байтових символів. Локаль, установлену по замовчуванні на тій машині, де виконується програма, можна вияснити статичним методом `Locale.getDefault()`. Щоб працювати з другою локаллю, її треба спочатку створити. Для цього в класі `Locale` є два конструктори:

```
Locale(String language, String country)
Locale(String language, String country, String variant)
```

Параметр `language` — це рядок із двох літер, визначений стандартом `ISO639`, наприклад, `"ru"`, `"fr"`, `"en"`. Параметр `country` — рядок із двох заглавних літер, визначений стандартом `ISO3166`, наприклад, `"RU"`, `"US"`, `"EN"`. Параметр `variant` не визначається стандартом, це може бути, наприклад, рядок `"Traditional"`. Локаль часто указують одним рядком `"ru_RU"`, `"en_GB"`, `"en_US"`, `"en_CA"` і т. д. Після створення локалі можна зробити її локаллю по замовчуванню статичним методом:

```
Locale.setDefault(Locale newLocale);
```

Декілька статичних методів класу `Locale` дозволяють одержати параметри локалі по замовчуванні або локалі, заданої параметром `locale`:

`String getCountry()` - стандартний код країни із двох літер;
`String getDisplayCountry()` - країна записується словом, що може бути виведене на екран;
`String getDisplayCountry (Locale locale)` - те ж для указаної локалі.

Такі ж методи єсть для мови і варіанта. Можна продивитись список всіх локалів, визначених для даної `JVM`, і їх параметрів, що виводиться в стандартному вигляді:

```
Locale[] getAvailableLocales()
String[] getISOCountries()
String[] getISOLanguages()
```

Установленна локаль надалі використовується при виведенні даних в місцевому форматі.

7.3. Робота з датами і часом

Методи роботи з датами часом зібрані в два класи: `Calendar` і `Date` із пакету `java.util`. Об'єкт класу `Date` зберігає число мілісекунд, що пройшло з 1 січня 1970 г. 00:00:00 по Грінвічу. Це "день народження" `UNIX`, він називається `"Epoch"`. Клас `Date` зручно використовувати для відрахунку проміжків часу в мілісекундах. Одержані поточне число мілісекунд, що пройшли з моменту `Epoch` на тій машині, де виконується програма, можна статичним методом

```
System.currentTimeMillis()
```

В класі `Date` два конструктори. Конструктор `Date()` заносить в створюваний об'єкт поточний час машини, на якій виконується програма, по системному годиннику, а конструктор `Date(long millisec)` — указане число. Одержані значення, що зберігається в об'єкті, можна методом `long getTime()`, установити нове значення - методом `setTime(long newTime)`. Три логічних методи порівнюють час:

- `boolean after (long when)` - повертає `true`, якщо час `when` більше даного;
- `boolean before (long when)` - повертає `true`, якщо `when` менше даного;
- `boolean after (Object when)` - повертає `true`, якщо `when` - об'єкт класу `Date` і часи співпадають.

Ще два методи, порівнюючи час, повертають відємне число типу `int`, якщо даний час менше аргумента `when`; нуль, якщо часи співпадають; додатне число, якщо даний час більше аргумента `when`:

- `int compareTo(Date when);`
- `int compareTo(Object when)` — якщо `when` не відноситься до об'єктів класу `Date`, створюється виключна ситуація.

Перетворення мілісекунд, що зберігаються в об'єктах класу `Date`, в поточні час і дату виконується методами класу `calendar`.

7.3.1. Часовий пояс і літній час

Методи установки і зміни часового поясу (`time zone`), а також літнього часу `DST (Daylight Savings Time)`, зібрані в абстрактному класі `Timezone` із пакету `java.util`. В цьому ж пакеті є його реалізація — підклас `SimpleTimeZone`. В класі `SimpleTimeZone` три конструктори, але частіше всього об'єкт створюється статичним методом `getDefault()`, що повертає часовий пояс, установлений на машині, яка виконує

програму. В цих класах багато методів роботи з часовими поясами, але в більшоті випадків вимагається тільки узнати часовий пояс на машині, що виконує програму, статичним методом `getDefault()`, перевірити, чи здійснюється перехід на літній час, логічним методом `useDaylightTime()`, і установити часовий пояс методом `setDefault(TimeZone zone)`.

7.3.2. Клас Calendar

Клас `Calendar` — абстрактний, в ньому зібрані загальні властивості календарів: юліанського, григоріанського, місячного. В Java API поки що є тільки одна його реалізація — підклас `GregorianCalendar`. Оскільки `Calendar` — абстрактний клас, його екземпляри створюються чотирма статичними методами по заданій локалі і/або часовому поясі:

- `Calendar getInstance()`
- `Calendar getInstance(Locale loc)`
- `Calendar getInstance(TimeZone tz)`
- `Calendar getInstance(TimeZone tz, Locale loc)`

Для роботи з місяцями визначені цілочисельні константи від `JANUARY` до `DECEMBER`, а для роботи з днями тижня — константи від `MONDAY` до `SUNDAY`. Перший день тижня можна узнати методом `int getFirstDayOfWeek()`, а установити — методом `setFirstDayOfWeek(int day)`, наприклад:

```
setFirstDayOfWeek(Calendar.MONDAY)
```

Решта методів дозволяють проглянути часі часовий пояс або установити їх.

7.3.3. Підклас GregorianCalendar

В григоріанському календарі дві цілочисельні константи визначають ери: `BC` (before Christ) і `AD` (Anno Domini). Сім конструкторів визначають календар по часу, часовому поясу і/або локалі:

- `GregorianCalendar()`
- `GregorianCalendar(int year, int month, int date)`
- `GregorianCalendar(int year, int month, int date, int hour, int minute)`
- `GregorianCalendar(int year, int month, int date, int hour, int minute, int second)`
- `GregorianCalendar(Locale loc)`
- `GregorianCalendar(TimeZone tz)`
- `GregorianCalendar(TimeZone tz, Locale loc)`

Після створення обєкта треба визначити дату переходу з юліанського календаря на григоріанський календар методом `setGregorianChange(Date date)`. По замовчуванню це 15 жовтня 1582 г. На території Росії переход був здійснений 14 лютого 1918 р., значить, створення обєкта `greg` треба виконати так:

```
GregorianCalendar greg = new GregorianCalendar();
greg.setGregorianChange(new GregorianCalendar(1918, Calendar.FEBRUARY, 14).getTime());
```

Узнати, чи являється рік високосним в григоріанському календарі, можна логічним методом `isLeapYear()`. Метод `get(int field)` повертає елемент календаря, заданий аргументом `field`. Для цього аргумента в класі `Calendar` визначені наступні статичні цілочисельні константи: `ERA`, `WEEK_OF_YEAR`, `DAY_OF_WEEK`, `SECOND`, `YEAR`, `WEEK_OF_MONTH`, `DAY_OF_WEEK_IN_MONTH`, `MILLISECOND`, `MONTH`, `DAY_OF_YEAR`, `HOUR_OF_DAY`, `ZONE_OFFSET`, `DATE`, `DAY_OF_MONTH`, `MINUTE`, `DST_OFFSET`. Декілька методів `set()`, що використовують ці константи, устанавливають відповідні значення.

7.3.4. Представлення дати і часу

Різноманітні способи представлення дат і часу можна здійснювати методами, зібраними в абстрактний клас `DateFormat` і його підклас `SimpleDateFormat` із пакету `Java.text`. Клас `DateFormat` пропонує чотири стилі представлення дати і часу:

- стиль `SHORT` представляє дату і час в короткому числовому вигляді: 27.04.01 17:32; в локалі

США: 4/27/01 5:32 PM;

- стиль **MEDIUM** задає рік чотирма цифрами і показує секунди: 27.04.2001 17:32:45; в локалі США місяць представляється трьома буквами;
- стиль **LONG** представляє місяць словом і додає часовий пояс: 27 квітень 2001 р. 17:32:45 GMT+03.-00;
- стиль **FULL** в російській локалі такий же, як і стиль **LONG**; в локалі США додається ще день тижня.

Єсть ще стиль **DEFAULT**, співпадаючий зі стилем **MEDIUM**.

При створенні обєкта класу **SimpleDateFormat** можна задати в конструкторі шаблон, що визначає якийсь інший формат, наприклад:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy hh.iran");
System.out.println(sdf.format(new Date()));
```

Одержано виведення в такому вигляді: 27-04-2001 17.32.

В шаблоні літера **d** означає цифру дня місяця, **M** — цифру місяця, **y** — цифру року, **h** — цифру години, **m** — цифру хвилин. Решта позначень для шаблону указані в документації по класу **SimpleDateFormat**. Ці буквені позначення можна змінити за допомогою класу **DateFormatSymbols**. Не у всіх локалях можна створити об'єкт класу **SimpleDateFormat**. В таких випадках використовуються статичні методи **getInstance()** класу **DateFormat**, що повертає об'єкт класу **DateFormat**. Параметрами цих методів служать стиль представлення дати і часу і, може бути, локаль. Після створення об'єкта метод **format()** класу **DateFormat** повертає рядок з датою і часом, згідно заданому стилю. В якості аргумента задається об'єкт класу **Date**. Наприклад:

```
System.out.println("LONG: " + DateFormat.getDateInstance(
DateFormat.LONG, DateFormat.LONG).format(new Date()));
```

або

```
System.out.println("FULL: " + DateFormat.getDateInstance(
DateFormat.FULL, DateFormat.FULL, Locale.US).format(new Date()));
```

7.4. Одержання випадкових чисел

Одержані випадкове невідємне число, строго менше одиниці, типу **double** можна статичним методом **random()** із класу **java.lang.Math**. При першому зверненні до цього методу створюється генератор псевдовипадкових чисел, який використовується потім при одержанні наступних випадкових чисел. Більш серйозні дії з випадковими числами можна організувати за допомогою методів класу **Random** із пакету **java.util**. В класі два конструктори:

- Random (long seed)** — створює генератор псевдовипадкових чисел, що використовує для початку роботи число **seed**;
- Random()** — вибирає в якості початкового значення поточний час.

Створивши генератор, можна одержувати випадкові числа відповідного типу методами **nextBoolean()**, **nextDouble()**, **nextFloat()**, **nextGaussian()**, **next int()**, **nextLong()**, **nextInt(int max)** або записати зразу послідовність випадкових чисел в заздалегідь визначений масив байтів **bytes** методом **nextBytes(byte[] bytes)**. Дійсні випадкові числа рівномірно розподіляються в діапазоні від 0,0 включно до 1,0 включно. Цілі випадкові числа рівномірно розподіляються по всьому діапазону відповідного типу за одним виключенням: якщо в аргументі указано цле число **max**, то діапазон випадкових чисел буде від нуля включно до **max** включно.

7.5. Копіювання масивів

В класі **System** із пакету **java.lang** єсть статичний метод копіювання масивів, який використовує сама виконуюча система **Java**. Цей метод діє швидко і надійно, його зручно застосовувати в програмах. Синтаксис:

```
static void arraycopy(Object src, int src_ind, Object dest, int dest_ind, int count)
```

Із масиву, на який указує посилка `src`, копіюється `count` елементів, починаючи з елемента з індексом `src_ind`, в масив, на який указує посилка `dest`, починаючи з його елемента з індексом `dest_ind`. Всі індекси повинні бути задані так, щоб елементи лежали в масивах, типи масивів повинні бути сумісними, а примітивні типи зобовязані повністю співпадати. Посилки на масив не повинні бути рівні `null`. Посилки `src` і `dest` можуть співпадати, при цьому для копіювання створюється проміжний буфер. Метод можна використовувати, наприклад, для зсуву елементів в масиві. Після виконання

```
int[] arr = {5, 6, 1, 8, 9, 1, 2, 3, 4, 5, -3, -7};  
System.arraycopy(arr, 2, arr, 1, arr.length - 2);
```

одержимо `(5, 7, 8, 9, 1, 2, 3, 4, 5, -3, -7, -7)`.

7.6. Взаємодія з системою

Клас `System` дозволяє здійснювати і деяку взаємодію з системою під час виконання програми (`run time`). Але крім нього для цього є спеціальний клас `Runtime`. Клас `Runtime` містить деякі методи взаємодії з `JVM` під час виконання програми. Кожний додаток може одержати тільки один екземпляр даного класу статичним методом `getRuntime()`. Всі виклики цього методу повертають посилку на один і той же об'єкт.

- Методи `freeMemory()` і `totalMemory()` повертають об'єм вільної і всієї пам'яті, що знаходитьться в розпорядженні `JVM` для розміщення об'єктів, в байтах, у вигляді числа типу `long`. Не варто твердо спиратися на ці числа, оскільки об'єм пам'яті змінюється динамічно.
- Метод `exit(int status)` запускає процес останова `JVM` і передає операційній системі статус завершення `status`. По домовленості, ненульовий статус означає ненормальне завершення. Зручніше використовувати аналогічний метод класу `System`, який являється статичним.
- Метод `halt(int status)` здійснює негайний останов `JVM`. Він не завершує запущені процеси нормально і повинен використовуватися лише в аварійних ситуаціях.
- Метод `loadlibrary(string libName)` дозволе підвантажити динамічну бібліотеку під час виконання по її імені `libName`.
- Метод `load (string fileName)` підвантажити динамічну бібліотеку по імені файла `fileName`, в якому вона зберігається. Взагалі ж, замість цих методів зручніше використовувати статичні методи класу `System` з тими ж іменами і аргументами.
- Метод `gc()` запускає процес звільнення непотрібної оперативної пам'яті (`garbage collection`). Цей процес періодично запускається самою віртуальною машиною `Java` і виконується на фоні с невеликим пріоритетом, але можна його запустити і із програми. Знову-таки зручніше використовувати статичний `Метод System.gc()`.

Нарешті, декілька методів `exec()` запускають в окремих процесах виконувані файли. Аргументом цих методів служить командний рядок виконуваного файла. Наприклад, `Runtime.getRuntime().exec ("notepad")` запускає програму `Блокнот` на платформі `MS Windows`.

- Методи `exec()` повертають екземпляр класу `process`, що дозволяють керувати запущеним процесом. Методом `destroy()` можна зупинити процес, методом `exitValue()` одержати його код завершення.
- Метод `waitFor()` призупиняє основний підпроцес до тих пір, поки не закінчиться запущений процес.
- Три методи `getInputStream()`, `getOutputStream()` і `getErrorStream()` повертають вхідний, вихідний потік і потік помилок запущеного процесу (дивись урок 18).

Програмування у Java

Урок 8. Принципи побудови графічного інтерфейса

- Компонент і контейнер
- Ієрархія класів AWT
- Заключення

Класи використовувані в цьому уроці:

Button
Canvas
Checkbox
CheckboxMenuItem
Choice
Color
Component
Container
Frame
Graphics
Label
List
Menu
Menubar
MenuItem
PopupMenu
Scrollbar
TextArea
TextField

Для вашої зручності описання цих класів поміщено в папку [Java 8](#)

8.1. Важкі і легкі компоненти

В попередніх уроках ми писали програми, звязані з текстовим терміналом і запускали їх із командного рядка. Такі програми називаються *консольними додатками*. Вони розробляються для виконання на серверах, там, де не потрібний інтерактивний зв'язок з користувачем. Програми, тісно взаємодіючі з користувачем, сприймаючи сигнали від клавіатури і миші, працюють в графічному середовищі. Кожний додаток, призначений для роботи в графічному середовищі, повинен створити хоча б одне вікно, в якому буде виконуватися його робота, і зареєструвати його в графічній оболонці операційної системи, щоб вікно могло взаємодіяти з операційною системою і іншими вікнами: перекриватися, переміщатися, змінювати розміри, звертатися в ярлик.

Єсть багато різних графічних систем: [MS Windows](#), [X Window System](#), [Macintosh](#). В кожній з них свої правила побудови вікон і їх компонентів: меню, полів введення, кнопок, списків, смуг прокрутки. Ці правила складні і заплутані. Графічні API містять сотні функцій. Для полегшення створення вікон і їх компонентів написані бібліотеки класів: [MFC](#), [Motif](#), [OpenLook](#), [Qt](#), [Tk](#), [Xview](#), [OpenWindows](#) і багато інших. Кожний клас такої бібліотеки описує зразу цілий графічний компонент, що управляється методами цього і інших класів.

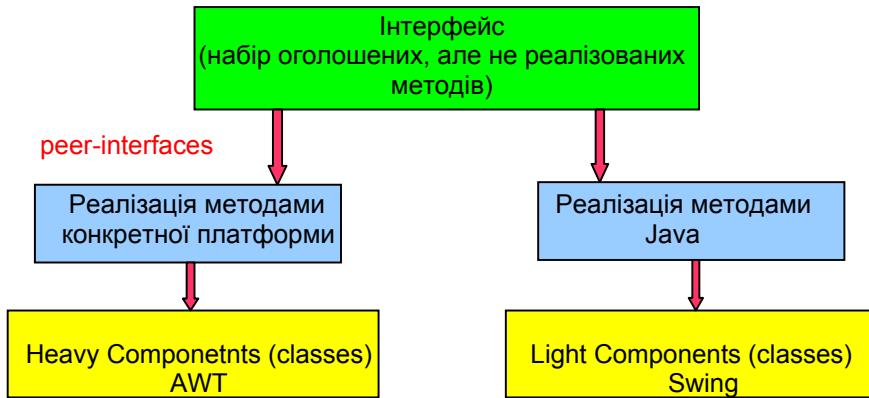
В технології Java справа ускладнюється тим, що додатки Java повинні працювати в будь-якому або хоча б в багатьох графічних середовищах. Потрібна бібліотека класів, незалежна від конкретної графічної системи. В першій версії [JDK](#) задачу вирішили наступним способом: були розроблені інтерфейси, що містили методи роботи з графічними об'єктами. Класи бібліотеки [AWT](#) реалізують ці інтерфейси для створення додатків. Додатки Java використовують дані методи для розміщення і переміщення графічних об'єктів, зміни їх розмірів, взаємодії об'єктів.

З другого боку, для роботи з екраном в конкретному графічному середовищі ці інтерфейси реалізуються в кожному такому середовищі окремо. В кожній графічній оболонці це робиться по-своєму, засобами цієї

оболонки за допомогою графічних бібліотек даної операційної системи. Такі інтерфейси були названі [peer-інтерфейсами](#). Бібліотека класів [Java](#), основаних на [peer-інтерфейсах](#), отримала назву [AWT \(Abstract Window Toolkit\)](#). При виведенні обєкта, створеного в додатку [Java](#) і основаного на [peer-інтерфейсі](#), на екран створюється парний йому ([peer-to-peer](#)) об'єкт графічної підсистеми операційної системи, котрий і відображається на екрані. Ці об'єкти тісно взаємодіють під час роботи додатку. Тому графічні об'єкти [AWT](#) в кожному графічному середовищі мають вигляд, характерний для цього середовища: в [MS Windows](#), [Motif](#), [OpenLook](#), [OpenWindows](#), скрізь вікна, створені в [AWT](#), виглядають як "рідні" вікна. Якраз із-за такої реалізації [peer-інтерфейсів](#) і інших "рідних" методів, написаних, головним чином, на мові [C++](#), приходиться для кожної платформи випускати свій варіант [JDK](#).

У версії [JDK 1.1](#) бібліотека [AWT](#) була перероблена. В неї 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](#)).

"Легкі" компоненти скрізь виглядають однаково, зберігаючи заданий при створенні вигляд ([look and feel](#)). Більше того, додаток можна розробити таким способом, щоб після його запуску можна було вибрати якийсь певний вигляд: [Motif](#), [Metal](#), [Windows 95](#) або якийсь інший, і змінити цей вигляд в будь-який момент роботи. Ця цікава особливість "легких" компонентів отримала назву [PL&F \(Pluggable Look and Feel\)](#) або [plaf](#). Була створена обширна бібліотека "легких" компонентів [Java](#), названа [Swing](#). В ній були переписані всі компоненти бібліотеки [AWT](#), так що бібліотека [Swing](#) може використовуватися самостійно, незважаючи на те, що всі класи з неї розширяють класи бібліотеки [AWT](#). Бібліотека класів [Swing](#) поставлялась як доповнення до [JDK 1.1](#).



В склад [Java 2 SDK](#) вона включена як основна графічна бібліотека класів, реалізуюча ідею ["100% Pure Java"](#), поряд з [AWT](#). В [Java 2](#) бібліотека [AWT](#) значно розширеня доданням нових засобів рисування, виведення текстів і зображень, одержавших назву [Java 2D](#), і засобів, реалізуючих переміщення тексту методом [DnD \(Drag and Drop\)](#). Крім того, в [Java 2](#) включені нові методи введення/виведення [Input Method Framework](#) і засоби звязку з додатковими пристроями введення/виведення, такими як світлове перо або клавіатура Бройля, названі [Accessibility](#). 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 займе цілу книгу, тому ми обмежимося представленням тільки основних засобів бібліотеки [AWT](#).

8.2. Компонент і контейнер

Основне поняття графічного інтерфейса користувача (ГІК) — [компонент \(component\)](#) графічної системи. В українській мові це слово підрозуміває просто складову частину, елемент чого-небудь, але в графічному інтерфейсі це поняття більш конкретне. Воно означає окремий, повністю визначений елемент, котрий можна використовувати в графічному інтерфейсі незалежно від інших елементів. Наприклад, це поле введення, кнопка, рядок меню, смуга прокрутки, радіокнопка. Саме вікно додатку — теж є компонент. Компоненти можуть бути і невидимими, наприклад, панель, об'єднує компоненти, також являється компонентом.

Ви не здивуєтесь, узявши, що в [AWT](#) компонентом вважається об'єкт класу [Component](#) або об'єкт всякого

класу, розширяючого клас **Component**. В класі **Component** зібрани загальні методи роботи з будь-яким компонентом графічного інтерфейса користувача. Цей клас — центр бібліотеки **AWT**. **Ознайомтеся з цим класом**. Кожний компонент перед виведенням на екран поміщається в **контейнер** (**container**). Контейнер "знає", як розмістити компоненти на екрані. Розуміється, в мові **Java** контейнер — це об'єкт класу **Container** або всякого його розширення. **Ознайомтеся з цим класом**. Прямий нащадок цього класу — клас **jcomponent** — вершина ієархії багатьох класів бібліотеки **Swing**.

Увага! Ви постійно будете зустрічати імена компонентів двох типів, наприклад **Button** і **jButton**. Знайте, що перший компонент важкий і для його використання треба підключати бібліотеку **AWT(import java.awt.*)**. Другий компонент легкий і для його використання треба підключати бібліотеку **Swing(import java.swing.*)**. При створенні додатків для програміста немає значення, яку бібліотеку використовувати. А от при створенні аплетів, які запускаються браузером, використовується виключно **AWT**, хоча всього можна очікувати від нових версій **Java**.

Створивши компонент — об'єкт класу **Component** або його розширення, належить додати його до попередньо створеного об'єкту класу **container** або його розширення одним із методів **add()**. Клас **Container** сам являється невидимим компонентом, він розширяє клас **Component**. Таким чином, в контейнер поряд з компонентами можна поміщати контейнери, в яких знаходяться інші компоненти, досягаючи тим самим більшої гнучкості розміщення компонентів. Основне вікно додатку, активно взаємодіюче з операційною системою, необхідно побудувати по правилах графічної системи. Воно повинне переміщатися по екрану, змінювати розміри, реагувати на дії миші і клавіатури. У вікні повинні бути, як мінімум, наступні стандартні компоненти.

- **Рядок заголовку** (**title bar**), з лівої сторони якого необхідно розмістити кнопку контекстного меню, а з правої — кнопки звертання і розвертання вікна і кнопку закриття додатку.
- **Необов'язковий рядок меню** (**menu bar**) з випадаючими пунктами меню.
- **Горизонтальна і вертикальна смуги прокрутки** (**scrollbars**).
- Вікно повинне бути обмежено **рамкою** (**border**), реагуючи на дії миші.

Вікно з цими компонентами в готовому вигляді описане в класі **Frame**. Щоб створити вікно, досить зробити свій клас розширенням класу **Frame**, як показано в лістингі 8.1. Всього вісім рядків тексту і вікно готове.

Лістинг 8.1. Найпростіше вікно додатку

```
import java.awt.*;
class TooSimpleFrame extends Frame{
    public static void main(String[] args){
        Frame fr = new TooSimpleFrame(); //Конструктор вікна
        fr.setSize(400, 150); //Методами класу установлюємо властивості об'єкту
        fr.setVisible(true);
    }
}
```



Клас **TooSimpleFrame** володіє всіма властивостями класу **Frame**, являючись його розширенням. В ньому створюється екземпляр вікна **fr**, і установлюються розміри вікна на екрані - 400x150 пікселів - методом **setSize()**. Якщо не задати розмір вікна, то на екрані з'явиться вікно мінімального розміру - тільки рядок заголовку. Звичайно, потім його можна розтянути за допомогою миші до будь-якого розміру.



Потім вікно виводиться на екран методом `setVisible(true)`. Справа в тому, що, з точки зору бібліотеки AWT, створити вікно значить виділити область оперативної пам'яті, заповнену потрібними пікселями, а вивести вміст цієї області на екран - уже інша задача, яку і вирішує метод `setVisible(true)`. Звичайно, таке вікно непридатне для роботи. Не кажучи уже про те, що у нього немає заголовка і тому вікно не можна закрити. Хоча його можна переміщати по екрану, міняти розміри, звертати на панель задач і розкривати, але команду завершення додатку ми не запрограмували. Вікно не можна закрити ані клацанням кнопкою миші по кнопці з хрестиком в правому верхньому куті вікна, ані комбінацією клавіш `<Alt>+<F4>`. Приходиться завершати роботу додатку способами операційної системи, наприклад, комбінацією клавіш `<Ctrl>+<C>`, або закриттям вікна [Командний рядок](#). В лістингі 8.2 до програми лістингу 8.1 додані заголовок вікна і звернення до методу, що дозволяє завершити додаток.

Лістинг 8.2. Просте вікно додатку

```
import java.awt.*;
import java.awt.event.*;
class SimpleFrame extends Frame{
    SimpleFrame(String s){
        super (s);
        setSize(400, 150);
        setVisible(true);
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent ev) {
                System.exit (0);
            }
        });
    }
    public static void main(String[] args){
        new SimpleFrame(" My program");
    }
}
```



Якщо ви запустите програму в середовищі [JBuilder](#), то позбавитесь від надокучливого вікна [Командна строка](#).

В програму додано конструктор класу `SimpleFrame`, що звертається до конструктора свого суперкласу `Frame`, який записує свій аргумент `s` в рядок заголовка вікна. В конструктор перенесена установка розмірів вікна, виведення його на екран і додано звернення до методу `addWindowListener()`, реагуючому на дії з вікном. В якості аргумента цьому методу передається екземпляр безіменного внутрішнього класу, розширяючого клас `WindowAdapter`. Цей безіменний клас реалізує метод `windowClosing()`, що обробляє закриття вікна.

Дана реалізація дуже проста — додаток завершується статичним методом `exit()` класу `System`. Вікно при цьому закривається автоматично. Все це ми детально розберемо в уроці 12, а поки що просто додавайте ці рядки у всі ваші програми для закріття вікна і завершення роботи додатку. Отже, вікно готове. Але воно поки що пусте. Виведемо в нього, по традиції, привітання `"Hello, World!"`, правда, злегка змінене. В лістингі 3.3 приведена повна програма цього виведення, а рис. 8.1 демонструє вікно.

Лістинг 8.3. Графічна програма з привітанням

```
import java.awt.*;
import java.awt.event.*;
class Hello extends Frame{
    Hello(String s){
        super(s);
    }
    public void paint(Graphics g){
        g.setFont(new Font("Serif", Font.ITALIC | Font.BOLD, 30));
        g.drawString("Hello, XXI century World!", 20, 100);
    }
    public static void main(String[] args){
        Frame f = new Hello("Вітаю тебе, XXI століття!");
        f.setSize(400, 150);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent ev){
                System.exit(0);
            }
        });
    }
}
```

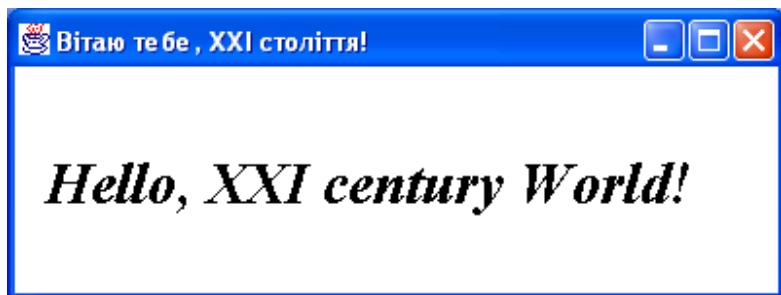


Рис. 8.1. Вікно програми-привітання

Для виведення тексту ми перевизначаєм метод `paint()` класу `Component`. Клас `Frame` наслідує цей метод з пустою реалізацією. Метод `paint()` одержує в якості аргумента екземпляр `g` класу `Graphics`, що вміє, зокрема, виводити текст методом `drawString()`. В цьому методі крім тексту ми указуємо положення початку рядка у вікні - 20 пікселів від лівого краю і 100 пікселів зверху. Ця точка - ліва нижня точка першої літери тексту `H`. Крім того, ми установили новий шрифт `"Serif"` більшого розміру - 30 пунктів, напівжирний, курсив. Всякий шрифт - об'єкт класу `Font`, а задається він методом `setFont()` класу `Graphics`. Роботу з шрифтами ми розглянемо в наступному уроці. В лістингі 8.3 ми винесли виклики методів установки розмірів вікна, виведення його на екран і завершення програми в метод `main()`.

Як бачимо із цього простого прикладу, бібліотека `AWT` велика і розгалужена, в ній багато класів, взаємодіючих один з одним. Розглянемо ієархію деяких найчастіше використовуваних класів `AWT`.

8.3. Ієархія класів AWT

На рис. 8.2 показана ієархія основних класів `AWT`. Основу її складають готові компоненти: `Button`,

[Canvas](#), [Checkbox](#), [Choice](#), [Container](#), [Label](#), [List](#), [Scrollbar](#), [TextArea](#), [TextField](#), [Menubar](#), [Menu](#), [PopupMenu](#), [MenuItem](#), [CheckboxMenuItem](#). Якщо цього набору замало, то від класу [Canvas](#) можна породити власні "важкі" компоненти, а від класу [Component](#) — "легкі" компоненти.

Основні контейнери — це класи [Panel](#), [ScrollPane](#), [Window](#), [Frame](#), [Dialog](#), [FileDialog](#). Свої "важкі" контейнери можна породити від класу [Panel](#), а "легкі" — від класу [Container](#). Цілий набір класів допомагає розміщувати компоненти, задавати колір, шрифт, рисунки і зображення, реагувати на сигнали від миші і клавіатури.

На рис. 8.2 показані і початкові класи ієрархії бібліотеки [Swing](#) — класи [JComponent](#), [JWindow](#), [JFrame](#), [JDialog](#), [JApplet](#).

Заключення

Як бачимо, бібліотека графічних класів [AWT](#) дуже велика і детально опрацьована. Ця різноманітність класів тільки відображає різноманітність задач побудови графічного інтерфейса. Бажання покращити інтерфейс безмежне. Воно приводить до створення все нових бібліотек класів і розширенню існуючих. Незалежними виробниками створено уже багато графічних бібліотек Java: [KL Group](#), [JBCL](#), і з'являються все нові і нові бібліотеки. Інформацію про них можна одержати на сайтах, указаних в уроці 1. В наступних уроках ми детально розглянемо, як можна використовувати бібліотеку [AWT](#) для створення власних додатків з графічним інтерфейсом користувача, зображеннями, анімацією і звуком.

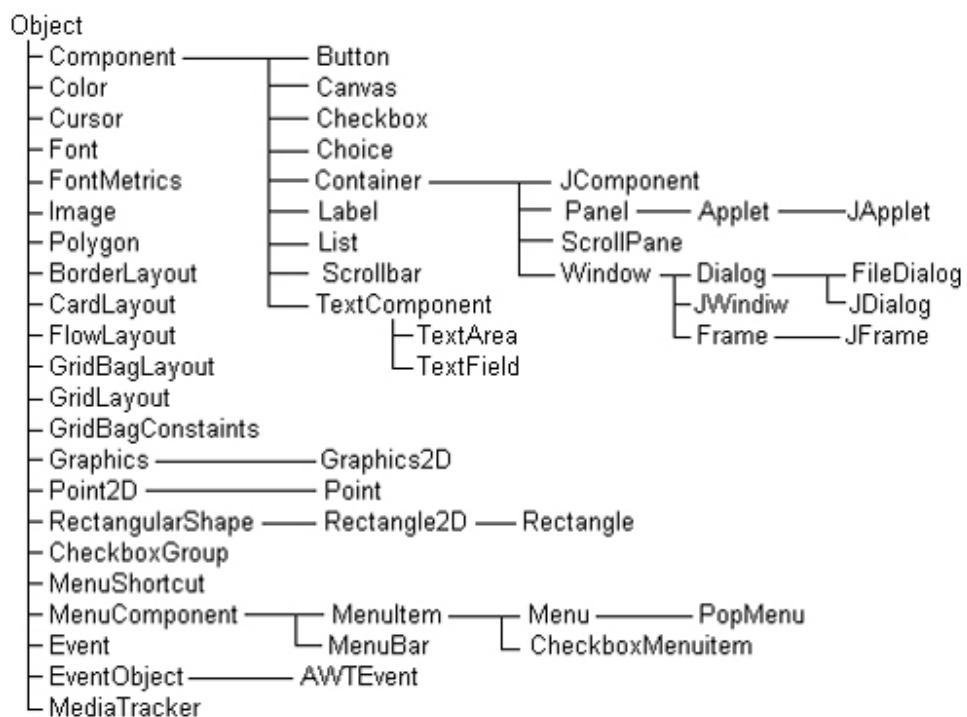


Рис. 8.2. Ієрархія основних класів [AWT](#)

Лабораторна робота 7. Робота з графічними компонентами

1. Створіть вікно більшого розміру на зразок лістингу 8.3.
2. Видаліть з нього текстовий рядок.
3. Попробуйте додати в нього послідовно компоненти `Button`, `Canvas`, `Checkbox`, `Choice`, `Container`, `Label`, `List`, `Scrollbar`, `TextArea`, `TextField`, `Menubar`, `Menu`, `PopupMenu`, `MenuItem`, `CheckboxMenuItem`, і кожного разу запускайте програму, щоб пересвідчитися який вигляд мають ці компоненти. При одночасному виведенні вони можуть накластися один на один, так що в деяких із них можете і не побачити. Давайте компонентам те ж саме імя, але з індексом,

наприклад `Button button1`. Майте на увазі, що ви працюєте з контейнером і створивши компонент він автоматично в ньому не зявиться. Перечитайте уважно початок параграфу 8.2, щоб зрозуміти, яким методом компонент додається до контейнера. Дивись зразок нижче.

```
class Hello extends Frame{
    Hello(String s){
        super(s);
        Button button1 = new Button("Button1");
        add(button1);
    }
}
```

4. Ви помічаєте, що картина дещо інша, ніж у [VB](#), [Delphi](#), [MFC](#). Границі компонентів не чіткі. Щоб їх краще розрізняти, закрашуйте в різні кольори. Колір, як і все у [Java](#), це клас. Установити колір значить створити об'єкт цього класу. Ось приклад:

```
Color cl = new Color(255,0,0);
button1.setBackground(cl);
```

Про інші конструктори класу `Color` почитайте у відповідному файлі.

5. Тепер ви бачите, що [by default](#) у [Java](#) компонент займає все вікно. Щоб покінчити з цим неподобством зразу ж після команди `super(s);` вставте `setLayout(null);` Тепер розтягніть ваші компоненти по формі методом `setBounds()` визначеним у класі `Component` і, таким чином, діючим для всіх компонентів. Зробіть так, щоб усі вказані вище компоненти були видимі одночасно. Відцентруйте написи на них. Учітесь знаходити необхідні методи у відповідних класах.
6. Поміняйте колір фону вікна з білого на якийсь інший двома методами: у методі `main()` і зовні цього методу.

Урок 9

Графічні примітиви

- [Методи класу Graphics](#)
- [Як задати колір](#)
- [Як нарисувати фігуру](#)
- [Клас Polygon](#)
- [Як вивести текст](#)
- [Як установити шрифт](#)
- [Як задати шрифт](#)
- [Клас FontMetrics](#)
- [Можливості Java 2D](#)
- [Перетворення координат](#)
- [Клас AffineTransform](#)
- [Рисування фігур засобами Java 2D](#)
- [Клас BasicStroke](#)
- [Клас GeneralPath](#)
- [Класи GradientPaint і TexturePaint](#)
- [Виведення тексту засобами Java 2D](#)
- [Методи покращення візуалізації](#)
- [Заключення](#)

Потрібні в цьому уроці нові класи ви можете знайти в папці [Java9](#).

При створенні компонента, тобто об'єкта класу [Component](#), автоматично формується його *графічний контекст* ([graphics context](#)). В контексті розміщується область рисування і виведення тексту та зображень. Контекст містить поточний і альтернативний колір рисування і колір фону — об'єкти класу [Color](#), поточний шрифт для виведення тексту — об'єкт класу [Font](#). В контексті визначена система координат, початок якої з координатами [\(0, 0\)](#) розташований у верхньому лівому куті області рисування, вісь [Ox](#) направлена вправо, вісь [Oy](#) — вниз. Точки координат знаходяться між пікселями.

Керує контекстом клас [Graphics](#) або новий клас [Graphics2D](#), введений в [Java 2](#). Оскільки графічний контекст сильно залежить від конкретної графічної платформи, ці класи зроблені абстрактними. Тому не можна безпосередньо створити екземпляри класу [Graphics](#) або [Graphics2D](#). Однакож кожна віртуальна машина [Java](#) реалізує методи цих класів, створює їх екземпляри для компонента і представляє об'єкт класу [Graphics](#) методом [getGraphics\(\)](#) класу [Component](#) або як аргумент методів [paint\(\)](#) і [update\(\)](#). Подивимось спочатку, які методи роботи з графікою і текстом представляє нам клас [Graphics](#).

9.1. Методи класу Graphics

При створенні контексту в ньому задається поточний колір для рисування, зазвичай чорний, і колір фону області рисування — білий або сірий. Змінити поточний колір можна методом [setColor\(Color newColor\)](#), аргумент [newColor](#) которого — об'єкт класу [Color](#). Узнати поточний колір можна методом [getColor\(\)](#), повертуючим об'єкт класу [Color](#).

9.1.1. Як задати колір

Колір, як і все в [Java](#), — об'єкт певного класу, а саме, класу [Color](#). Основу класу складають сім конструкторів кольору. Самий простий конструктор:

`Color(int red, int green, int blue)`

створює колір, одержаний змішуванням червоного [red](#), зеленого [green](#) і синього [blue](#). Ця кольорова модель називається [RGB](#). Кожна складова змінюється від [0](#) (відсутність складової) до [255](#) (повна інтенсивність цієї складової). Наприклад:

```
Color pureRed = new Color(255, 0, 0);
Color pureGreen = new Color(0, 255, 0);
```

визначають чистий яскраво-червоний `pureRed` і чистий яскраво-зелений `pureGreen` кольори.

В другому конструкторі інтенсивність складової можна змінювати більш гладко дійсними числами від `0.0` (відсутність складової) до `1.0` (повна інтенсивність складової):

`Color(float red, float green, float blue)`

Наприклад:

`Color someColor = new Color(0.05f, 0.4f, 0.95f);`

Третій конструктор

`Color(int rgb)`

задає всі три складові в одному цілому числі. В бітах `16—23` записується червона складова, в бітах `8—15` — зелена, а в бітах `0—7` — синя складова кольору. Наприклад:

`Color c = new Color(0xFF8F48FF);`

Тут червона складова задана з інтенсивністю `0x8F`, зелена — `0x48`, синя — `0xFF`.

Наступні три конструктори

`Color(int red, int green, int blue, int alpha)`

`Color(float red, float green, float blue, float alpha)`

`Color(int rgb, boolean hasAlpha)`

вводять четверту складову кольору, так звану "альфа", що визначає прозорість кольору. Ця складова проявляє себе при накладанні одного кольору на другий. Якщо альфа рівна `255` або `1.0`, то колір повністю непрозорий, попередній колір не просвічується крізь нього. Якщо альфа рівна `0` або `0.0`, то колір абсолютно прозорий, для кожного пікселя видно тільки попередній колір. Останній із цих конструкторів враховує складову `альфа`, що знаходиться в бітах `24—31`, якщо параметр `hasAlpha` рівний `true`. Якщо ж `hasAlpha` рівне `false`, то складова `альфа` вважається рівною `255`, незалежно від того, що записано в старших бітах параметра `rgb`. Перші три конструктори створюють непрозорий колір з `альфою`, рівною `255` або `1.0`.

Сьомий конструктор

`Color(ColorSpace cspace, float[] components, float alpha)`

дозволяє створювати колір не тільки в кольоровій моделі (`color model`) `RGB`, але і в інших моделях: `CMYK`, `HSB`, `CIEXYZ`, визначених об'єктом класу `ColorSpace`. Для створення кольору в моделі `HSB` можна скористатися статичним методом

`getHSBColor(float hue, float saturation, float brightness).`

Якщо немає необхідності ретельно підбирати кольори, то можна просто скористатися однією із тринадцяти статичних констант типу `color` класу `Color`. Наперекір стандарту "Code Conventions" вони записуються рядковими літерами: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white`, `yellow`.

Методи класу `Color` дозволяють одержати складові поточного кольору: `getRed()`, `getGreen()`, `getBlue()`, `getAlpha()`, `getRGB()`, `getColorSpace()`, `getComponents()`. Два методи створюють більш яскравий `brighter()` і більш темний `darker()` кольори в порівнянні з поточним кольором. Вони корисні, якщо треба виділити активний компонент або, навпаки, показати неактивний компонент блідніше решти компонентів. Два статичних методи повертають колір, перетворений із кольорової моделі `RGB` в `HSB` і навпаки:

`float[] RGBtoHSB(int red, int green, int blue, float[] hsb)`

```
int HSBtoRGB(int hue, int saturation, int brightness)
```

Створивши колір, можна рисувати ним в графічному контексті.

9.1.2. Як нарисувати фігуру

Основний метод рисування

```
drawLine(int x1, int y1, int x2, int y2)
```

рисує поточним кольором відрізок прямої між точками з координатами (x_1, y_1) і (x_2, y_2) . Одного цього методу достатньо, щоб нарисувати будь-яку картину по точках, рисуючи кожну точку з координатами (x, y) методом `drawLine(x, y, x, y)` і змінюючи кольори від точки до точки. Але ніхто, зрозуміло, не стане цього робити.

Інші графічні примітиви:

- `drawRect(int x, int y, int width, int height)` — рисує прямокутник зі сторонами, паралельними краям екрана, і задається координатами верхнього лівого кута (x, y) , ширину `width` пікселів і висотою `height` пікселів;
- `draw3DRect(int x, int y, int width, int height, boolean raised)` — рисує прямокутник, що ніби-то виділяється із площини рисування, якщо аргумент `raised` рівний `true`, або ніби-то вдавлений в площину, якщо аргумент `raised` рівний `false`;
- `drawOval(int x, int y, int width, int height)` — рисує овал, вписаний в прямокутник, заданий аргументами метода. Якщо `width == height`, то одержимо коло;
- `drawArc(int x, int y, int width, int height, int startAngle, int arc)` — рисує дугу овала, вписаного в прямокутник, заданий першими чотирма аргументами. Дуга має величину `arc` градусів і відраховується від кута `startAngle`. Кут відраховується в градусах від вісі Ox . Додатній кут відраховується проти годинникової стрілки, відємний — по годинниковій стрілці;
- `drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)` — рисує прямокутник із закругленими краями. Закруглення викresлюються четвертінками овалів, вписаних в прямокутники шириною `arcWidth` і висотою `arcHeight`, побудовані в кутах основного прямокутника;
- `drawPolyline(int[] xPoints, int[] yPoints, int nPoints)` — рисує ламану з вершинами в точках `xPoints[i], yPoints[i]` і числом вершин `nPoints`;
- `drawPolygon(int[] xPoints, int[] yPoints, int nPoints)` — рисує замкнуту ламану, проводячи замикаючий відрізок прямої між першою і останньою точками;
- `drawPolygon(Polygon p)` — рисує замкнуту ламану, вершини якої задані об'єктом `p` класу `Polygon`.

9.2. Клас Polygon

Цей клас призначений для роботи з многоокутником, зокрема, з трикутниками і довільними чотирикутниками. Об'єкти цього класу можна створити двома конструкторами:

- `Polygon ()` — створює пустий об'єкт;
- `Polygon(int[] xPoints, int[] yPoints, int nPoints)` — задається вершини многоокутника `(xPoints[i], yPoints[i])` і їх число `nPoints`

Після створення об'єкта в нього можна додавати вершини методом

```
addPoint(int x, int y)
```

Логічні методи `contains()` дозволяють перевірити, чи не лежить в многоокутнику задана аргументами метода точка, відрізок прямої або цілий прямокутник зі сторонами, паралельними сторонам екрана. Логічні методи `intersects()` дозволяють перевірити, чи не перетинаються з даним многоокутником відрізок прямої, заданий аргументами метода, або прямокутник зі сторонами, паралельними сторонам екрана. Методи `getBounds()` і `getBounds2D()` повертають прямокутник, цілком вміщаючий в себе даний многоокутник.

Повернемося до методів класу `Graphics`. Декілька методів викresлюють фігури, залиті поточним

кольором: `fillRect()`, `fill3DRect()`, `fillArco()`, `fillOval()`, `fillPolygon()`, `fillRoundRect()`. У них такі ж аргументи, як і у відповідних методів, викresлюючих незаповнені фігури. Наприклад, якщо ви хочете змінити колір фону області рисування, то установіть новий поточний колір і накресліть ним заповнений прямокутник величиною у всю область:

```
public void paint(Graphics g){
    Color initColor = g.getColor(); // Зберігаємо початковий колір
    g.setColor(new Color(0, 0, 255)); // Установлюємо колір фону
    // Заливаємо область рисування
    g.fillRect(0, 0, getSize().width - 1, getSize().height - 1);
    g.setColor(initColor); // Відновлюємо початковий колір
    // Подальші дії
}
```

Як бачимо, в класі `Graphics` зібрані тільки самі необхідні засоби рисування. Немає навіть методу, що задає колір фону (хоча можна задати колір фону компонента методом `setBackground()` класу `Component`). Засоби рисування, виведення тексту в область рисування і виведення зображень значно доповнені і розширені в підкласі `Graphics2D`, що входить в систему `Java 2D`. Наприклад, в ньому є метод задання кольору фону `setBackground(Color c)`. Перед тим як звернутися до класу `Graphics2D`, розглянемо засоби класу `Graphics` для виведення тексту.

9.3. Як вивести текст

Для виведення тексту в область рисування поточним ольором і шрифтом, починаючи з точки `(x, y)`, в класі `Graphics` є декілька методів:

- `drawString (String s, int x, int y)` — виводить рядок `s`;
- `drawBytes(byte[] b, int offset, int length, int x, int y)` — виводить `length` елементів масиву байтів `b`, починаючи з індексу `offset`;
- `drawChars(char[] ch, int offset, int length, int x, int y)` — виводить `length` елементів масиву символів `ch`, починаючи з індексу `offset`.

Четвертий метод виводить текст, занесений в об'єкт класу, реалізуючого інтерфейс `AttributedCharacterIterator`. Це дозволяє задавати свій шрифт для кожного символу:

`drawString(AttributedCharacterIterator iter, int x, int y)`

Точка `(x, y)` — це ліва нижня точка першої літери тексту на базовій лінії (`baseline`) виведення шрифта.

9.3.1. Як установити шрифт

Метод `setFont(Font newFont)` класу `Graphics` установлює поточний шрифт для виведення тексту. Метод `getFont()` повертає поточний шрифт. Як і все в мові `Java`, шрифт — це об'єкт класу `Font`. Подивимося, які можливості представляє цей клас.

9.3.2. Як задати шрифт

Об'єкти класу `Font` зберігають накреслення (`glyphs`) символів, що утворюють шрифт. Їх можна створити двома конструкторами:

- `Font (Map attributes)` — задає шрифт із заданими аргументом `attributes`. Ключі атрибутів і деякі їх значення задаються константами класу `TextAttribute` із пакету `java.awt.font`. Цей конструктор характерний для `Java 2D` і буде розглянутий далі в цьому уроці.
- `Font (String name, int style, int size)` — задає шрифт по імені `name`, із стилем `style` і розміром `size` типографських пунктів. Цей конструктор характерний для `JDK 1.1`, але широко використовується і в `Java 2D` в силу своєї простоти.

Типографський пункт в Россії і деяких європейських країнах рівний 0,376 мм, Точніше, 1/72 частини французького дюйма. В англо-американській системі мір пункт рівний 1/72 частини англійського дюйма,

0,351 мм. Останній пункт і використовується в комп'ютерній графіці. Ім'я шрифту name може бути рядком із фізичним іменем шрифту, наприклад, "Courier New", або один із рядків "Dialog", "DialogInput", "Monospaced", "Serif", "SansSerif", "Symbol". Це так звані *логічні імена шрифтів* (logical font names). Якщо name == null, то задається шрифт по замовчуванню. Стиль шрифту style — це одна із констант класу Font:

- **BOLD** — напівжирний;
- **ITALIC** — курсив;
- **PLAIN** — звичайний.

Напівжирний курсив (**bolditalic**) можна задати операцією побітового додавання, Font. BOLD | Font. ITALIC, як це зроблено в лістинзі 8.3. При виведенні тексту логічним іменам шрифтів і стилям співставляються *фізичні імена шрифтів* (font face name) або імена сімейств шрифтів (font name). Ці імена реальних шрифтів, наявних у графічній підсистемі операційної системи. Наприклад, логічному імені "Serif" може бути співставлено ім'я сімейства (family) шрифтів *Times New Roman*, а в комбінації зі стилями — конкретні фізичні імена *Times New Roman Bold*, *Times New Roman Italic*. Ці шрифти повинні знаходитися в складі шрифтів графічної системи тієї машини, на якій виконується додаток. Список імен доступних шрифтів можна прородивитися наступними операторами:

```
Font[] fnt = Toolkit.getGraphicsEnvironment.getAvailableFonts();
for (int i = 0; i < fnt.length; i++)
    System.out.println(fnt[i].getFontName());
```

В склад **SUN J2SDK** входить сімейство шрифтів **Lucida**. Установивши **SDK**, ви можете бути впевнені, що ці шрифти єсть у вашій системі. Таблиці співставлення логічних і фізичних імен шрифтів знаходяться у файлах з іменами

- font.properties;
- font.properties.ar;
- font.properties.ja;
- font.properties.ru.

і т. д. Ці файли повинні бути розташовані в **JDK** в каталозі **jdk1.3\jre\lib** або в якому-небудь іншому підкаталозі **lib** кореневого каталога **JDK** тієї машини, на якій виконується додаток. Потрібний файл вибирається віртуальною машиною **Java** по закінченні імені файла. Це закінчення співпадає з міжнародним кодом мови, установленого в локалі або в системній властивості **user.language** (див. рис. 6.2). Якщо у вас установлена російська локаль з міжнародним кодом мови "ru", то для співставлення буде вибраний файл **font.properties.ru**. Якщо такий файл не знайдено, то використовується файл **font.properties**, що не відповідає ніякій конкретній локалі. Тому можна залишити в системі тільки один файл **font.properties**, переписавши в нього зміст потрібного файла або створивши файл заново. Для будь-якої локалі буде використовуватися цей файл. В лістинзі 9.1 показано скорочений зміст файла **font.properties.ru** із **JDK 1.3** для платформи **MS Windows**.

Лістинг 9.1. Примірний файл **font.properties.ru** :

```
# %W% %E%
# Це просто коментарі
# AWT Font default Properties for Russian Windows
#
# Три співставлення логічному імені "Dialog":
dialog.0=Arial,RUSSIAN_CHARSET
dialog.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
# По три співставлення стилям ITALIC, BOLD, ITALIC+BOLD:
dialog.italic.0=Arial Italic,RUSSIAN_CHARSET
dialog.italic.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.italic.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bold.0=Arial Bold,RUSSIAN_CHARSET
dialog.bold.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bold.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
```

```

dialog.bolditalic.0=Arial Bold Italic,RUSSIAN_CHARSET
dialog.bolditalic.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialog.bolditalic.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
# По три співставлення імені "Dialoglnput" і стилям:
dialoginput.0=Courier New,RUSSIAN_CHARSET
dialoginput.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
dialoginput.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
dialoginput.italic.0=Courier New Italic,RUSSIAN_CHARSET
# і так далі
#
# По три співставлення імені "Serif" і стилям:
serif.0=Times New Roman,RUSSIAN_CHARSET
serif.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
serif.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
serif.italic.0=Times New Roman Italic,RUSSIAN_CHARSET
# і так далі
# Інші логічні імена
sansserif. CMArial,RUSSIAN_CHARSET
sansserif.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
sansserif.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
sansserif.italic. 0=Arial Italic,ROSSIAN_CHARSET
# і так далі
#
monospaced.0=Courier New,RUSSIAN_CHARSET
monospaced.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
monospaced.2=Symbol,SYMBOL_CHARSET,NEED_CONVERTED
monospaced.italic.0=Courier New Italic,RUSSIAN_CHARSET
# і так далі
# Default font definition
#
default.char=2751
# for backword compatibility
# Старі логічні імена версії JDK 1.0
timesroman.0=Times New Roman,RUSSIAN_CHARSET
helvetica.0=Arial,RUSSIAN_CHARSET
courier.0=Courier New,RUSSIAN_CHARSET
zapfdingbats.0=WingDings,SYMBOL_CHARSET
# font filenames for reduced initialization time
# файли зі шрифтами
filename.Arial=ARIAL.TTF
filename.Arial_Bold=ARIALBD.TTF
filename.Arial_Italic=ARIALI.TTF
filename.Arial_Bold_Italic=ARIALBI.TTF
filename.Courier_New=COUR.TTF
filename.Courier_New_Bold=COURBD.TTF
filename.Courier_New_Italic=COURI.TTF
filename.Courier_New_Bold_Italic=COURBI.TTF
filename.Times_New_Roman=TIMES.TTF
filename.Times_New_Roman_Bold=T1MESBD.TTF
filename.Times_New_Roman_Italic=TIMES3.TTF
filename.Times_New_Roman_Bold_Italic=TIMESBI.TTF
filename.WingDings=WINGDING.TTF
filename.Symbol=SYMBOL1.TTF
# name aliases
# Псевдоімми логічних імен закоментовані
# alias.timesroman=serif
# alias.helvetica=sansserif
# alias.courier=monospaced
# Static FontCharset info

```

```

#
# Класи перетворення символів у байти
fontcharset.dialog.0=sun.io.CharToByteCP1251
fontcharset.dialog.1=sun.awt.windows.CharToByteWingDings
fontcharset.dialog.2=sun.awt.CharToByteSymbol
fontcharset.dialoginput.0=sun.io.CharToByteCP1251
fontcharset.dialoginput.1=sun.awt.windows.CharToByteWingDings
fontcharset.dialoginput.2=sun.awt.CharToByteSymbol
fontcharset.serif.0=sun.io.CharToByteCP1251
fontcharset.serif.1=sun.awt.windows.CharToByteWingDings
fontcharset.serif.2=sun.awt.CharToByteSymbol
fontcharset.sansserif.0=sun.io.CharToByteCP1251
fontcharset.sansserif.1=sun.awt.windows.CharToByteWingDings
fontcharset.sansserif.2=sun.awt.CharToByteSymbol
fontcharset.monospaced.0=sun.io.CharToByteCP1251
fontcharset.monospaced.1=sun.awt.windows.CharToByteWingDings
fontcharset.monospaced.2=sun.awt.CharToByteSymbol
# Exclusion Range info
#
# Не проглядати в цьому шрифті вказані діапазони
exclusion.dialog.0=0100-0400,0460-ffff
exclusion.dialoginput.0=0100-0400, 0460-ffff
exclusion.serif.0=0100-0400,04 60-ffff
exclusion.sansserif.0=0100-0400, 0460-ffff
exclusion.monospaced.0=0100-0400,0460-ffff
# charset for text input
#
# Введені байтові символи кодуються в кириличний діапазон кодування Unicode
inputtextcharset=RUSSIAN_CHARSET

```

Більша частина цього файла зайнята співставленнями логічних і фізичних імен. Ви бачите, що під номером 0:

- логічному імені "Dialog" співставлено ім'я сімейства Arial;
- логічному імені "Dialoginput" співставлено ім'я сімейства Courier New;
- логічному імені "Serif" співставлено ім'я сімейства Times New Roman;
- логічному імені "Sansserif" співставлено ім'я сімейства Arial;
- логічному імені "Monospaced" співставлено ім'я сімейства Courier New.

Там, де указаній стиль: `dialog.italic`, `dialog.bold` і т.д., підставлений відповідний фізичний шрифт. В рядках лістинга 9.1, що починаються зі слова `filename`, указані файли з відповідними фізичними шрифтами, наприклад:

`filename.Arial=ARIAL.TTF`

Ці рядки необов'язкові, але вони прискорюють пошук файлів зі шрифтами. Тепер подивимося на останні рядки лістинга 9.1. Рядок

`exclusion.dialog.0=0100-0400, 0460-ffff`

означає, що в шрифті, співставленому логічному імені "Dialog" під номером 0, а саме, Arial, не стануть відшукуватися накреслення (glyphs) символів з кодами в діапазонах '\u0100' — '\u0400' і '\u0460' — '\uFFFF'. Вони будуть взяті із шрифту, співставленого цьому імені під номером 1, а саме, WingDings. Те ж саме буде відбуватися, якщо потрібні накреслення не знайдені в шрифті, співставленому логічному імені під номером 0. Не всі файли зі шрифтами Unicode містять накреслення (glyphs) всіх символів. Якщо потрібні накреслення не знайдені і в співставленні 1 (в даному прикладі в шрифті WingDings), вони будуть відшукуватися у співставленні 2 (тобто в шрифті Symbol) і т. д. Подібних співставлень можна написати скільки завгодно. Таким чином, кожному логічному імені шрифта можна співставити різні діапазони різних реальних шрифтів, а також застрахуватися від відсутності накреслень деяких символів в шрифтах

Unicode.

Всі співставлення під номерами 0, 1, 2, 3, 4 належить повторити для всіх стилів: **bold**, **italic**, **bolditalic**. Якщо в графічній системі використовуються шрифти **Unicode**, як, наприклад, в MS Windows NT/2000, то більше ні про що не треба турбуватися. Якщо ж графічна система використовує байтові **ASCII**-шрифти як, наприклад, **MS Windows 95/98/ME**, то треба потурбуватися про їх правильне перекодування в **Unicode** і навпаки. Для цього на платформі **MS Windows** використовуються константи **Win32 API RUSSIAN_CHARSET, SYMBOL_CHARSET, ANSI_CHARSET, OEM_CHARSET** і ін., показуючі, яку кодову таблицю використовувати при перекодуванні, так же, як це відмічалося в уроці 5 при створенні рядка із масиву байтів. Якщо логічним іменам співставлені байтові **ASCII**-шрифти (в прикладі це шрифти **WingDings** і **Symbol**), то необхідність перекодування задається константою **NEED_CONVERTED**. Перекодуванням займаються методи спеціальних класів **charToByteCP1251**, **TiarToByteWingDings**, **CharToByteSymbol**. Вони указуються для кожного співставлення імен в рядках, що починаються зі слова **fontcharset**. Ці рядки обовязкові для всіх шрифтів, помічених константою **NEED_CONVERTED**. В останньому рядку файла указана кодова сторінка для перекодування в **Unicode** символів, що вводяться в поля введення:

inputtextcharset = RUSSIAN_CHARSET

Цей запис задає кодову таблицю CP1251. Отже, збираючись виводити рядок **str** в графічний контекст методом **drawstring()**, ми створюємо поточний шрифт конструктором класу **Font**, указану в ньому логічне ім'я шрифту, наприклад, **"Serif"**. Виконуюча система Java відшукує у файле **font.properties**, відповідному локальній мові, співставленій цьому логічному імені фізичний шрифт операційної системи, наприклад, **Times New Roman**. Якщо це **Unicode**-шрифт, то із нього добуваються накреслення символів рядка **str** по їх кодуванню **Unicode** і відображаються в графічний контекст. Якщо це байтовий **ASCII**-шрифт, то рядок **str** попередньо перекодовується в масив байтів методами класу, указаного в одному із рядків **fontcharset**, наприклад, **CharToByteCP1251**. Хороші приклади файлів **font.properties.ru** зібрані на сторінці Сергія Астахова, указаній в уроці 0. Обговорення цих питань і прикладів файлів **font.properties** для **X Window System** дані в документації **SUN J2SDK** в файлі **docs/guide/intl /fontprop.html**.

Завершуючи обговорення логічних і фізичних імен шрифтів, належить сказати, що в **JDK 1.0** використовувались логічні імена **"Helvetica"**, **"TimesRoman"**, **"Courier"**, замінені в **JDK 1.1.3** **"SansSerif"**, **"Serif"**, **"Monospaced"**, відповідно, із ліцензійних міркувань. Старі імена залишилися у файлах **font.properties** для сумісності.

При виведенні рядка у вікно додатку дуже часто виникає необхідність розташувати її певним способом відносно інших елементів зображення: центрувати, вивести над або під іншим графічним об'єктом. Для цього треба знати метрику рядка: її висоту і ширину. Для вимірювання розмірів окремих символів і рядка в цілому розроблений клас **FontMetrics**. В **Java 2D** клас **FontMetrics** замінений класом **TextLayout**. Його ми розглянемо в кінці цього уроку, а зараз вияснимо, яку користь можна отримати із методів класу **FontMetrics**.

9.4. Клас **FontMetrics**

Клас **FontMetrics** являється абстрактним, тому не можна скористатися його конструктором. Для одержання об'єкта класу **FontMetrics**, що містить набір метричних характеристик шрифту **f**, треба звернутися до методу **getFontMetrics (f)** класу **Graphics** або класу **Component**. Клас **FontMetrics** дозволяє узнати ширину окремого символу **ch** в пікселях методом **charwidth(ch)**, загальну ширину всіх символів масиву або під-масиву символів або байтів методами **getChars()** і **getBytes()**, ширину цілого рядка **str** в пікселях методом **stringwidth(str)**.

Декілька методів повертають в пікселях вертикальні розміри шрифту. Інтерліньяж (**leading**) — відстань між нижньою точкою звисаючих елементів таких літер, як **p, r, u** і верхньою точкою виступаючих елементів таких літер, як **b, i, n** в наступного — повертає метод **getLeading()**. Середня відстань від базової лінії шрифту до верхньої точки прописних літер і виступаючих елементів того ж рядка (**ascent**) повертає метод **getAscent()**, а максимальне — метод **getMaxAscent()**. Середню відстань звисаючих елементів від базової лінії того ж рядка (**descent**) повертає метод **getDescent()**, а максимальну — метод **getMaxDescent()**. Нарешті, висоту шрифту (**height**) — суму **ascent + descent + leading** — повертає метод **getHeight()**. Висота шрифту рівна відстані між базовими лініями сусідніх рядків. Ці елементи показані на рис. 9.1.

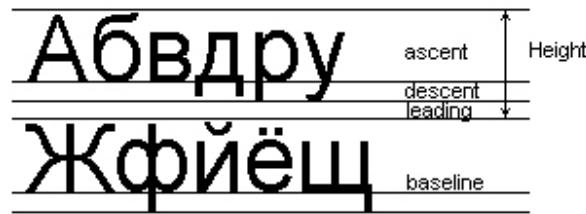


Рис. 9.1. Елементи шрифту

Додаткові характеристики шрифту можно визначити методами класу `LineMetrics` із пакету `java.awt.font`. Об'єкт цього класу можна одержати кількома методами `getLineMetrics()` класу `FontMetrics`. Лістинг 9.2 показує застосування графічних примітивів і шрифтів, а рис. 9.2 — результат виконання програми із цього лістингу.

Лістинг 9.2. Використання графічних примітивів і шрифтів

```

import java.awt.*;
import java.awt.event.*;
class GraphTest extends Frame{
GraphTest(String s){
super(s);
}
public void paint(Graphics g){
Dimension d = getSize();
int dx = d.width / 20, dy = d.height / 20;
g.drawRect(dx, dy + 20, d.width - 2*dx, d.height - 2*dy - 20);
g.drawRoundRect(2 * dx, 2*dy + 20, d.width - 4*dx, d.height - 4*dy - 20, dx, dy);
g.fillArc(d.width / 2 - dx, d.height - 2*dy + 1, 2*dx, dy - 1, 0, 360);
g.drawArc(d.width / 2 - 3*dx, d.height - 3*dy / 2 - 5, dx, dy / 2, 0, 360);
g.drawArc(d.width / 2 + 2*dx, d.height - 3*dy / 2 - 5, dx, dy / 2, 0, 360);
Font f1 = new Font("Serif", Font.BOLD|Font.ITALIC, 2 * dy);
Font f2 = new Font ("Serif", Font.BOLD, 5 * dy / 2);
FontMetrics fm1 = getFontMetrics(f1);
FontMetrics fm2 = getFontMetrics(f2);
String s1 = "Всяка останя помилка";
String s2 = "являється передостанньою.";
String s3 = "Закон налаштування програми";
int firstLine = d.height / 3;
int nextLine = fm1.getHeight();
int secondLine = firstLine + nextLine / 2;
g.setFont(f2);
g.drawString(s3, (d.width-fm2.stringWidth(s3)) / 2, firstLine);
g.drawLine(d.width / 4, secondLine - 2,3 * d.width / 4, secondLine - 2);
g.drawLine(d.width / 4, secondLine - 1, 3 * d.width / 4, secondLine - 1);
g.drawLine(d.width / 4, secondLine, 3 * d.width / 4, secondLine);
g.setFont(f1);
g.drawString(s1, (d.width - fm1.stringWidth(s1)) / 2, firstLine + 2 * nextLine);
g.drawString(s2, (d.width - fm1.stringWidth(s2)) / 2, firstLine + 3 * nextLine);
}
public static void main(String[] args){
GraphTest f = new GraphTest(" Приклад рисування");
f.setBounds(0, 0, 500, 300);
f.setVisible(true);
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
})
}

```

```
 }  
 } ) ;  
 }  
 }
```

В лістинзі 9.2 використаний простий клас `Dimension`, головне призначення якого — зберігати ширину і висоту прямокутного об'єкта в своїх полях `width` і `height`. Метод `getsize()` класу `Component` повертає розміри компонента у вигляді об'єкта класу `Dimension`. В лістинзі 9.2 розміри компонента `f` типу `GraphMest` установлені в конструкторі методом `setBounds()` рівніми 500x300 пікселів. Ще одна особливість лістингу 9.2 — для викреслювання товстої лінії, відділяючої заголовок від тексту, прийшлось провести три паралельні прямі на відстані один піксель одна від одної.

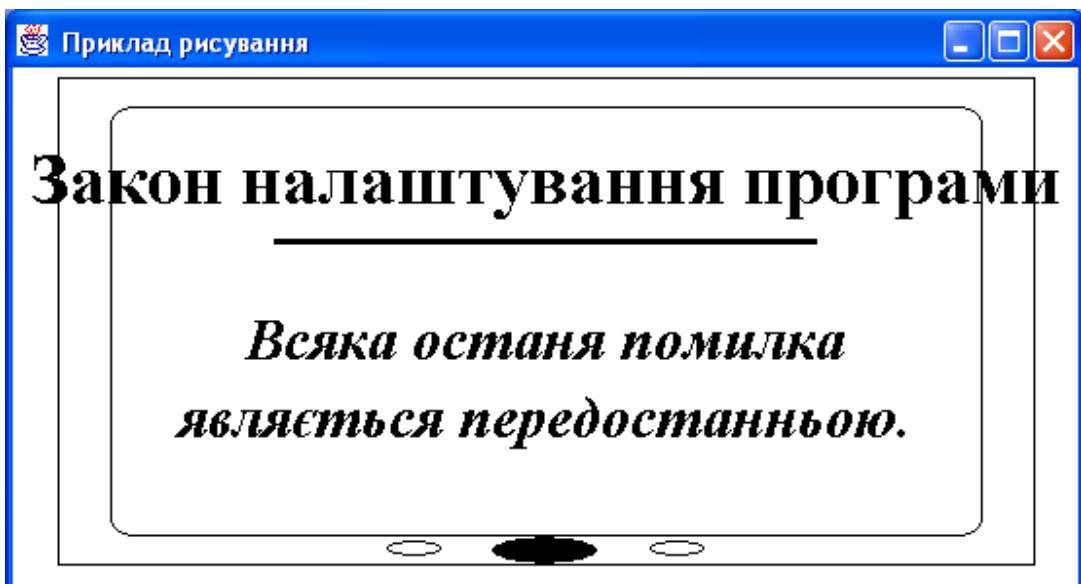


Рис. 9.2.

Приклад використання класу Graphics

Як
бачимо
із огляду
класу
[Graphics](#) і
супутніх
йому
класів,
засоби
рисуван

ня і виведення тексту в цьому класі досить обмежені. Лінії можна проводити тільки суцільні і тільки товщиною в один піксель, текст виводиться тільки горизонтально і зліва направо, не враховуються особливості пристрію виведення, наприклад, розрішмість екрана. Ці обмеження можна обійти різними хитрощами: креслити декілька паралельних ліній, притиснутих одна до одної, як в лістинзі 9.2, або вузький заповнений прямокутник, виводити текст по одній літері, одердати розрішмість екрана методом `getScreenSize()` класу `Java.awt.Toolkit` і використовувати його надалі. Але все це утруднює програмування, лишає його стрункості і природності, порушує принцип `KISS`. В `Java 2` клас `Graphics`, в рамках системи `Java 2D`, значно розширений класом `Graphics2D`.

9.5. Можливості Java 2D

В систему пакетів і класів Java 2D, основа якої— клас `Graphics2D` пакета `java.awt`, внесено декілька принципово нових положень.

- Крім координатної системи, прийнятої в класі `Graphics` і названої координатним простором користувача (`User Space`), введена ще система координат пристрою виведення (`Device Space`): екрана монітора, принтера. Методи класу `Graphics2D` автоматично переводять (`transform`) систему координат користувача в систему координат пристрою при виведенні графіки.
 - Перетворення координат користувача в координати пристрою можна задати "вручну", причому перетворенням може служити будь-яке афінне перетворення площини, зокрема, поворот на довільний кут і/або стиснення/розтягнення. Воно визначається як об'єкт класу `AffineTransform`. Його можна установити як перетворення по замовчуванню методом `setTransform()`. Можливо виконати перетворення "на льоту" методами `transform()` і `translate()` і робити композицію перетворень методом `concatenate()`.
 - Оскільки афінне перетворення дійсне, координати задаються дійсними, а не ціліми числами.
 - Графічні примітиви: прямоугольник, овал, дуга і ін., реалізують тепер новий інтерфейс `shape` пакету

java.awt. Для їх викреслювання можна використовувати новий єдиний для всіх фігур метод `draw()`, аргументом якого може бути будь-який об'єкт, реалізувавши інтерфейс `shape`. Введено метод `fill()`, заповнюючий фігури — об'єкти класу, реалізувавшого інтерфейс `shape`.

- Для викреслювання (`stroke`) ліній введено поняття пера (`pen`). Властивості пера описує інтерфейс `stroke`. Клас `BasicStroke` реалізує цей інтерфейс. Перо володіє чотирма характеристиками:
 - воно має толщину (`width`) в один (по замовчуванню) або декілька пікселів;
 - воно може закінчити лінію (`end cap`) закругленням — статична константа `CAP_ROUND`, прямим обрізом — `CAP_SQUARE` (по замовчуванню), або не фіксувати певний спосіб закінчення — `CAP_BUTT`;
 - воно може спрягати лінії (`line joins`) закругленням — статична константа `JOIN_ROUND`, відрізком прямої — `JOIN_BEVEL`, або просто зістикувати — `JOIN_MITER` (по замовчуванню);
 - воно може креслити лінію різними пунктирами (`dash`) і штрих-пунктирами, довжини штрихів і проміжків задаються в масиві, елементи масиву з парними індексами задають довжину штриха, з непарними індексами — довжину проміжку між штрихами.
- Методи заповнення фігур описані в інтерфейсі `Paint`. Три класи реалізують цей інтерфейс. Клас `color` реалізує його суцільною (`solid`) заливкою, клас `GradientPaint` — градієнтним (`gradient`) заповненням, при якому колір плавно змінюється від однієї заданої точки до другої заданої точки, клас `TexturePaint` — заповненням по попередньо заданному зразку (`pattern fill`).
- Літери тексту розуміються як фігури, тобто об'єкти, реалізуючі інтерфейс `shape`, і можуть викреслюватися методом `draw()` з використанням всіх можливостей цього методу. При їх викреслюванні використовується перо, всі методи заповнення і перетворення.
- Крім імені, стилю і розміру, шрифт отримав багато додаткових атрибутів, наприклад, перетворення координат, підкреслювання або закреслювання тексту, виведення тексту справа наліво. Колір тексту і його фону являються тепер атрибутами самого тексту, а не графічного контексту. Можна задати різну ширину символів шрифту, надрядкові і підрядкові індекси. Атрибути уstanовлюються константами класу `TextAttribute`.
- Процес візуалізації (`rendering`) регулюється правилами (`hints`), визначеними константами класу `RenderingHints`.

З такими можливостями **Java 2D** стала повноцінною системою рисування, виведення тексту і зображень. Подивимося, як реалізовані ці можливості, і як ними можна скористатися.

9.6. Перетворення координат

Правило перетворення координат користувача в координати графічного пристрою (`transform`) задається автоматично при створенні графічного контексту так же, як колір і шрифт. Надалі його можна змінити методом `setTransform()` так же, як змінюються колір або шрифт. Аргументом цього методу служить об'єкт класу `AffineTransform` із пакету `java.awt.geom`, подібно об'єктам класу `Color` або `Font` при заданні кольору або шрифту. Прозглянемо детальніше клас `AffineTransform`.

Афінне перетворення координат задається двома основними конструкторами класу `AffineTransform`:

`AffineTransform(double a, double b, double c, double d, double e, double f)`
`AffineTransform (float a, float b, float c, float d, float e, float f)`

При цьому точка з координатами (x, y) в просторі користувача перейде в точку з координатами ($a*x+c*y+e, b*x+d*y+f$) в просторі графічного пристрою. Таке перетворення не викривляє площину — прямі лінії переходять в прямі, кути між лініями зберігаються. Прикладами афінних перетворень служать повороти навколо будь-якої точки на будь-який кут, паралельні зсуви, відображення від осей, стиснення і розтягнення по вісям.

Наступні два конструктори використовують в якості аргумента масив `{a, b, c, d, e, f}` або `{a, b, c, d}`, якщо $e = f = 0$, складений із таких же коефіцієнтів в тому ж порядку:

`AffineTransform(double[] arr)`
`AffineTransform(float[] arr)`

П'ятий конструктор створює новий об'єкт по іншому, уже наявному, об'єкту:

[AffineTransform\(AffineTransform at\)](#)

Шостий конструктор — конструктор по замовчуванню — створює тотожне перетворення:

[AffineTransform \(\)](#)

Ці конструктори математично точні, але незручні при заданні конкретних перетворень. Попробуйте розрахувати коефіцієнти повороту на 57° навколо точки з координатами (20, 40) або догадатися, як буде перетворено простір користувача після виконання методів:

```
AffineTransform at = new AffineTransform(-1.5, 4.45, -0.56, 34.7, 2.68, 0.01);
g.setTransform(at);
```

В багатьох випадках зручніше створювати перетворення статичними методами, повертаючими об'єкт класу [AffineTransform](#).

[getRotateInstance \(double angle\)](#) — повертає поворот на кут [angle](#), заданий в радіанах, навколо початку координат. Додатній напрям повороту такий, що точки від Ох повертуються в напряму до від Оу. Якщо від Ох користувача не змінювалися перетворенням відображення, то додатнє значення [angle](#) задає поворот по годинникової стрілці.

- [getRotateInstance\(double angle, double x, double y\)](#) — такий же поворот навколо точки з координатами (x, y).
- [getScaleInstance \(double sx, double sy\)](#) — змінює маштаб по від Ох в sx раз, по від Оу — в sy раз.
- [getShearInstance \(double shx, double shy\)](#) — перетворює кожну точку (x, y) в точку (x + shx*y, y + shy*x).
- [getTranslateInstance \(double tx, double ty\)](#) — здигає кожну точку (x, y) в точку (x + tx, y + ty).

Метод [createInverse\(\)](#) повертає перетворення, обернене поточному перетворенню. Після створення перетворення його можна змінювати методами:

- [setTransform\(AffineTransform at\)](#)
- [setTransform\(double a, double b, double c, double d, double e, double f\)](#)
- [setTolIdentity\(\)](#)
- [setToRotation\(double angle\)](#)
- [setToRotation\(double angle, double x, double y\)](#)
- [setToScale\(double sx, double sy\)](#)
- [setToShear\(double shx, double shy\)](#)
- [setToTranslate\(double tx, double ty\)](#)

зробивши поточним перетворення, задане одним із цих методів. Перетворення, задані методами:

- [concatenate\(AffineTransform at\)](#)
- [rotate\(double angle\)](#)
- [rotate\(double angle, double x, double y\)](#)
- [scale\(double sx, double sy\)](#)
- [shear\(double shx, double shy\)](#)
- [translate\(double tx, double ty\)](#)

виконується перед поточним перетворенням, створюючи композицію перетворень.

Перетворення, задане методом [preconcatenate\(AffineTransform at\)](#), навпаки, здійснюються після поточного перетворення. Інші методи класу [AffineTransform](#) виконують перетворення різних фігур в просторі користувача. Пора привести приклад. Додамо в початок методу [paint\(\)](#) в лістингі 9.2 чотири оператори, як записано в лістингі 9.3.

[Лістинг 9.3. Перетворення простору користувача](#)

```
import java.awt.*;
```

```

import java.awt.event.*;
import java.awt.geom.*;
class GraphTest extends Frame{
GraphTest(String s){
super(s);
}
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
AffineTransform at = AffineTransform.getRotateInstance(-Math.PI/4.0, 250.0,150.0);
at.concatenate(new AffineTransform(0.5, 0.0, 0.0, 0.5, 100.0, 60.0));
g.setTransform(at);
Dimension d = getSize();
int dx = d.width / 20, dy = d.height / 20;
g.drawRect(dx, dy + 20, d.width - 2*dx, d.height - 2*dy - 20);
g.drawRoundRect(2 * dx, 2*dy + 20, d.width - 4*dx, d.height - 4*dy - 20, dx, dy);
g.fillArc(d.width / 2 - dx, d.height - 2*dy + 1, 2*dx, dy - 1, 0, 360);
g.drawArc(d.width / 2 - 3*dx, d.height - 3*dy / 2 - 5, dx, dy / 2, 0, 360);
g.drawArc(d.width / 2 + 2*dx, d.height - 3*dy / 2 - 5, dx, dy / 2, 0, 360);
Font f1 = new Font("Serif", Font.BOLD|Font.ITALIC, 2 * dy);
Font f2 = new Font ("Serif", Font.BOLD, 5 * dy / 2);
FontMetrics fm1 = getFontMetrics(f1);
FontMetrics fm2 = getFontMetrics(f2);
String s1 = "Всяка останя помилка";
String s2 = "являється передостанньою.";
String s3 = "Закон налаштування програми";
int firstLine = d.height / 3;
int nextLine = fm1.getHeight();
int secondLine = firstLine + nextLine / 2;
g.setFont(f2);
g.drawString(s3, (d.width-fm2.stringWidth(s3)) / 2, firstLine);
g.drawLine(d.width / 4, secondLine - 2,3 * d.width / 4, secondLine - 2);
g.drawLine(d.width / 4, secondLine - 1, 3 * d.width / 4, secondLine - 1);
g.drawLine(d.width / 4, secondLine, 3 * d.width / 4, secondLine);
g.setFont(f1);
g.drawString(s1, (d.width - fm1.stringWidth(s1)) /2, firstLine + 2 * nextLine);
g.drawString(s2, (d.width - fm1.stringWidth(s2)) / 2, firstLine + 3 * nextLine);
}
public static void main(String[] args){
GraphTest f = new GraphTest(" Приклад рисування");
f.setBounds(0, 0, 500, 300);
f.setVisible(true);
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}

```

Метод `paint()` починається з отримання екземпляру `g` класу `Graphics2D` простим приведенням аргументу `gr` до типу `Graphics2D`. Потім, методом `getRotateInstance()` визначаються поворот на 45° проти годинникової стрілки навколо точки (250.0, 150.0). Це перетворення — екземпляр `at` класу `AffineTransform`. Метод `concatenate()`, виконуваний об'єктом `at`, додає до цього перетворення стиснення в два рази по обом вісям координат і перенос початку координат в точку (100.0, 60.0). Нарешті, композиція цих перетворень установлюється як поточне перетворення об'єкта `g` методом `setTransform()`. Перетворення виконується в наступному порядку. Спочатку простір користувача стискається в два рази відповідно до обох вісей, потім початок координат користувача — лівий верхній кут — переноситься в точку (100.0, 60.0) простору графічного пристроя. Потім картинка повертається на кут 45° проти годинникової стрілки навколо точки (250.0, 150.0). Результат цих перетворень показаний на рис. 9.3.



Рис. 9.3. Перетворення координат

9.7. Рисування фігур засобами Java2D

Характеристики пера для рисування фігур описані в інтерфейсі `stroke`. В Java 2D єсти поки що тільки один клас, реалізуючий цей інтерфейс — клас `BasicStroke`.

9.7.1. Клас `BasicStroke`

Конструктори класу `BasicStroke` визначають характеристики пера. Основний конструктор

`BasicStroke(float width, int cap, int join, float miter, float[] dash, float dashBegin)`

задає:

- товщину пера `width` в пікселях;
- оформлення кінця лінії `cap`; це одна із констант:
 - `CAP_ROUND` — закруглений кінець лінії;
 - `CAP_SQUARE` — квадратний кінець лінії;
 - `CAP_BUTT` — оформлення відсутнє;
- спосіб спряження ліній `join`; це одна із констант:
 - `JOIN_ROUND` — лінії спрягаються дугою кола;
 - `JOIN_BEVEL` — лінії спрягаються відрізком прямої, перпендикулярним бісектрисі кута між лініями;
 - `JOIN_MITER` — лінії просто стикуються;
- відстань між лініями `miter`, починаючи з якої застосовується спряження `JOIN_MITER`;
- довжину штрихів і проміжків між штрихами — масив `dash`; елементи масиву з парними індексами задають довжину штриха в пікселях, елементи з непарними індексами — довжину проміжка; масив перебирається циклічно;
- індекс `dashBegin`, починаючи з якого перебираються елементи масиву `dash`.

Решта конструкторів задають деякі характеристики по замовчуванню:

- `BasicStroke (float width, int cap, int join, float miter)` — суцільна лінія;
- `BasicStroke (float width, int cap, int join)` — суцільна лінія із спряженням `JOIN_ROUND` або `JOIN_BEVEL`; для спряження `JOIN_MITER` задається значення `miter = 10.0f`;

- `BasicStroke (float width)` — прямий обріз `CAP_SQUARE` і спряження `JOIN_MITER` із значенням `miter = 10.0f`;
- `BasicStroke ()` — ширина `1.0f`.

Краще один раз побачити, ніж сто раз прочитати. В лістингі 9.4 визначено п'ять пер з різними характеристиками, рис. 9.4 показує, як вони рисують.

Лістинг 9.4. Визначення пер

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
class StrokeTest extends Frame{
StrokeTest(String s) {
super (s) ;
setSize(500, 400);
setVisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
g.setFont(new Font("Serif", Font.PLAIN, 15));
BasicStroke pen1 = new BasicStroke(20,
BasicStroke.CAP_BUTT,BasicStroke.JOIN_MITER,30);
BasicStroke pen2 = new BasicStroke(20, BasicStroke.CAP_ROUND,
BasicStroke.JOIN_ROUND);
BasicStroke pen3 = new BasicStroke(20, BasicStroke.CAP_SQUARE,
BasicStroke.JOIN_BEVEL);
float[] dash1 = {5, 20};
BasicStroke pen4 = new BasicStroke(10,
BasicStroke.CAP_ROUND,BasicStroke.JOIN_BEVEL, 10, dash1, 0);
float[] dash2 = {10, 5, 5, 5};
BasicStroke pen5 = new BasicStroke(10, BasicStroke.CAP_BUTT,
BasicStroke.JOIN_BEVEL, 10, dash2, 0);
g.setStroke(pen1);
g.draw(new Rectangle2D.Double(50, 50, 50, 50));
g.draw(new Line2D.Double(50, 180, 150, 180));
g.setStroke(pen2);
g.draw(new Rectangle2D.Double(200, 50, 50, 50));
g.draw(new Line2D.Double(50, 230, 150, 230));
g.setStroke(pen3);
g.draw(new Rectangle2D.Double(350, 50, 50, 50));
g.draw(new Line2D.Double(50, 280, 150, 280));
g.drawString("JOIN_MITER", 40, 130);
g.drawString("JOIN_ROUND", 180, 130);
g.drawString("JOINBEVEL", 330, 130);
g.drawString("CAP_BUTT", 170, 190);
g.drawString("CAP_ROUND", 170, 240);
g.drawString("CAP_SQUARE", 170, 290);
g.setStroke(pen5);
g.draw(new Line2D.Double(50, 330, 250, 330));
g.setStroke(pen4);
g.draw(new Line2D.Double(50, 360, 250, 360));
g.drawString("{10, 5, 5, 5,...}", 260, 335);
g.drawString("(5, 10,...)", 260, 365);
```

```

}
public static void main(String[] args) {
new StrokeTest("Мої пера");
}
}

```

Після створення пера одним із конструкторів і установки пера методом `setStroke()` можна рисувати різні фігури методами `draw()` і `fill()`. Загальні властивості фігур, які можна нарисувати методом `draw()` класу `Graphics2D`, описані в інтерфейсі `shape`. Цей інтерфейс реалізований для створення звичайного набору фігур — прямокутників, прямих, эліпсів, дуг, точок — класами `Rectangle2D`, `RoundRectangle2D`, `Line2D`, `Ellipse2D`, `Arc2D`, `Point2D` пакету `java.awt.geom`. В цьому пакеті є ще класи `CubicCurve2D` і `QuadCurve2D` для створення кривих третього і другого порядку. Всі ці класи абстрактні, але існують їх реалізації — вкладені класи `Double` і `Float` для задання координат числами відповідного типу. В Лістингі 9.4 використані класи `Rectangle2D.Double` і `Line2D.Double` для викреслювання прямокутників і відрізків.

9.7.2. Клас GeneralPath

В пакеті `java.awt.geom` є ще один цікавий клас — `GeneralPath`. Об'єкти цього класу можуть містити складні конструкції, складені із відрізків прямих або кривих ліній і інших фігур, або не зєднаних між собою. Більше того, оскільки цей клас реалізує інтерфейс `shape`, його екземпляри самі являються фігурами і можуть бути елементами інших об'єктів класу `GeneralPath`.

Спочатку створюється пустий об'єкт класу `GeneralPath` конструктором по замовчуванню `GeneralPath()` або об'єкт, що містить одну фігуру, конструктором `GeneralPath(Shape sh)`. Потім до цього об'єкта додаються фігури методом `append(Shape sh, boolean connect)`. Якщо параметр `connect` рівний `true`, то нова фігура зєднується з попередніми фігурами за допомогою поточного пера. В об'єкті є поточна точка. Спочатку її координати `(0, 0)`, потім її можна перемістити в точку `(x, y)` методом `moveTo(float x, float y)`.

Від поточної точки до точки `(x, y)` можна провести:

- відрізок прямої методом `lineTo(float x, float y)`;
- відрізок квадратичної кривої методом `quadTo(float x1, float y1, float x, float y)`;
- криву Безье методом `curveTo(float x1, float y1, float x2, float y2, float x, float y)`.

Поточною точкою після цього становиться точка `(x, y)`. Початкову і кінцеву точки можна зєднати методом `closePath()`. Ось як можна створити трикутник із заданими вершинами:

```

GeneralPath p = new GeneralPath();
p.moveTo(x1, y1); // Переносимо поточну точку в першу вершину,
p.lineTo(x2, y2); // проводимо сторону трикутника до другої вершини,
p.lineTo(x3, y3); // проводимо другу сторону,
p.closePath(); // проводимо третю сторону до першої вершини

```

Способи заповнення фігур визначені в інтерфейсі `Paint`. Зараз `Java 2D` містить три реалізації цього інтерфейса — класи `Color`, `GradientPaint` і `TexturePaint`. Клас `Color` нам відомий, подивимось, які способи заливки пропонують класи `GradientPaint` і `TexturePaint`.

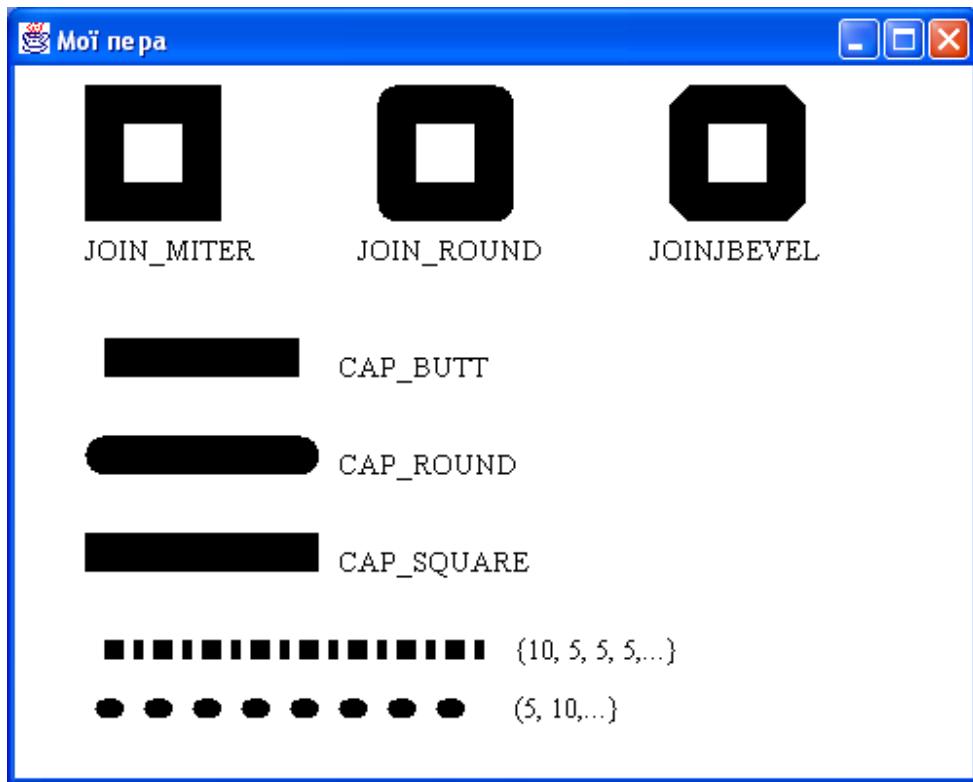


Рис. 9.4. Пера з різними характеристиками

9.7.3. Класи GradientPaint і TexturePaint

Клас [GradientPaint](#) пропонує зробити заливку наступним чином. В двох точках [M](#) і [N](#) уstanавлюються різні кольори. В точці [M\(x1, y1\)](#) задається колір [c1](#), в точці [N\(x2, y2\)](#) — колір [c2](#). Колір заливки гладко змінюється від [c1](#) до [c2](#) впідовж прямої, що зєднує точки [M](#) і [N](#), залишаючись постійним впідовж кожної прямої, перпендикулярної прямій [MN](#). Таку заливку створює конструктор

`GradientPaint(float x1, float y1, Color c1, float x2, float y2, Color c2)`

При цьому зовні відрізка [MN](#) колір залишається постійним: за точкою [M](#) — колір [c1](#), за точкою [N](#) — колір [c2](#).

Другий конструктор

`GradientPaint(float xl, float yl, Color cl, float x2, float y2, Color c2, boolean cyclic)`

при заданні параметра `cyclic == true` повторює заливку смуги [MN](#) у всій фігури. Ще два конструктори задають точки як об'єкти класу [Point2D](#).

Клас [TexturePaint](#) поступє складніше. Спочатку створюється буфер — об'єкт класу [BufferedImage](#) із пакету [java.awt.image](#). Це великий складний клас. Ми з ним ще зустрінемося в уроці 15, а поки що нам знадобиться тільки його графічний контекст, що управляється екземпляром класу [Graphics2D](#). Цей екземпляр можна отримати методом `createGraphics()` класу [BufferedImage](#). Графічний контекст буфера заповнюється фігурою, яка буде служити зразком заповнення. Потім по буферу створюється об'єкт класу [TexturePaint](#). При цьому ще задається прямокутник, розміри якого будуть розмірами зразка заповнення. Конструктор виглядає так:

`TexturePaint(BufferedImage buffer, Rectangle2D anchor)`

Після створення заливки — об'єкта класу [Color](#), [GradientPaint](#) або [TexturePaint](#) — вона уstanовлюється в

графічному контексті методом `setPaint (Paint p)` і використовується надалі методом `fill (Shape sh)`. Все це демонструє лістинг 9.5 і рис. 9.5.

Лістинг 9.5. Способи заливки

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.awt.event.*;
class PaintTest extends Frame{ PaintTest(String s){ super(s) ;
setSize(300, 300);
setVisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){ System.exit(0); } });
}
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
BufferedImage bi = new BufferedImage(20, 20, BufferedImage.TYPE_INT_RGB);
Graphics2D big = bi.createGraphics();
big.draw(new Line2D.Double(0.0, 0.0, 10.0, 10.0));
big.draw(new Line2D.Double(0.0, 10.0, 10.0, 0.0));
TexturePaint tp = new TexturePaint(bi,
new Rectangle2D.Double(0.0, 0.0, 10.0, 10.0));
g.setPaint(tp);
g.fill(new Rectangle2D.Double (50, 50, 200, 200));
GradientPaint gp = new GradientPaint(100, 100, Color.white,
150, 150, Color.black, true); g.setPaint(gp);
g.fill(new Ellipse2D.Double (100, 100, 200, 200));
}
public static void main(String[] args){
new PaintTest(" Способи заливки");
}
}
```

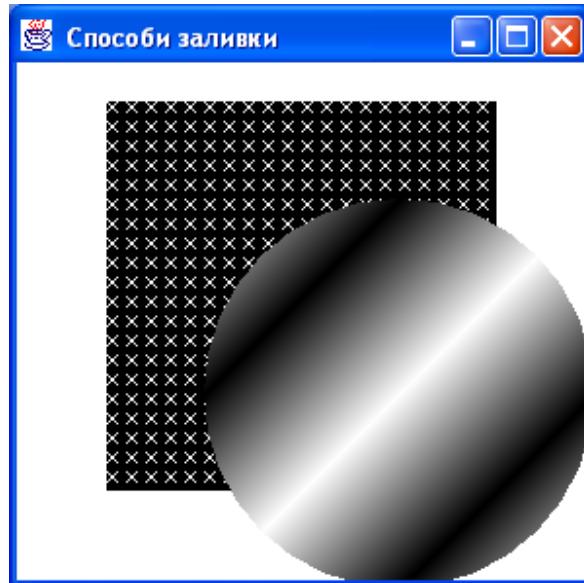


Рис. 9.5. Способи заливки

9.8. Виведення тексту засобами Java 2D

Шрифт — об'єкт класу `Font` — крім імені, стилю і розміру має ще півтора десятка атрибутів: підкреслювання, закреслювання, нахили, колір шрифту і колір фону, ширину товщину символів, аффінне перетворення, розташування зліва направо або справа наліво. Атрибути шрифту задаються як статичні константи класу `TextAttribute`. Найбільш використовувані атрибути перечислені в табл. 9.1.

Таблиця 9.1. Атрибути шрифту

Атрибут	Значення
BACKGROUND	Колір фону. Об'єкт, реалізуючий інтерфейс <code>Paint</code>
FOREGROUND	Колір тексту. Об'єкт, реалізуючий інтерфейс <code>Paint</code>
BIDI EMBEDDED	Рівень вкладеності перегляду тексту. Ціле від 1 до 1 5
CHAR_REPLACEMENT	Фігура, що заміняє символ. Об'єкт <code>GraphicAttribute</code>
FAMILY	Сімейство шрифта. Рядок типу <code>string</code>
FONT	Шрифт. Об'єкт класу <code>Font</code>
JUSTIFICATION	Допуск при вирівнюванні абзаца. Об'єкт класу <code>Float</code> із значеннями від 0,0 до 1,0. Єсть дві константи: <code>JUSTIFICATION FULL</code> і <code>JUSTIFICATION NONE</code>
POSTURE	Нахил шрифту. Об'єкт класу <code>Float</code> . Єсть дві константи: <code>POSTURE_OBLIQUE</code> і <code>POSTURE_REGULAR</code>
RUN_DIRECTION	Перегляд тексту: <code>RUN DIRECTION LTR</code> — зліва направо, <code>RUN DIRECTION RTL</code> — справа наліво
SIZE	Розмір шрифта в пунктах. Об'єкт класу <code>Float</code>
STRIKETHROUGH	Перекреслювання шрифта. Задається константою <code>STRIKETHROUGH ON</code> , по замовчуванню перекреслювання немає
SUPERSCRIPT	Нижні або верхні індекси. Константи: <code>SUPERSCRIPT_NONE</code> , <code>SUPERSCRIPT_SUB</code> , <code>SUPERSCRIPT_SUPER</code>
SWAP_COLORS	Переміна місцями кольору тексті і кольору фону. Константа <code>SWAP_COLORS ON</code> , по замовчуванню заміни немає
TRANSFORM	Перетворення шрифту. Об'єкт класу <code>AffineTransform</code>
UNDERLINE	Підкреслювання шрифту. Константи: <code>UNDERLINE_ON</code> , <code>UNDERLINE_LOW_DASHED</code> , <code>UNDERLINE_LOW_DOTTED</code> , <code>UNDERLINE_LOW_GRAY</code> , <code>UNDERLINE_LOW_ONE_PIXEL</code> , <code>UNDERLINE_LOW_TWO_PIXEL</code>
WEIGHT	Товщина шрифту. Константи: <code>WEIGHT_ULTRA_LIGHT</code> , <code>WEIGHT_EXTRA_LIGHT</code> , <code>WEIGHT_LIGHT</code> , <code>WEIGHT_DEMILIGHT</code> , <code>WEIGHT_REGULAR</code> , <code>WEIGHT_SEMIBOLD</code> , <code>WEIGHT_MEDIUM</code> , <code>WEIGHT_DEMIBOLD</code> , <code>WEIGHT_BOLD</code> , <code>WEIGHT_HEAVY</code> , <code>WEIGHT_EXTRABOLD</code> , <code>WEIGHT_ULTRABOLD</code>

WIDTH	Ширина шрифту. Константи: WIDTH_CONDENSED, WIDTH_SEMI_CONDENSED, WIDTH_REGULAR, WIDTH_SEMI_EXTENDED, WIDTH_EXTENDED
-------	---

На жаль, не всі шрифти дозволяють задавати всі атрибути. Побачити список допустимих атрибутів для даного шрифта можна методом `getAvailableAttributes()` класу `Font`. В класі `Font` є конструктор `Font(Map attributes)`, яким можна зразу задати потрібні атрибути створюваному шрифту. Це вимагає попереднього запису атрибутів у спеціально створений для цього об'єкту класу, реалізуючий інтерфейс `Map`: класу `HashMap`, `WeakHashMap` або `Hashtable` (див. урок 7). Наприклад:

```
HashMap hm = new HashMap ();
hm.put(TextAttribute.SIZE, new Float(60.0f));
hm.put(TextAttribute.POSTURE, TextAttribute.POSTURE_JDBLIQUE);
Font f = new Font(hm);
```

Можна створити шрифт і другим конструктором, яким ми користувалися в лістингі 9.2, а потім додавати і змінювати атрибути методами `deriveFont()` класу `Font`. Текст в `Java 2D` має свій власний контекст — об'єкт класу `FontRenderContext`, що зберігає всю інформацію, необхідну для виведення тексту. Отримати його можна методом `getFontRenderContext()` класу `Graphics2D`. Вся інформація про текст, в тому числі і про його контекст, збирається в об'єкті класу `TextLayout`. Цей клас в `Java 2D` заміняє клас `FontMetrics`. В конструкторі класу `TextLayout` задається текст, шрифт і контекст. Початок методу `paint()` з усіма цими визначеннями може виглядати так:

```
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
FontRenderContext frc = g.getFontRenderContext();
Font f = new Font("Serif", Font.BOLD, 15);
String s = "Якийсь текст";
TextLayout tl = new TextLayout(s, f, frc); // Продовження методу }
```

В класі `TextLayout` єсть не тільки більше двадцяти методів `getxxxo`, що дозволяють узнати різні дані про текст, його шрифте і контекст, але і метод

`draw(Graphics2D g, float x, float y)`

що викresлює зміст об'єкта класу `TextLayout` в графічній області `g`, починаючи з точки `(x, y)`. Ще один цікавий метод

`getOutline(AffineTransform at)`

повертає контур шрифту у вигляді об'єкта `shape`. Цей контур можна потім заповнити по якому-небудь зразку або вивести тільки контур, як показано в лістингі 9.6.

Лістинг 9.6. Виведення тексту засобами Java 2D

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.event.*;
class StillText extends Frame{
StillText(String s) {
super(s);
setSize(400, 200);
setVisible(true);
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent ev) {
System.exit(0) ;
}
});
```

```

}
public void paint(Graphics gr){
Graphics2D g = (Graphics2D)gr;
int w = getSize().width, h = getSize().height;
FontRenderContext frc = g.getFontRenderContext();
String s = "Тінь";
Font f = new Font("Serif", Font.BOLD, h/3);
TextLayout tl = new TextLayout(s, f, frc);
AffineTransform at = new AffineTransform();
at.setToTranslation(w/2-tl.getBounds().getWidth()/2, h/2);
Shape sh = tl.getOutline(at);
g.draw(sh);
AffineTransform atsh = new AffineTransform(1, 0.0, 1.5, -1, 0.0, 0.0);
g.transform(at);
g.transform(atsh);
Font df = f.deriveFont(atsh);
TextLayout dtl = new TextLayout(s, df, frc);
Shape sh2 = dtl.getOutline(atsh);
g.fill(sh2); }

public static void main(String[] args) {
new StillText(" Ефект тіні");
}
}

```

На рис. 9.6 показано виведення цієї програми.

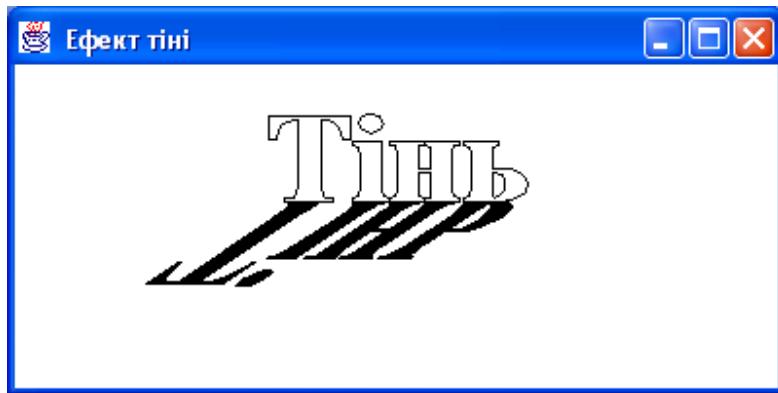


Рис. 9.6. Виведення тексту засобами Java 2D

Ще одна можливість створити текст з атрибутами — визначити об'єкт класу [AttributedString](#) із пакета [java.text](#). Конструктор цього класу

`AttributedString(String text, Map attributes)`

задає зразу і текст, і його атрибути. Потім можна додати або змінити характеристики тексту одним із трьох методів `addAttribute()`.

Якщо текст займає декілька рядків, то постає питання його форматування. Для цього замість класу `TextLayout` використовується клас `LineBreakMeasurer`, методи якого дозволяють відформатувати абзац. Для кожного сегмента тексту можна одержати екземпляр класу `TextLayout` і вивести текст, використовуючи його атрибути. Для редагування тексту необхідно відслідкувати курсором (`caret`) поточну позицію в тексті. Це здійснюється методами класу `TextHitInfo`, а методи класу `TextLayout` дозволяють одержати позицію курсора, виділити блок тексту і підсвітити його. Нарешті, можна задати окремі правила для виведення кожного символу текста. Для цього треба одержати екземпляр класу `Glyphvector` методом

`createGlyphVector()` класу `Font`, змінити позицію символу методом `setGlyphPosition()`, задати перетворення символу, якщо це допустимо для даного шрифта, методом `setGlyphTransform()`, і вивести змінений текст методом `drawGlyphVector()` класу `Graphics2D`. Все це показано в лістингі 9.7 і на рис. 9.7 — виведення програми лістинга 9.7.

Лістинг 9.7. Виведення окремих символів

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.event.*;
class GlyphTest extends Frame{ GlyphTest(String s){ super(s) ;
setSize(400, 150);
setVisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
public void paint(Graphics gr){
int h = 5;
Graphics2D g = (Graphics2D)gr;
FontRenderContext frc = g.getFontRenderContext();
Font f = new Font("Serif", Font.BOLD, 30);
GlyphVector gv = f.createGlyphVector(frc, "Танцюючий текст");
int len = gv.getNumGlyphs();
for (int i = 0; i < len; i++){
Point2D.Double p = new Point2D.Double(25 * i, h = -h);
gv.setGlyphPosition(i, p) ;
}
g.drawGlyphVector(gv, 10, 100); }
public static void main(String[] args) {
new GlyphTest(" Виведення окремих символів");
}
}
```

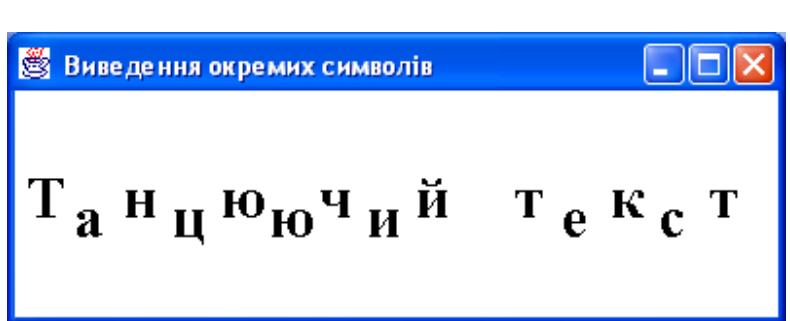


Рис. 9.7. Виведення окремих символів

9.9. Методи покращення візуалізації

Візуалізацію (`rendering`) створеної графіки можна удосконалити, установивши один із методів (`hint`) покращення одним із методів класу `Graphics2D`:

```
setRenderingHints(RenderingHints.Key key, Object value)
setRenderingHints(Map hints)
```

Ключі — методи покращення — і їх значення задаються константами класу `RenderingHints`,

перечисленними в табл. 9.2.

Таблиця 9.2. Методи візуалізації і їх значення

Метод	Значення
KEY_ANTIALIASING	Розмивання крайніх пікселів ліній для гладкості зображення; три значення, що задаються константами VALUE_ANTIALIAS_DEFAULT, VALUE_ANTIALIAS_ON, VALUE_ANTIALIAS_OFF
KEY_TEXT_ANTIALIASING	Те ж для тексту. Константи: VALUE_TEXT_ANTIALIASING_DEFAULT, VALUE_TEXT_ANTIALIASING_ON, VALUE_TEXT_ANTIALIASING_OFF
KEY_RENDERING	Три типи візуалізації. Константи: VALUE_RENDER_SPEED, VALUE_RENDER_QUALITY, VALUE_RENDER_DEFAULT
KEY_COLOR_RENDERING	Те ж для кольору. Константи: VALUE_COLOR_RENDER_SPEED, VALUE_COLOR_RENDER_QUALITY, VALUE_COLOR_RENDER_DEFAULT
KEY_ALPHA_INTERPOLATION	Плавне спряження ліній. Константи: VALUE_ALPHA_INTERPOLATION_SPEED, VALUE_ALPHA_INTERPOLATION_QUALITY, VALUE_ALPHA_INTERPOLATION_DEFAULT
KEY_INTERPOLATION	Способи спряження. Константи: VALUE_INTERPOLATION_BILINEAR, VALUE_INTERPOLATION_BICUBIC, VALUE_INTERPOLATION_NEAREST_NEIGHBOR
KEY_DITHERING	Заміна близьких кольорів. Константи: VALUE_DITHER_ENABLE, VALUE_DITHER_DISABLE, VALUE_DITHER_DEFAULT

Не всі графічні системи забезпечують виконання цих методів, тому задання указаних атрибутів не означає, що визначені ними методи будуть застосовуватися на самім ділі. Ось як може виглядати початок метода `paint()` з указням методів покращення візуалізації:

```
public void paint(Graphics gr){
    Graphics2D g = (Graphics2D)gr;
    g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
    g.setRenderingHint(RenderingHints.KEY_RENDERING,
    RenderingHints.VALUE_RENDER_QUALITY);
    // Продовження методу
}
```

9.10. Заключення

В цьому уроці ми, розуміється, не змогли детально разібрати всі можливості [Java 2D](#). Ми не торкнулися моделів задання кольору і змішування кольорів, друку графіки і тексту, динамічного завантаження шрифтів, зміни області рисування. В уроці 15 ми розглянемо засоби [Java 2D](#) для роботи із зображеннями, в уроці 18 — засоби друку. В документації [SUN J2SDK](#), в каталозі [docs\guide\2d\spec](#), єсть керівництво [Java 2 API Guide](#) з оглядом всіх можливостей [Java 2D](#). Там єсть посилання на посібники по [Java 2D](#). В каталозі [demo\jfc\Java2D\src](#) приведені вихідні тексти програм, що використовують [Java 2D](#).

Лабораторна робота 8. Робота з класом `Graphics` і `Graphics2D`

1. Ви вже, без сумніву, випробували програму лістинга 9.2. Візьміть її за зразок і виведіть у вікно всі фігури, перечислені в параграфі 9.1.2. Нижче кожної фігури виведіть її назву. Вибираєте для кожної фігури свій колір. Дляожної назви виберіть свій шрифт, розмір, колір і стиль. Особливу увагу зверніть на способи задання вершин полігона і ламаної. Ви повинні проілюструвати всі можливі випадки.
2. Розберіться з програмою 9.5 і попробуйте залити пару фігур узорами відмінними від використаних у цій програмі, в тому числі розгляніть і заливку суцільним кольором.

Програмування у Java

Урок 10. Основні компоненти

- Клас Component
- Клас Cursor
- Як створити свій курсор
- Події
- Клас Container
- Компонент Label
- Компонент Button
- Компонент Checkbox
- Клас CheckboxGroup
- Як створити групу радіокнопок
- Компонент Choice
- Компонент List
- Компоненти для введення тексту
- Клас TextComponent
- Компонент TextField
- Компонент TextArea
- Компонент ScrollBar
- Контейнер Panel
- Контейнер ScrollPane
- Контейнер Window
- Контейнер Frame
- Контейнер Dialog
- Контейнер FileDialog
- Створення власних компонентів
- Компонент Canvas
- Створення “легкого” компонента

Графічна бібліотека [AWT](#) пропонує більше двадцяти готових компонентів. Вони показані на рис. 8.2. Найбільш часто використовуються підкласи класу [Component](#): класи [Button](#), [Canvas](#), [Checkbox](#), [Choice](#), [Container](#), [Label](#), [List](#), [Scrollbar](#), [TextArea](#), [TextField](#), [Panel](#), [ScrollPane](#), [Window](#), [Dialog](#), [FileDialog](#), [Frame](#). Ще одна група компонентів — це компоненти меню — класи [MenuItem](#), [MenuBar](#), [Menu](#), [PopupMenu](#), [CheckboxMenuItem](#). Ми розглянемо їх в уроці 13. Забігаючи наперед, для кожного компонента перечислимо події, які в ньому відбуваються. Обробку подій ми розберемо в уроці 12. Почнемо вивчати ці компоненти від простих компонентів до складних і від найбільш часто використовуваних до використовуваних рідше. Але спочатку подивимося на те спільне, що єсть у всіх цих компонентах, а саме клас [Component](#).

10.1. Клас Component

Клас Component — центр бібліотеки [AWT](#) — дуже великий і володіє багатьма можливостями. В ньому п'ять статичних констант, визначаючих розміщення компонента всередині простору, виділеного для компонента у вміщаючому його контейнері: [BOTTOM_ALIGNMENT](#), [CENTER_ALIGNMENT](#), [LEFT_ALIGNMENT](#), [RIGHT_ALIGNMENT](#), [TOP_ALIGNMENT](#), і близько сотні методів. Більшість методів — це методи доступу [getxxx\(\)](#), [isxxx\(\)](#), [setxxx\(\)](#). Вивчати їх немає рації, треба просто подивитися, як вони використовуються в підкласах.

Конструктор класу недоступний — він захищений ([protected](#)), тому, що клас [Component](#) абстрактний, він не може використовуватися сам по собі, використовуються лише його підкласи. Компонент завжди займає прямокутну область зі сторонами, паралельними сторонам екрана і в кожний момент часу має певні розміри, вимірювані в пікселях, які можна узнати методом [getSize\(\)](#), повертуючим об'єкт класу [Dimension](#), або цілочисельними методами [getHeight\(\)](#) і [getWidth\(\)](#), повертуючими висоту і ширину прямокутника. Новий розмір компонента можна установити із програми методами [setSize\(Dimension d\)](#) або [setSize\(int width, int height\)](#), якщо це дозволяє менеджер розміщення контейнера, що містить компонент. У

компоненту єсть оптимальний розмір, при якому він виглядає найбільш пропорціонально. Його можна одержати методом `getPreferredSize()` у вигляді об'єкта `Dimension`.

Компонент має мінімальний і максимальний розміри. Їх повертають методи `getMinimumSize()` і `getMaximumSize()` у вигляді об'єкта `Dimension`. В компоненті єсть система координат. Її початок - точка з координатами `(0, 0)` - знаходиться в лівому верхньому куті компонента, вісь `Ox` іде вправо, вісь `Oy` - вниз, координатні точки розташовані між пікселями. В компоненті зберігаються координати його лівого верхнього кута в системі координат вміщаючого його контейнера. Їх можна узнати методами `getLocation()`, а змінити — методами `setLocation()`, перемістивши компонент в контейнер, якщо це дозволить менеджер розміщення компонентів.

Можна вияснити зразу і положення, і розмір прямокутної області компонента методом `getBounds()`, повертаючим об'єкт класу `Rectangle`, і змінити разом і положення, і розмір компонента методами `setBounds()`, якщо це дозволить зробити менеджер розміщення. Компонент може бути недоступним для дій користувача, тоді він виділяється на екрані світло-сірим кольором. Доступність компонента можна перевірити логічним методом `isEnabled()`, а змінити — методом `setEnabled(boolean enable)`. Для багатьох компонентів визначається графічний контекст — об'єкт класу `Graphics`, — який керується методом `paint()`, описаним в попередньому уроці, і який можна одержати методом `getGraphics()`.

В контексті єсть поточний колір і колір фону — об'єкти класу `Color`. Колір фону можна одержати методом `getBackground()`, а змінити — методом `setBackground(Color color)`. Поточний колір можна одержати методом `getForeground()`, а змінити — методом `setForeground(Color color)`. В контексті єсть шрифт — об'єкт класу `Font`, що повертається методом `getFont()` і змінюється методом `setFont(Font font)`. В компоненті визначається локаль — об'єкт класу `Locale`. Його можна одержати методом `getLocale()`, змінити — методом `setLocale(Locale locale)`.

В компоненті існує курсор, що показує положення миші, — об'єкт класу `Cursor`. Його можна одержати методом `getCursor()`, змінюється форма курсора у "важких" компонентах за допомогою метода `setCursor(Cursor cursor)`. Зупинимося на цих класах детальніше.

10.2. Клас Cursor

Основа класу — статичні константи, що визначають форму курсора:

- `CROSSHAIR_CURSOR` — курсор у вигляді хреста, появляється при пошуку позиції для розміщення якогось елемента;
- `DEFAULT_CURSOR` — звичайна форма курсора — стрілка вліво вверх;
- `HAND_CURSOR` — "указуючий перст", появляється при виборі якогось елемента списку;
- `MOVE_CURSOR` — хрест зі стрілками, появляється при переміщенні елемента;
- `TEXT_CURSOR` — вертикальна риска, появляється в текстових полях;
- `WAIT_CURSOR` — зображення годинника, появляється при очікуванні.

Наступні курсори появляються при наближенні до краю або кута компонента:

- `E_RESIZE_CURSOR` — стрілка вправо з упором;
- `N_RESIZE_CURSOR` — стрілка вверх з упором;
- `NE_RESIZE_CURSOR` — стрілка вправо вверх, упираюча в кут;
- `NW_RESIZE_CURSOR` — стрілка вліво вверх, упираюча в кут;
- `S_RESIZE_CURSOR` — стрілка вниз з упором;
- `SE_RESIZE_CURSOR` — стрілка вправо вниз, упираюча в кут;
- `SW_RESIZE_CURSOR` — стрілка вліво вниз, упираюча в кут;
- `W_RESIZE_CURSOR` — стрілка вліво з упором.

Перечислені константи являються аргументом `type` в конструкторі класу `Cursor(int type)`. Замість конструктора можна звернутися до статичного методу `getPredefinedCursor(int type)`, створюючому об'єкт класу `Cursor` і повертаючому посилання на нього. Одержані курсор по замовчуванню можна статичним методом `getDefaultcursor()`. Потім створений курсор треба установити в компонент. Наприклад, після виконання:

```
Cursor curs = new Cursor(Cursor.WAIT_CURSOR);
SomeComp.setCursor(curs);
```

при появі показчика миші в компоненті `Somecomp` показчик прийме вигляд годинника.

10.3. Як створити свій курсор

Крім цих наперед визначених курсорів можна задати свою власну форму курсора. Її тип носить назву `CUSTOM_CURSOR`. Сформувати свій курсор можна методом

```
createCustomCursor(Image cursor, Point hotspot, String name)
```

створюючим об'єкт класу `Cursor` і повертаючим посилання на нього. Перед цим належить створити зображення курсора `cursor` — об'єкт класу `Image`. Як це зробити, розповідається в уроці 15. Аргумент `name` задає ім'я курсору, можна написати просто `null`. Аргумент `hotspot` задає точку фокуса курсора. Ця точка повинна бути в межах зображення курсора, точніше, в межах, показуваних методом

```
getBestCursorSize(int desiredWidth, int desiredHeight)
```

повертаючим посилання на об'єкт класу `Dimension`. Аргументи методу означають бажаний розмір курсора. Якщо графічна система не допускає створення курсорів, повертається `(0,0)`. Цей метод показує приблизно розмір того курсора, який створить графічна система, наприклад, `(32, 32)`. Зображення `cursor` буде підігнано під цей розмір, при цьому можливі спотворення.

Третій метод — `getMaximumCursorColors()` — повертає найбільшу кількість кольорів, наприклад, 256, яку можна використовувати в зображенні курсора. Це методи класу `java.awt.Toolkit`, з яким ми ще не працювали. Клас `Toolkit` містить деякі методи, зв'язуючі додатки `Java` із засобами платформи, на якій виконується додаток. Тому не можна створити екземпляр класу `Toolkit` конструктором, для його одержання належить виконати статичний метод `Toolkit.getDefaultToolkit()`. Якщо додаток працює у вікні `Window` або його розширеннях, наприклад, `Frame`, то можна одержати екземпляр `Toolkit` методом `getToolkit()` класу `Window`.

Зберемо все це разом:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
class SimpleFrame extends Frame{
    SimpleFrame(String s){
        super (s);
        setSize(400, 150);
        setVisible(true);
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent ev) {
                System.exit (0);
            }
        });
        Toolkit tk = Toolkit.getDefaultToolkit();
        int colorMax = tk.getMaximumCursorColors(); // Найбільше число кольорів
        Dimension d = tk.getBestCursorSize(50, 50); // d - розмір зображення
        int w = d.width, h = d.height, k = 0;
        Point p = new Point(0, 0); // Фокус курсора буде в його верхньому лівому куті
        int[] pix = new int[w * h]; // Тут будуть пікселі зображення
        for(int i = 0; i < w; i++) {
            for(int j = 0; j < h; j++) {
                if (j < i) pix[k++] = 0xFFFF0000; // Лівий нижній кут - червоний
                else pix[k++] = 0; // Правий верхній кут - прозорий
            }
        }
        // Створюється прямокутне зображення розміром (w, h),
```

```

// заповнене масивом пікселів pix, з довжиною рядка w
Image im = createImage(new MemoryImageSource(w, h, pix, 0, w));
Cursor curs = tk.createCustomCursor(im, p, null);
setCursor(curs);
}
public static void main(String[] args) {
new SimpleFrame(" My program");
}
}

```



В цьому прикладі створюється курсор у вигляді червоного прямокутного трикутника з катетами розміром 32 пікселі і установлюється в якомусь компоненті `someComp`.

10.4. Події

- Подія `ComponentEvent` відбувається при переміщенні компонента, зміні його розміру, видаленні з екрана і появлі на екрані.
- Подія `FocusEvent` відбувається при одерженні або втраті фокуса.
- Подія `KeyEvent` проявляється при кожному натисненні і звільненні клавіші, якщо компонент має фокус введення.
- Подія `MouseEvent` відбувається при маніпуляціях миші на компоненті.

Кожний компонент перед виведенням на екран поміщається в контейнер — підклас класу `Container`. Познайомимося з цим класом.

10.5. Клас Container

Клас `Container` — прямий підклас класу `Component`, і наслідує всі його методи. Крім них основу класу складають методи додавання компонентів у контейнер:

- `add (Component comp)` — компонент `comp` додається в кінець контейнера;
- `add (Component comp, int index)` — компонент `comp` додається в позицію `index` в контейнері, якщо `index == -1`, то компонент додається в кінець контейнера;
- `add (Component comp, object constraints)` — менеджеру розміщення контейнера даються вказівки об'єктом `Constraints`;
- `add (String name, Component comp)` — компонент отримує ім'я `name`.

Два методи видаляють компоненти із контейнера:

- `remove (Component comp)` — видаляє компонент з іменем `comp`;
- `remove (int index)` — видаляє компонент з індексом `index` в контейнері.

Один із компонентів в контейнері отримує фокус вводу (`input focus`), на нього направляється введення з клавіатури. Фокус можна переносити з одного компонента на інший клавішами `<Tab>` і `<Shift>+<Tab>`. Компонент може запросити фокус методом `requestFocus()` і передати фокус наступному компоненту методом `transferFocus()`. Компонент може перевірити, чи має він фокус, своїм логічним методом `hasFocus(f)`. Це методи класу `Component`.

Для полегшення розміщення компонентів в контейнері визначається менеджер розміщення (`layout`

`manager`) — об'єкт, реалізуючий інтерфейс `LayoutManager` або його підінтерфейс `LayoutManager2`. Кожний менеджер розміщує компоненти в якомусь своєму порядку: один менеджер розставляє компоненти в таблицю, другий норовить розтягти компоненти по сторонах, третій просто розміщує їх один за другим, як слова в тексті. Менеджер визначає зміст слів "додати в кінець контейнера" і "додати в позицію `index`". В контейнері в будь-який момент часу може бути установлений тільки один менеджер розміщення. В кожному контейнері єсть свій менеджер по замовчуванню, установка іншого менеджера виконується методом `setLayout(LayoutManager manager)`. Менеджери розміщення ми розглянемо детальніше в наступному уроці. В даному уроці ми будемо розміщувати компоненти вручну, відключивши менеджер по замовчуванню методом `setLayout(null)`.

Події

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` при додаванні і видаленні компонентів в контейнері відбувається подія `ContainerEvent`. Переїдемо до розгляду конкретних компонентів. Самий простий компонент описує клас `Label`.

10.6. Компонент `Label`

Компонент `Label` — це просто рядок тексту, оформленій як графічний компонент для розміщення в контейнері. Текст можна поміняти тільки методом доступу `setText(String text)`, але не введенням користувача з клавіатури або за допомогою миші. Створюється об'єкт цього класу одним із трьох конструкторів:

- `Label ()` — пустий об'єкт без тексту;
- `Label (String text)` — об'єкт з текстом `text`, який притискується до лівого краю компонента;
- `Label (String text, int alignment)` — об'єкт з текстом `text` і визначенням розміщенням в компоненті тексту, задаваного однією із трьох констант: `CENTER`, `LEFT`, `RIGHT`.

Розміщення можна змінити методом доступу `setAlignment(int alignment)`. Решта методів, крім методів, унаслідуваних від класу `Component`, дозволяють одержати текст `getText()` і розміщення `getAlignment()`.

Події

В класі `Label` відбуваються події класів `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`. Ненабагато складніший клас `Button`.

10.7. Компонент `Button`

Компонент `Button` — це кнопка стандартного для даної графічної системи вигляду з написом, здатна реагувати на клік кнопки миші — при натискуванні вона "вдавлюється" в площину контейнера, при відпусканні — становиться "випуклою". Два конструктори `Button()` і `Button (String label)` створюють кнопку без напису і з написом `label` відповідно. Методи доступа `getLabel()` і `setLabel (String label)` дозволяють одержати і змінити напис на кнопці. Головна функція кнопки — реагувати на клік миші, і інші методи класу обробляють ці дії. Ми розглянемо їх в уроці 12.

Події

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при кліку на кнопку відбувається подія `ActionEvent`. Трохи складніший за клас `Label` клас `Checkbox`, створюючий кнопки вибору.

10.8. Компонент `Checkbox`

Компонент `Checkbox` — це напис справа від невеликого квадратика, в якому в деяких графічних системах появляється галочка після кліку кнопкою миші — компонент переходить в стан (`state`) `on`. Після наступного кліку галочка пропаде — це стан `off`. В інших графічних системах стан `on` відмічається "вдавлюванням" квадратика. В компоненті `Checkbox` стани `on/off` відмічаються логічними значеннями `true/false` відповідно.

Три конструктори `Checkbox ()`, `Checkbox(String label)`, `Checkbox(String label, boolean state)` створюють

компонент без напису, з написом `label` в стані `off`, і в заданому стані `state`. Методи доступу `getLabel()`, `setLabel (String label)`, `getState()`, `setState (boolean state)` повертають і змінюють ці параметри компонента. Компоненти `Checkbox` зручні для швидкого і наочного вибору із списку, цілком розташованого на екрані, як показано на рис. 10.1. Там же продемонстрована ситуація, в якій потрібно вибрати тільки один пункт із декількох. В таких ситуаціях створюється група так званих `радіокнопок` (`radio buttons`). Вони помічаються кружком або ромбиком, а не квадратиком, вибір позначається жирною точкою в кружку або "вдавлюванням" ромбика.

Події

В класі `Checkbox` відбуваються події класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, а при зміні стану кнопки виникає подія `ItemEvent`. В бібліотеці `AWT` радіокнопки не створюють окремий компонент. Замість цього декілька компонентів `Checkbox` об'єднуються в групу за допомогою об'єкту класу `CheckboxGroup`.

10.9. Клас `CheckboxGroup`

Клас `CheckboxGroup` дуже малий, оскільки його завдання — просто дати спільне ім'я всім об'єктам `Checkbox`, утворюючим одну групу. В нього входить один конструктор по замовчуванню `CheckboxGroup()` і два методи доступу:

- `getSelectedCheckbox()`, повертаючий вибраний об'єкт `Checkbox`;
- `setSelectedCheckbox (Checkbox box)`, задаючий вибір.

10.10. Як створити групу радіокнопок

Щоб організувати групу радіокнопок, треба спочатку сформувати об'єкт класу `CheckboxGroup`, а потім створити кнопки конструкторами

- `Checkbox(String label, CheckboxGroup group, boolean state)`
- `Checkbox(String label, boolean state, CheckboxGroup group)`

Ці конструктори ідентичні, просто при запису конструктора можна не думати про порядок слідування його аргументів. Тільки одна радіокнопка в групі може мати стан `state = true`. Пора привести приклад. В лістингі 10.1 приведена програма, поміщаюча в контейнер `Frame` дві мітки `Label` зверху, під ними зліва три об'єкти `Checkbox`, справа — групу радіокнопок. Внизу — три кнопки `Button`. Результат виконання програми показаний на рис. 10.1.

Лістинг 10.1. Розміщення компонентів

```
import java.awt.*;
import java.awt.event.*;
class SimpleComp extends Frame{
SimpleComp(String s){ super(s);
setLayout(null);
Font f = new Font("Serif", Font.BOLD, 15);
setFont(f);
Label L1 = new Label("Choose things:", Label.CENTER);
L1.setBounds(0, 50, 120, 30); add(L1);
Label L2 = new Label("Choose paying:");
L2.setBounds(160, 50, 200, 30); add(L2);
Checkbox ch1 = new Checkbox("Books");
ch1.setBounds(20, 90, 100, 30); add(ch1);
Checkbox ch2 = new Checkbox("Discs");
ch2.setBounds(20, 120, 100, 30); add(ch2);
Checkbox ch3 = new Checkbox("Tois");
ch3.setBounds(20, 150, 100, 30); add(ch3);
CheckboxGroup grp = new CheckboxGroup();
Checkbox chg1 = new Checkbox("Mail", grp,true);
```

```
chgl.setBounds(170, 90, 200, 30); add(chg1);
Checkbox chg2 = new Checkbox("Credit Card", grp, false);
chg2.setBounds(170, 120, 200, 30); add(chg2);
Button b1 = new Button("Continue");
b1.setBounds( 30, 220, 100, 30); add(b1);
Button b2 = new Button("Cancel");
b2.setBounds(140, 220, 100, 30); add(b2);
Button b3 = new Button("Exit");
b3.setBounds(250, 220, 100, 30); add(b3);
setSize(400, 300);
setVisible(true);
}
public static void main(String[] args){
Frame f = new SimpleComp (" Прості компоненти");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

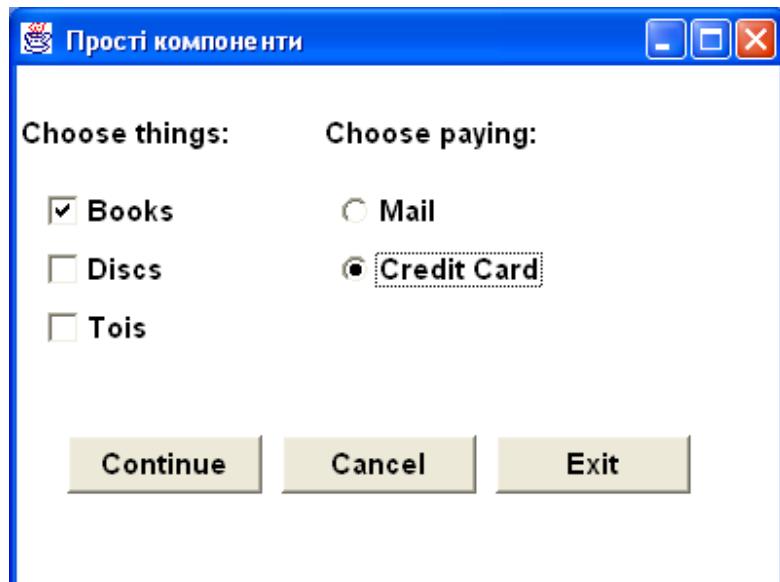


Рис. 10.1. Прості компоненти

Ми не програмували реакції на натискання кнопок, тому такі натискання не мають жодного ефекту. Відмітьте, що кожний створюваний компонент належить заносити в контейнер, в даному випадку [Frame](#), методом [add\(\)](#). Лівий верхній кут компонента поміщається в точку контейнера з координатами, указаними першими двома аргументами методу [setBounds\(\)](#). Розмір компонента задається останніми двома параметрами цього методу. Якщо немає необхідності відображати весь список на екрані, то замість групи радіокнопок можна створити розкриваючийся список — об'єкт класу [Choice](#).

10.11. Компонент Choice

Компонент **Choice** — це розкриваючийся список, один, вибраний, пункт (**item**) якого видимий в полі, а інші появляються при кліку кнопкою миші на невелику кнопку справа від поля компонента. Спочатку конструктором **Choice()** створюється пустий список. Потім, методом **add (String text)**, в список додаються нові пункти з текстом **text**. Вони розміщуються в порядку написання методов **add()** і нумеруються від нуля. Вставити новий пункт в потрібне місце можна методом **insert (String text, int position)**. Вибір пункту можна

зробити із програми методом `select (String text)` або `select(int position)`. Видалити один пункт із списку можна методом `remove(String text)` або `remove (int position)`, а всі пункти зразу — методом `remove()`. Число пунктів у списку можна узнати методом `getItemCount()`. Вияснити, який пункт знаходиться в позиції `pos` можна методом `getItem(int pos)`, повертаючим рядок. Нарешті, визначення вибраного пункту виконується методом `getSelectedIndex()`, повертаючим позицію цього пункту, або методом `getSelectedItem()`, повертаючим виділений рядок.

Події

В класі `Choice` відбуваються події класу `Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent`, а при виборі пункта відбувається подія `ItemEvent`. Якщо треба показати на екрані декілька пунктів списку, то створіть об'єкт класу `List`.

10.12. Компонент List

Компонент `List` — це список із смugoю прокрутки, в якому можна виділити один або декілька пунктів. Кількість видимих на екрані пунктів визначається конструктором списку і розміром компонента. В класі три конструктори:

- `List()` — створює пустий список з чотирма видимими пунктами;
- `List (int rows)` — створює пустий список з `rows` видимими пунктами;
- `List (int rows, boolean multiple)` — створює пустий список в якому можна відмітити декілька пунктів, якщо `multiple == true`.
-

Після створення об'єкта в список додаються пункти з текстом `item`:

- метод `add (String item)` — додає новий пункт в кінець списку;
- метод `add (String item, int position)` — додає новий пункт в позицію `position`.

Позиції нумеруються по порядку, починаючи з нуля. Видалити пункт можна методами

`remove (String item)`, `remove (int position)`, `removeAll ()`.

Метод `replaceItem(String newitem, int pos)` дозволяє замінити текст пункта в позиції `pos`. Кількість пунктів у списку повертає метод `getItemCount()`. Виділений пункт можна отримати методом `getSelectedItem()`, а його позицію — методом `getSelectedIndex()`. Якщо список дозволяє здійснювати множинний вибір, то виділені пункти у вигляді масиву типу `String[]` можна одержати методом `getSelectedItems()`, позиції виділених пунктів у вигляді масиву типу `int[]` — методом `getSelectedIndexes()`. Крім цих необхідних методів клас `List` містить багато інших, дозволяючих маніпулювати пунктами списку і отримувати його характеристики.

Події

Крім подій класу `Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent`, при подвійному кліку кнопкою миші на вибраному пункті відбувається подія `ActionEvent`. В лістингі 10.2 задаються компоненти, аналогічні компонентам лістинга 10.1, за допомогою класів `Choice` і `List`, а рис. 10.2 показує, як зміниться при цьому інтерфейс.

Лістинг 10.2. Використання списків

```
import java.awt.*;
import java.awt.event.*;
class ListTest extends Frame{
ListTest(String s){ super(s);
setLayout(null);
setFont(new Font("Serif", Font.BOLD, 15));
Label l1 = new Label("Choice things:", Label.CENTER);
l1.setBounds(0, 50, 120, 30); add (l1);
Label l2 = new Label("Choice paying mode:");
l2.setBounds(170, 50, 200, 30); add(l2);
```

```

List l = new List(2, true);
l.add("Books");
l.add("Discs");
l.add("Tois");
l.setBounds(20, 90, 100, 40); add(l);
Choice ch = new Choice();
ch.add("By mail transfer");
ch.add("By credit card");
ch.setBounds(170, 90, 200, 30); add(ch);
Button b1 = new Button("Continue");
b1.setBounds(30, 150, 100, 30); add(b1);
Button b2 = new Button("Cancel");
b2.setBounds(140, 150, 100, 30); add(b2);
Button b3 = new Button("Exit");
b3.setBounds(250, 150, 100, 30); add(b3);
setSize(400, 200); setVisible(true);
}
public static void main(String[] args){
Frame f = new ListTest("Прості компоненти");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
}

```

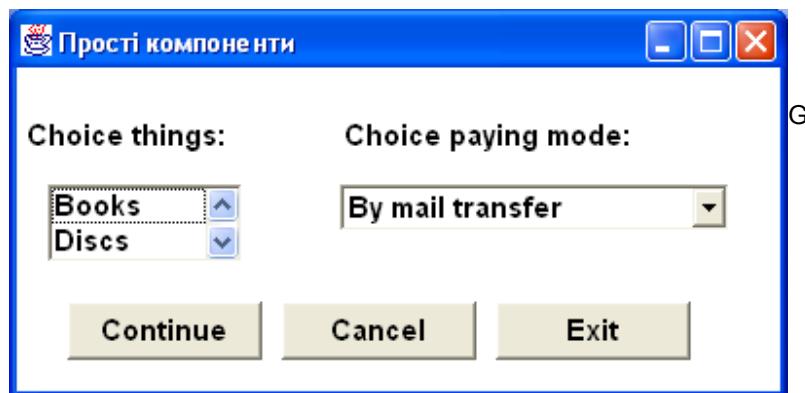


Рис. 10.2. Використання списків

10.13. Компоненти для введення тексту

В бібліотеці **AWT** є два компоненти для введення тексту з клавіатури: **TextField**, дозволяючий ввести тільки один рядок, і **TextArea**, в якому можна ввести декілька рядків. Обидва класи розширяють клас **TextComponent**, в якому зібрані їх спільні методи, такі як виділення тексту, позиціювання курсора, одержання тексту.

10.13.1. Клас **TextComponent**

В класі **TextComponent** немає конструктора, цей клас не використовується самостійно. Основний метод класу — метод **getText()** — повертає текст, що знаходиться в полі введення, у вигляді рядка. Поле введення може бути нередактованим, в цьому стані текст в полі не можна змінити з клавіатури або мишкою. Узнати стан поля можна логічним методом **isEditable()**, змінити значення в ньому — методом **setEditable(boolean editable)**. Текст, що знаходиться в полі, зберігається як об'єкт класу **String**, тому у кожного символу є індекс (у першого — індекс 0). Індекс використовується для визначення позиції

курсора (`caret`) методом `getCaretPosition()`, для установки позиції курсора методом `setCaretPosition(int ind)` і для виділення тексту. Текст виділяється мишею або клавішами зі стрілками при натисканні клавіші `<Shift>`, але можна виділити його із програми методом `select(int begin, int end)`. При цьому помічається текст від символа з індексом `begin` включно, до символа з індексом `end` виключно. Весь текст виділяє метод `selectAll()`. Можна відмітити початок виділення методом `setSelectionStart(int ind)` і кінець виділення методом `setSelectionEnd(int ind)`. Важливіше все-таки не задати, а отримати виделений текст. Його повертає метод `getSelectedText()`, а початковий і кінцевий індекс виділення повертають методи `getSelectionStart()` і `getSelectionEnd()`.

Події

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при зміні тексту користувачем відбувається подія `TextEvent`.

10.13.2. Компонент `TextField`

Компонент `TextField` — це поле для введення одного рядка тексту. Ширина поля вимірюється в колонках (`column`). Ширина колонки — це середня ширина символу в шрифті, яким вводиться текст. Натискування клавіші `<Enter>` закінчує введення і служить сигналом до початку обробки введеного тексту, тобто при цьому відбувається подія `ActionEvent`. В класі чотири конструктори:

- `TextField()` — створює пусте поле ширину в одну колонку;
- `TextField(int columns)` — створює пусте поле з числом колонок `columns`;
- `TextField(String text)` — створює поле з текстом `text`;
- `TextField(String text, int columns)` — створює поле з текстом `text` і числом колонок `columns`.

До методів, унаслідуваних від класу `TextComponent`, додаються ще методи `getColumns()` і `setColumns(int col)`. Цікава різновидність поля введення — поле для введення пароля. В такому полі замість введених символів появляється який-небудь особливий ехо-символ, частіше всього зірочка, щоб пароль ніхто не підглянув через плече. Дане поле введення одержується виконанням методу `setEchoChar(char echo)`. Аргумент `echo` — це символ, який буде появлятися в полі. Перевірити, чи установлений ехо-символ, можна логічним методом `echoCharIsSet()`, одержати ехо-символ — методом `getEchoChar()`. Щоб првернути поле введення в звичайний стан, достатньо виконати метод `setEchoChar(0)`.

Події

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при зміні тексту користувачем відбувається подія `TextEvent`, а при натисканні на клавішу `<Enter>` — подія `ActionEvent`.

10.13.3. Компонент `TextArea`

Компонент `TextArea` — це область введення з довільним числом рядків. Натискання клавіші `<Enter>` просто переводить курсор в початок наступного рядка. В області введення можуть бути установлені смуги прокрутки, одна або обидві. Основний конструктор класу

`TextArea(String text, int rows, int columns, int scrollbars)`

створює область введення з текстом `text`, числом видимих рядків `rows`, числом колонок `columns`, і заданим смугам прокрутки `scrollbars` однієї із чотирьох констант: `SCROLLBARS_NONE`, `SCROLLBARS_HORIZONTAL_ONLY`, `SCROLLBARS_VERTICAL_ONLY`, `SCROLLBARS_BOTH`. Решта конструкторів задають деякі параметри по замовчуванню:

- `TextArea (String text, int rows, int columns)` — присутні обидві смуги прокрутки;
- `TextArea (int rows, int columns)` — в полі пустий рядок;
- `TextArea (String text)` — розміри устанавлює контейнер;
- `TextArea ()` — конструктор по замовчуванню.
-

Серед методів класу `TextArea` найбільш важливі методи:

- `append (String text)`, додаючий текст `text` в кінець уже введеного тексту;
- `insert (String text, int pos)`, вставляючий текст в указану позицію `pos`;
- `replaceRange (String text, int begin, int end)`, видаляючий текст починаючи з індекса `begin` включно по `end` виключно, і поміщаючий замість нього текст `text`.

Інші методи дозволяють змінити і отримати кількість видимих рядків.

Події

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при зміні тексту користувачем відбувається подія `TextEvent`.

В лістингі 10.3 створюються три поля: `tf1`, `tf2`, `tf3` для введення імені користувача, його пароля і замовлення, і не редагована область введення, в якій накопичується замовлення. В полі введення пароля `tf2` появляється ехо-символ *. Результат показаний на рис. 10.3.

Лістинг 10.3. Поля введення

```
import java.awt.*;
import java.awt.event.*;
class TextTest extends Frame{
TextTest(String s){
super(s);
setLayout(null);
setFont(new Font("Serif", Font.PLAIN, 14));
Label l1 = new Label("Your Name:", Label.RIGHT);
l1.setBounds(20, 30, 70, 25); add(l1);
Label l2 = new Label("Password:", Label.RIGHT);
l2.setBounds(20, 60, 70, 25); add(l2);
TextField tf1 = new TextField(30) ;
tf1.setBounds(100, 30, 160, 25); add(tf1);
TextField tf2 = new TextField(30);
tf2.setBounds(100, 60, 160, 25);
add(tf2); tf2.setEchoChar('*');
TextField tf3 = new TextField("Input here your request", 30);
tf3.setBounds(10, 100, 250, 30); add(tf3);
TextArea ta = new TextArea("Your request:", 5, 50,
TextArea.SCROLLBARS_NONE);
ta.setEditable(false);
ta.setBounds(10, 150, 250, 140); add(ta);
Button b1 = new Button("Apply");
b1.setBounds(280, 180, 100, 30); add(b1);
Button b2 = new Button("Cancel");
b2.setBounds(280, 220, 100, 30); add(b2);
Button b3 = new Button("Exit");
b3.setBounds(280, 260, 100, 30); add(b3);
setSize(400, 300); setVisible(true);
}
public static void main(String[] args){
Frame f = new TextTest(" Поля введення");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

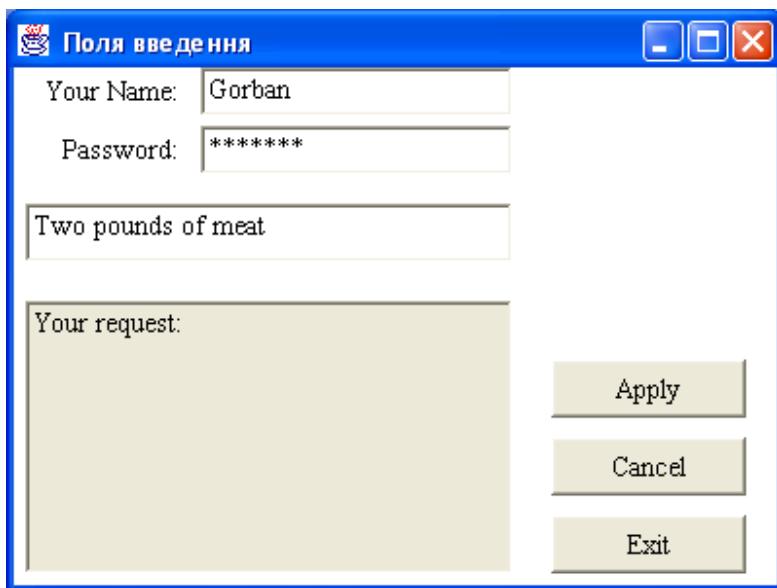


Рис. 10.3. Поля введення

10.14. Компонент Scrollbar

Компонент [Scrollbar](#) — це смуга прокрутки, але в бібліотеці [AWT](#) клас [Scrollbar](#) використовується ще й для організації

повзунка ([slider](#)). Об'єкт може розміщатися горизонтально або вертикально, зазвичай смуги прокрутки розміщуються внизу і справа. Кожна смуга прокрутки охвачує деякий діапазон значень і зберігає поточне значення із цього діапазону. В лінійці прокрутки є п'ять елементів управління для переміщення по діапазону. Дві стрілки на кінцях лінійки викликають переміщення на одну одиницю ([unit](#)) у відповідному напрямі при кліку на стрілку кнопкою миші. Положення движка або бігунка ([bubble, thumb](#)) показує поточне значення із діапазону і може його змінювати при переміщенні бігунка за допомогою миші. Два проміжки між движком і стрілками дозволяють переміститися на один блок ([block](#)) кліком кнопки миші. Зміст понять "одиниця" і "блок" залежить від об'єкта, з яким працює смуга прокрутки. Наприклад, для вертикальної смуги прокрутки при перегляді тексту це може бути рядок і сторінка або рядок і абзац. Методи роботи з даним компонентом описані в інтерфейсі [Adjustable](#), котрий реалізований класом [Scrollbar](#).

В класі [Scrollbar](#) три конструктори:

- [Scrollbar\(\)](#) — створює вертикальну смугу прокрутки з діапазоном 0—100, поточним значенням 0 і блоком 10 одиниць;
- [Scrollbar \(int orientation\)](#) — орієнтація [orientation](#) задається однією із двох констант [HORIZONTAL](#) або [VERTICAL](#);
- [Scrollbar\(int orientation, int value, int visible, int min, int max\)](#) — задає, крім орієнтації, ще початкове значення [value](#), розмір блоку [visible](#), діапазон значень [min—max](#).

Аргумент [visible](#) визначає ще і довжину движка — вона устанавлюється пропорціонально діапазону значень і довжині смуги прокрутки. Наприклад, конструктор по замовчуванню задасть довжину движка рівною 0,1 довжині смуги прокрутки. Основний метод класу — [getValue\(\)](#) — повертає значення поточного положення движка на смузі прокрутки. Останні методи доступу дозволяють узнати і змінити характеристики об'єкта, приклади їх використання показані в лістингі 12.6.

Події

Крім подій класу [Component](#): [ComponentEvent](#), [FocusEvent](#), [KeyEvent](#), [MouseEvent](#), при зміні значення користувачем відбувається подія [AdjustmentEvent](#).

В лістингі 10.4 створюються три вертикальні смуги прокрутки — червона, зелена і синя, дозволяючи вибрати якесь значення відповідного кольору в діапазоні 0—255, з початковим значенням 127. Крім них створюється область, заповнена отриманим кольором, і дві кнопки. Смуги прокрутки, їх заголовок і маштабні мітки поміщені в окремий контейнер [р](#) типу [Panel](#). Про це трохи пізніше в даному уроці. Як все це виглядує, показано на рис. 10.4. В лістинзі 12.6 ми "оживимо" цю програму.

Лістинг 10.4. Лінійки прокрутки для вибору кольору

```
import java.awt.*;
import java.awt.event.*;
class ScrollTest extends Frame{
Scrollbar sbRed = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
Scrollbar sbGreen = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
Scrollbar sbBlue = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
Color mixedColor = new Color(127, 127, 127);
Label lm = new Label();
Button b1 = new Button("Apply");
Button b2 = new Button("Cancel");
ScrollTest(String s){ super(s);
setLayout(null);
setFont(new Font("Serif", Font.BOLD, 15));
Panel p = new Panel();
p.setLayout(null);
p.setBounds(10,50, 150, 260); add(p);
Label lc = new Label("Choice color");
lc.setBounds(20, 0, 120, 30); p.add(lc);
Label lmin = new Label("0", Label.RIGHT);
lmin.setBounds(0, 30, 30, 30); p.add(lmin);
Label lmiddle = new Label("127", Label.RIGHT);
lmiddle.setBounds(0, 120, 30, 30); p.add(lmiddle);
Label lmax = new Label("255", Label.RIGHT);
lmax.setBounds(0, 200, 30, 30); p.add(lmax);
sbRed.setBackground(Color.red);
sbRed.setBounds(40, 30, 20, 200); p.add(sbRed);
sbGreen.setBackground(Color.green);
sbGreen.setBounds(70, 30, 20, 200); p.add(sbGreen);
sbBlue.setBackground(Color.blue);
sbBlue.setBounds(100, 30, 20, 200); p.add(sbBlue);
Label lp = new Label("Pattern:");
lp.setBounds(250, 50, 120, 30); add(lp);
lm.setBackground(new Color(127, 127, 127));
lm.setBounds(220, 80, 120, 80); add(lm);
b1.setBounds(240, 200, 100, 30); add(b1);
b2.setBounds(240, 240, 100, 30); add(b2);
setSize(400, 300); setVisible(true);
}
public static void main(String[] args){
Frame f = new ScrollTest("Color choice");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
```

```
}
```

В лістинзі 10.4 використаний контейнер [Panel](#). Розглянемо можливості цього класу.

10.15. Контейнер Panel

Контейнер [Panel](#) — це невидимий компонент графічного інтерфейса, призначений для обєднання декількох компонентів в один об'єкт типу [Panel](#). Клас [Panel](#) дуже простий, але важливий. В ньому всього два конструктори:

- [Panel\(\)](#) — створює контейнер з менеджером розміщення по замовчуванню [FlowLayout](#)
- [Panel\(LayoutManager layout\)](#) — створює контейнер з указанням менеджером розміщення компонентів [layout](#).

Після створення контейнера в нього додаються компоненти унаслідованим методом [add\(\)](#):

```
Panel p = new Panel()
p.add(compl);
p.add(comp2);
```

і т. д. Розміщує компоненти в контейнері його менеджер розміщення, про що ми поговоримо в наступному уроці.

Контейнер [Panel](#) використовується дуже часто. Він зручний для створення групи компонентів. В лістинзі 10.4 три смуги прокрутки разом із заголовком "Підберіть колір" і маштабними мітками 0, 127 і 255 утворюють природну групу. Якщо ми захочемо перемістити її в інше місце вікна, нам прийдеться переносити кожен із семи компонентів, що входять в указану групу. При цьому прийдеться слідкувати за тим, щоб їх взаємне положення не змінилося. Замість цього ми створили панель [p](#) і розмістили на ній всі сім елементів. Метод [setBounds\(\)](#) кожного із розглядуваних компонентів указує в даному випадку положення і розмір компонента в системі координат панелі [p](#), а не вікна [Frame](#). У вікно ми помістили зразу цілу панель, а не її окремі компоненти. Тепер для переміщення всієї групи компонентів досить перемістити панель, і розташовані на ній об'єкти автоматично перемістяться разом з нею, не змінивши свого взаємного положення.

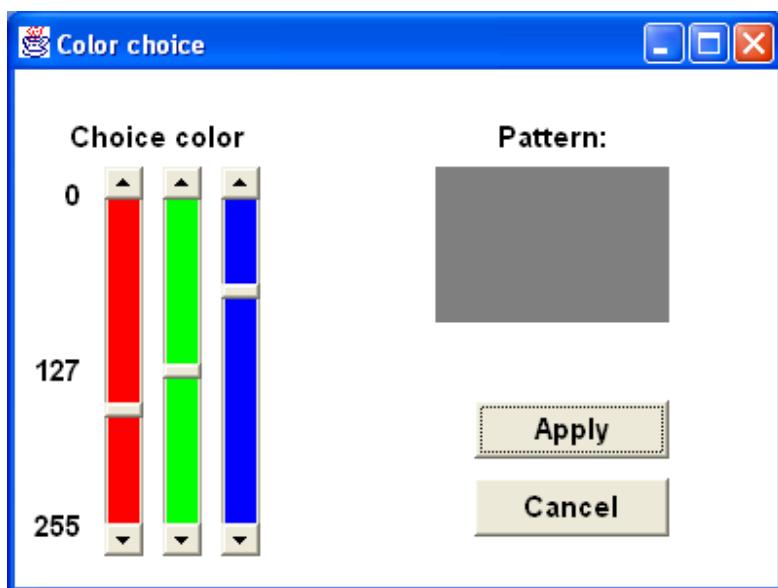


Рис. 10.4. Смуги прокрутки для вибору кольору

10.16. Контейнер ScrollPane

Контейнер [ScrollPane](#) може містити тільки один компонент, зате такий, який не поміщається цілком у вікні. Контейнер забезпечує засоби прокрутки для перегляду великого компонента. В контейнері можна установити смуги прокрутки або постійно, константою [SCROLLBARS_ALWAYS](#), або так, щоб вони появлялися тільки при необхідності (якщо компонент дійсно не поміщається у вікно) константою [SCROLLBARS_AS_NEEDED](#). Якщо смуги прокрутки не установлені, це задає константу [SCROLLBARS_NEVER](#), то переміщення компонента для перегляду потрібно забезпечити із програми одним із методов [setScrollPosition\(\)](#). В класі два конструктори:

- [ScrollPane\(\)](#) — створює контейнер, в якому смуги прокрутки появляються по необхідності;
- [ScrollPane\(int scrollbars\)](#) — створює контейнер, в якому появі лінійок прокрутки задається однією із трьох указаних вище констант.

Конструктори створюють контейнер розміром 100x100 пікселів, надалі можна змінити розмір унаслідуванням методом [setSize\(int width, int height\)](#). Обмеження, що [ScrollPane](#) може містити тільки один компонент, легко обходиться. Завжди можна зробити цим єдиним компонентом об'єкт класу [Panel](#), розмістивши на панелі що завгодно. Серед методів класу цікаві ті, що дозволяють прокручувати компонент в [ScrollPane](#):

- метод [getHAdjustable\(\)](#) і [getVAdjustable\(\)](#) повертають положення смуг прокрутки у вигляді інтерфейса [Adjustable](#);
- метод [getScrollPosition\(\)](#) показує координати [\(x,y\)](#) точки компонента, що знаходиться в лівому верхньому куті панелі [ScrollPane](#), у вигляді об'єкта класу [Point](#);
- метод [setScrollPosition\(Point p\)](#) або [setScrollPosition\(int x, int y\)](#) прокручує компонент в позицію [\(x, y\)](#).

10.17. Контейнер Window

Контейнер [Window](#) — це пусте вікно, без внутрішніх елементів: рамки, рядка заголовка, рядка меню, смуг прокрутки. Це просто прямокутна область на екрані. Вікно типу [Window](#) самостійно, не міститься ні в якому контейнері, його не треба заносити в контейнер методом [add\(\)](#). Однак воно не звязано з віконним менеджером графічної системи. Тому не можна змінити його розміри, перемістити в інше місце екрана. Воно може бути створено тільки яким-небудь уже існуючим вікном, власником ([owner](#)) або батьком ([parent](#)) вікна [Window](#). Коли вікно-власник приирається з екрана, разом з ним приирається і породжене вікно. Власник вікна указується в конструкторі:

- [window \(Frame f\)](#) — створює вікно, власник якого — фрейм [f](#);
- [Window \(Window owner\)](#) — створює вікно, власник якого - уже наявне вікно або підклас класу [Window](#).

Створене конструктором вікно не виводиться на екран автоматично. Його належить відобразити методом [show\(\)](#). Прибрати вікно з екрана можна методом [hide\(\)](#), а перевірити, чи видно вікно на екрані - логічним методом [isShowing\(\)](#). Вікно типу [Window](#) можна використовувати для створення спливаючих вікон попередження, повідомлення, підказки. Для створення діалогових вікон є підклас [Dialog](#), спливаючих меню — клас [popupMenu](#) (див. урок 13). Видиме на екране вікно виводиться на передній план методом [toFront\(\)](#) або, навпаки, поміщається на задній план методом [toBack\(\)](#). Знищити вікно, звільнивши зайняті ним ресурси, можна методом [dispose\(\)](#). Менеджер розміщення компонентів у вікні по замовчуванню — [BorderLayout](#). Вікно створює свій екземпляр класу [Toolkit](#), який можна отримати методом [getToolkit\(\)](#).

Події

Крім подій класу [Component](#): [ComponentEvent](#), [FocusEvent](#), [KeyEvent](#), [MouseEvent](#), при зміні розмірів вікна, його переміщенні або видаленні з екрана, а також показу на екрані відбувається подія [windowEvent](#).

10.18. Контейнер Frame

Контейнер [Frame](#) — це повноцінне готове вікно з рядком заголовку, в який поміщені кнопки контекстного

меню, звертання вікна в ярлик і розвертання на весь екран і кнопка закриття додатку. Заголовок вікна записується в конструкторі або методом `setTitle(String title)`. Вікно обмежено рамкою. В нього можна установити рядок меню методом `setMenuBar (MenuBar mb)`. Це ми обговоримо в уроці 13. На кнопці контекстного меню в лівій частині рядка заголовка зображена димляча чашечка кофе — логотип Java. Ви можете установити там друге зображення методом `setIconImage(image icon)`, створивши попередньо зображення `icon` у вигляді обєкта класу `image`. Як це зробити, пояснюється в уроці 15. Всі елементи вікна `Frame` викresлюються графічною оболонкою операційної системи по правилах цієї оболонки. Вікно `Frame` автоматично реєструється у віконному менеджері графічної оболонки і може переміщаться, змінювати розміри, звертаться в панель завдань (`task bar`) за допомогою миші або клавіатури, як "рідне" вікно операційної системи. Створити вікно типу `Frame` можна наступними конструкторами:

- `Frame()` — створює вікно з пустим рядком заголовка;
- `Frame(IString title)` — записує аргумент `title` в рядок заголовка.

Методи класу `Frame` здійснюють доступ до елементів вікна, але не забувайте, що клас `Frame` наслідує близько двохсот методів класів `Component`, `Container` і `Window`. Зокрема, наслідується менеджер розміщення по замовчуванню — `BorderLayout`.

Події

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при зміні розмірів вікна, його переміщенні або видаленні з екрана, а також показу на екрані відбувається подія `windowEvent`.

Програма лістингу 10.5 створює два вікна типу `Frame`, в які поміщаються рядки — мітки `Label`. При закритті основного вікна кліком по відповідній кнопці в рядку заголовка або комбінацією клавіш `<Alt>+<F4>` виконання програми завершується зверненням до методу `system.exit (0)`, і закриваються обидва вікна. При закритті другого вікна відбувається звернення до методу `dispose()`, і закривається тільки це вікно.

Лістинг 10.5. Створення двох вікон

```
import java.awt.* ;
import java.awt.event.*;
class TwoFrames{
public static void main(String[] args){
Fr1 f1 = new Fr1(" Головне вікно");
Fr2 f2 = new Fr2(" Друге вікно");
}
class Fr1 extends Frame{
Fr1(String s){
super(s);
setLayout(null);
Font f = new Font("Serif", Font.BOLD, 15);
setFont(f);
Label l = new Label("This is the main window", Label.CENTER);
l.setBounds(10, 30, 180, 30);
add(l);
setSize(200, 100);
setVisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit (0);
}
});
}
class Fr2 extends Frame{ Fr2(String s){
super(s);
setLayout(null) ;
```

```

Font f = new Font("Serif", Font.BOLD, 15);
setFont(f);
Label l = new Label("This is the second window", Label.CENTER);
l.setBounds(10, 30, 180, 30);
add(l);
setBounds(50, 50, 200, 100);
setVisible(true);
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev) {
dispose ();
}
});
}
}
}

```

На рис. 10.5 показано виведення цієї програми. Взаємне положення вікон визначається віконним менеджером операційної системи і може бути не таким, яке показане на рисунку.

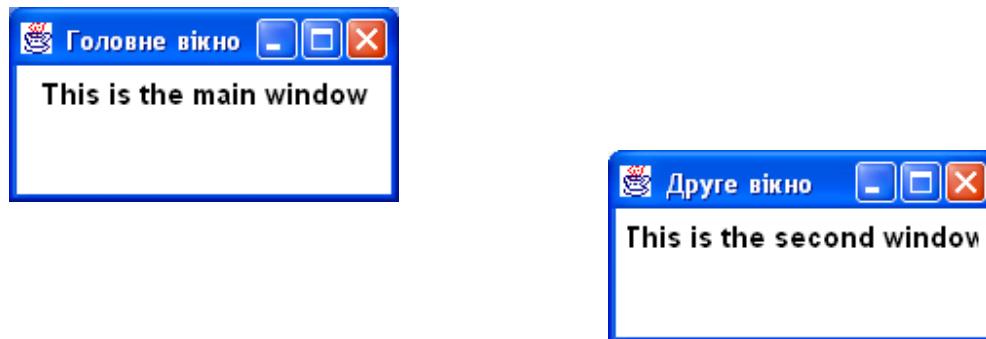


Рис. 10.5. Програма з двома вікнами

10.19. Контейнер Dialog

Контейнер [Dialog](#) — це вікно фіксованого розміру, призначене для відповіді на повідомлення додатку. Воно автоматично реєструється у віконному менеджері графічної оболонки, тому його можна переміщати по екрані, змінювати його розміри. Але вікно типу [Dialog](#), як і його суперклас — вікно типу [Window](#), — обовязково має власника [owner](#), який указується в конструкторі. Вікно типу [Dialog](#) може бути [модальним \(modal\)](#), в якому треба обовязково виконати всі передбачені дії, інакше із вікна не можна буде вийти. В класі сім конструкторів. Із них:

- [Dialog \(Dialog owner\)](#) — створює немодальне діалогове вікно з пустим рядком заголовка;
- [Dialog \(Dialog owner, string title\)](#) — створює немодальне діалогове вікно з рядком заголовка [title](#);
- [Dialog\(Dialog owner, String title, boolean modal\)](#) — створює діалогове вікно, яке буде модальним, якщо [modal == true](#).

Чотири інших конструктори аналогічні, але створюють діалогові вікна, належні вікну типу [Frame](#):

- [Dialog\(Frame owner\)](#)
- [Dialog\(Frame owner, String title\)](#)
- [Dialog\(Frame owner, boolean modal\)](#)
- [Dialog\(Frame owner, String title, Boolean modal\)](#)

Серед методів класу цікаві методи: [isModal\(\)](#), перевіряючий стан модальності, і [setModal\(boolean modal\)](#), змінюючий цей стан.

[Події](#)

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при зміні розмірів вікна, його переміщенні або видаленні з екрана, а також показу на екрані відбувається подія `windowEvent`.

В лістинзі 10.6. створюється модальне вікно доступу, в яке вводиться ім'я і пароль. Поки не буде зроблено правильне введення, інші дії неможливі. На рис. 10.6 показано вигляд цього вікна.

Лістинг 10.6. Модальне вікно доступу

```
import java.awt.*;
import java.awt.event.*;
class LoginWin extends Dialog{
LoginWin(Frame f, String s){
super(f, s, true);
setLayout(null);
setFont(new Font("Serif", Font.PLAIN, 14));
Label l1 = new Label("Your Name:", Label.RIGHT);
l1.setBounds(20, 30, 70, 25); add(l1);
Label l2 = new Label("Password:", Label.RIGHT);
l2.setBounds(20, 60, 70, 25); add(l2);
TextField tf1 = new TextField(30);
tf1.setBounds(100, 30, 160, 25); add(tf1);
TextField tf2 = new TextField(30);
tf2.setBounds(100, 60, 160, 25); add(tf2);
tf2.setEchoChar('*');
Button b1 = new Button("Hello");
b1.setBounds(50, 100, 100, 30); add(b1);
Button b2 = new Button("Cancel");
b2.setBounds(160, 100, 100, 30); add(b2);
setBounds(50, 50, 300, 150); } }
class DialogTest extends Frame{ DialogTest(String s){ super(s);
setLayout(null); setSize(200, 100);
setVisible(true);
Dialog d = new LoginWin(this, " Вікно входу"); d.setVisible(true);
}
public static void main(String[] args){
Frame f = new DialogTest("Owner window");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

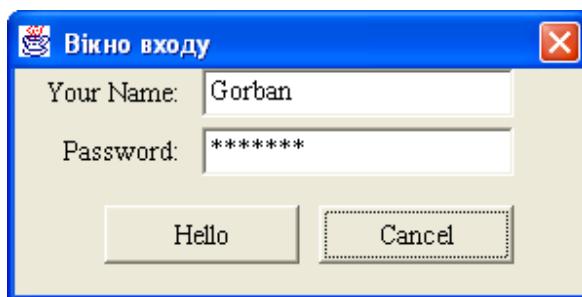


Рис. 10.6. Модальне вікно доступу

10.20. Контейнер `FileDialog`

Контейнер `FileDialog` — це модальне вікно з власником типу `Frame`, що містить стандартне вікно вибору файла операційної системи для відкриття (константа `LOAD`) або збереження (константа `SAVE`). Вікна операційної системи створюються і поміщаються в об'єкт класу `FileDialog` автоматично. В класі три конструктори:

- `FileDialog (Frame owner)` — створює вікно з пустим заголовком для відкриття файлів;
- `FileDialog (Frame owner, String title)` — створює вікно для відкриття файлів із заголовком `title`;
- `FileDialog(Frame owner, String title, int mode)` — створює вікно для відкриття або збереження документа, аргумент `mode` має два значення: `FileDialog.LOAD` і `FileDialog.SAVE`.

Методи класу `getDirectory()` і `getFile()` повертають тільки вибраний каталог і ім'я файла у вигляді рядка `String`. Завантаження або збереження файлу потім треба виконувати методами класів введення/виведення, як розповідається в уроці 18, там же приведені приклади використання класу `FileDialog`. Можна установити початковий каталог для пошуку файла і ім'я файла методами `setDirectory(String dir)` і `setFile(String fileName)`. Замість конкретного імені файла `fileName` можна написати шаблон, наприклад, `*.java` (перші символи — зірочка і точка), тоді у вікні будуть видимі тільки імена файлів, що закінчуються точкою і словом `java`. Метод `setFilenameFilter(FilenameFilter filter)` установлює шаблон `filter` для імені вибраного файла. У вікні будуть видимі тільки імена файлів, що підходять під шаблон. Цей метод не реалізований в `SUN JDK` на платформі `MS Windows`.

Події

Крім подій класу `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, при зміні розмірів вікна, його переміщенні або видаленні з екрана, а також показу на екрані відбувається подія `windowEvent`.

10.21. Створення власних компонентів

Створити свій компонент, доповнюючий властивості і методи уже існуючих компонентів `AWT`, дуже просто — треба лише створити свій клас як розширення існуючого класу `Button`, `TextField` або іншого класу-компоненту. Якщо треба скомбінувати декілька компонентів в один, новий, компонент, то достатньо розширити клас `Panel`, розташувавши компоненти на панелі. Якщо ж треба створити цілком новий компонент, то `AWT` пропонує дві можливості: створити "важкий" або "легкий" компонент. Для створення власних "важких" компонентів у бібліотеці `AWT` єсть клас `Canvas` — пустий компонент, для якого створюється свій `peer`-об'єкт графічної системи.

10.22. Компонент `Canvas`

Компонент `Canvas` — це пустий компонент. Клас `Canvas` дуже простий — в ньому тільки конструктор по замовчуванню `Canvas()` і пуста реалізація методу `paint(Graphics g)`. Щоб створити свій "важкий" компонент, необхідно розширити клас `Canvas`, доповнивши його потрібними полями і методами, і при необхідності перевизначити метод `paint()`. Наприклад, як ви помітили, на стандартній кнопці `Button` можна написати тільки один текстовий рядок. Не можна написати декілька рядків або відобразити на кнопці рисунок. Створимо свій "важкий" компонент — кнопку з рисунком.

В лістингі 10.7 кнопка з рисунком — клас `FlowerButton`. Рисунок задається методом `drawFlower()`, а рисується методом `paint()`. Метод `paint()`, крім того, креслить по краях кнопки внизу і справа відрізки прямих, зображені тінь, відкинутою "випуклою" кнопкою. При натисканні кнопки миші на компоненті такі ж відрізки кресляться зверху і зліва — кнопка "вдавилася". При цьому рисунок здвигається на два пікселя вправо вниз — він "вдавлюється" в площину вікна. Крім цього, в класі `FlowerButton` задана реакція на натискання і відпускання кнопки миші. Це ми обговоримо в уроці 12, а поки що скажемо, що при кожному натисканні і відпусканні кнопки змінюється значення поля `isDown` і кнопка перекреслюється методом `repaint()`. Це досягається виконанням методів `mousePressed()` і `mouseReleased()`. Для порівняння рядом розміщена стандартна кнопка типу `Button` того ж розміру. Рис. 10.7 демонструє вигляд цих кнопок.

Лістинг 10.7. Кнопка з рисунком

```
import java.awt.*;
import java.awt.event.*;
class FlowerButton extends Canvas implements MouseListener{
```

```

private boolean isDown=false;
public FlowerButton(){
super();
setBackground(Color.lightGray);
addMouseListener(this);
}
public void drawFlower(Graphics g, int x, int y, int w, int h){
g.drawOval(x + 2*w/5 - 6, y, w/5, w/5);
g.drawLine(x + w/2 - 6, y + w/5, x + w/2 - 6, y + h - 4);
g.drawOval(x + 3*w/10 - 6, y + h/3 - 4, w/5, w/5) ;
g.drawOval(x + w/2 - 6, y + h/3 - 4, w/5, w/5); }
public void paint(Graphics g){
int w = getSize().width, h = getSize().height;
if (isDown){
g.drawLine(0, 0, w - 1, 0) ;
g.drawLine(0, 1, w - 1, 1);
g.drawLine(0, 0, 0, h - 1);
g.drawLine (1, 1, 1, h - 1);
drawFlower(g, 8, 10, w, h);
}
else
{
g.drawLine(0, h - 2, w - 2, h - 2);
g.drawLine(0, h - 1, w - 1, h - 1);
g.drawLine(w - 2, h - 2, w - 2, 0);
g.drawLine(w - 1, h - 1, w - 1, 1);
drawFlower (g, 6, 8, w, h) ; } }
public void mousePressed(MouseEvent e){
isDown=true; repaint(); }
public void mouseReleased(MouseEvent e){
isDown=false; repaint(); }
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e) {}}
public void mouseClicked(MouseEvent e){}
}
class DrawButton extends Frame{
DrawButton(String s) {
super (s) ;
setLayout(null);
Button b = new Button("OK");
b.setBounds(200, 50, 100, 60); add(b);
FlowerButton d = new FlowerButton();
d.setBounds(50, 50, 100, 60); add(d);
setSize(400, 150);
setVisible(true);
}
public static void main(String[] args){
Frame f= new DrawButton(" Кнопка с рисунком");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
}); }
}

```

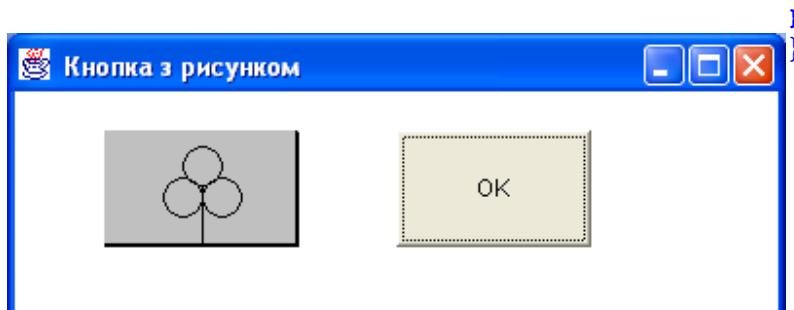


Рис. 10.7. Кнопка з рисунком

10.23. Створення "легкого" компонента

"Легкий" компонент, який не має свого *peer*-обєкта в графічній системі, створюється як пряме розширення класу *Component* або *Container*. При цьому необхідо задати ті дії, які у "важких" компонентах виконує *peer*-обєкт. Наприклад, замінивши в лістингі 10.7 заголовок класу *FlowerButton* рядком

```
class FlowerButton extends Component implements MouseListener
```

а потім перекомпілювавши і виконавши програму, ви одержите "легку" кнопку, але побачите, що її фон став білим, тому що метод

```
setBackground(Color.lightGray)
```

не спрацював. Це пояснюється тим, що тепер всю чорну роботу по зображенням кнопки на екрані виконує не *peer*-двійник кнопки, а "важкий" контейнер, в якому розташована кнопка, в нашому випадку клас *Frame*. Контейнер же нічого не знає про те, що треба звернутися до методу *setBackground()*, він рисує тільки те, що записано в методі *paint()*. Прийдеться прибрести метод *setBackground()* із конструктора і заливати фон сірим кольором вручну в методі *paint()*, як показано в лістингі 10.8. "Легкий" контейнер не уміє рисувати свої "легкі" компоненти, тому в кінці методу *paint()* "легкого" контейнера треба звернутися до методу *paint()* суперкласу: *super.paint(g)*. Тоді рисуванням займеться "важкий" суперклас-контейнер. Він нарисує і свій "легкий" контейнер, і розміщені в контейнері "легкі" компоненти.

Порада

Завершуйте метод *paint()* "легкого" контейнера зверненням до методу *paint()* суперкласу. Оптимальний розмір "важкого" компонента установлюється *peer*-обєктом, а для "легких" компонентів його треба задати явно, перевизначивши метод *getPreferredSize()*, інакше деякі менеджери розміщення, наприклад *FlowLayout()*, установлять нульовий розмір, і компонент не буде видно на екрані.

Порада

Перевизначайте метод *getPreferredSize()*.

Цікава особливість "легких" компонентів — вони із начально рисуються прозорими, не зафарбована частина прямокутного обєкта не буде видима. Це дозволяє створити компонент будь-якої видимої форми. Лістинг 10.8 показує, як можна змінити метод *paint()* лістинга 10.7 для створення круглої кнопки і задати додаткові методи, а рис. 10.8 демонструє її вигляд.

Лістинг 10.8. Створення круглої кнопки ;

```
import java.awt.*;
import java.awt.event.*;
class FlowerButton extends Canvas implements MouseListener{
private boolean isDown=false;
public FlowerButton(){
super();
addMouseListener(this);
}
public void drawFlower(Graphics g, int x, int y, int w, int h){
g.drawOval(x + 2*w/5 - 6, y, w/5, w/5);
```

```

g.drawLine(x + w/2 - 6, y + w/5, x + w/2 - 6, y + h - 4);
g.drawOval(x + 3*w/10 - 6, y + h/3 - 4, w/5, w/5) ;
g.drawOval(x + w/2 - 6, y + h/3 - 4, w/5, w/5); }
public void paint(Graphics g){
int w = getSize().width, h = getSize().height;
int d = Math.min(w, h); // Діаметр круга
Color c = g.getColor(); // Зберігаємо поточний колір
g.setColor(Color.lightGray); // Установлюємо сірий колір
g.fillArc(0, 0, d, d, 0, 360); // Заливаємо круг сірим кольором
g.setColor(c); // Відновлюємо поточний колір
if (isDown){
g.drawArc(0, 0, d, d, 43, 180);
g.drawArc(0, 1, d - 2, d - 2, 43, 180);
drawFlower(g, 8, 10, d, d);
}else{
g.drawArc(0, 0, d, -d, 229, 162);
g.drawArc(0, 1, d - 2, d - 2, 225, 170);
drawFlower(g, 6, 8, d, d);
}
}
public Dimension getPreferredSize(){
return new Dimension(30,30);
}
public Dimension getMinimumSize()
{
return getPreferredSize(); }
public Dimension getMaximumSize(){
return getPreferredSize(); }
}
public void mousePressed(MouseEvent e){
isDown=true; repaint(); }
public void mouseReleased(MouseEvent e){
isDown=false; repaint(); }
public void mouseEntered(MouseEvent e){ }
public void mouseExited(MouseEvent e) { }
public void mouseClicked(MouseEvent e){ }
}
class DrawButton extends Frame{
DrawButton(String s) {
super (s) ;
setLayout(null);
Button b = new Button("OK");
b.setBounds(200, 50, 100, 60); add(b);
FlowerButton d = new FlowerButton();
d.setBounds(50, 50, 100, 60); add(d);
setSize(400, 150);
setVisible(true);
}
public static void main(String[] args){
Frame f= new DrawButton(" Кнопка з рисунком");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
}
);
}
}

```

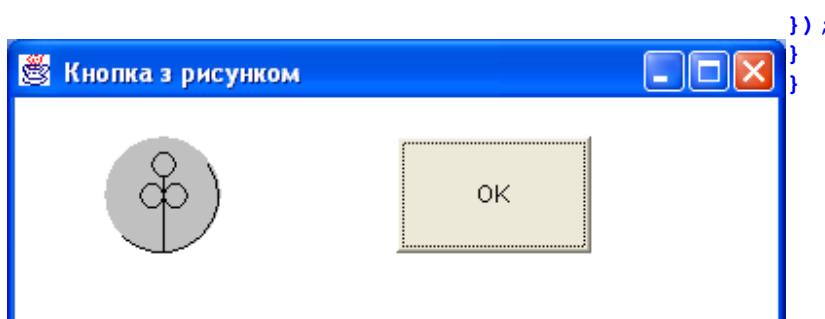


Рис. 10.8. Кругла кнопка

Зразу ж треба дати ще одну рекомендацію. "Легкі" контейнери не займаються обробкою подій без спеціальної вказівки. Тому в конструктор "легкого" компонента належить включити звернення до методу `enableEvents()` для кожного типу подій. В нашому прикладі в конструктор класу `FlowerButton` корисно додати рядок

```
enableEvents(AWTEvent.MOUSE_EVENT_MASK);
```

на випадок, якщо кнопка виявиться в "легкому" контейнері. Детальніше про це ми поговоримо в уроці 12. В документації єсть хороши прилади створення "легких" компонентів, подивіться сторінку [docs\guide\awt\demos\lightweight\index.html](http://docs.oracle.com/javase/tutorial/uiswing/components/index.html).

Лабораторна робота 9. Властивості компонентів

1. Ви вже розібрали програми лістингів 10.7 і 10.8 і випробували їх. Це перші програми, які реагують на кліки мишкою по кнопках. Заставте цю програму робити щось інше, при натисканні на кнопки. Наприклад, змінювати колір компонентів, виводити текст у вікно чи в компонент, додавати нові компоненти і т. п. Переробіть круглу кнопку в овальну. Будьте готові при здачі лабораторної роботи до того, що викладач заставить вас замість овальної кнопки зробити трикутну, пятикутну, шестикутну. А також скаже, що повинна зробити програма при натисканні на кнопку.

Програмування у Java

Урок 11. Розміщення компонентів

- [Менеджер FlowLayout](#)
- [Менеджер BorderLayout](#)
- [Менеджер GridLayout](#)
- [Менеджер CardLayout](#)
- [Менеджер GridBagLayout](#)
- [Заключення](#)

В попередньому уроці ми розміщували компоненти "вручну", задаючи їх розміри і положення в контейнері абсолютною координатами в координатній системі контейнера. Для цього ми застосовували метод `setBounds()`. Такий спосіб розміщує компоненти з точністю до пікселя, але не дозволяє переміщати їх. При зміні розмірів вікна за допомогою миші компоненти залишаться на своїх місцях, привязаними до лівого верхнього кута контейнера. Крім того, немає гарантії, що всі монстори відобразять компоненти так, як ви задумали. Щоб врахувати зміну розмірів вікна, треба задати розміри і положення компонента відносно розмірів контейнера, наприклад, так:

```
int w = getSize().width; // Одержано ширину
int h = getSize().height; // висоту контейнера
Button b = new Button("OK"); // Створюємо кнопку
b.setBounds(9*w/20, 4*h/5, w/10, h/10);
```

і при всякій зміні розмірів вікна задавати розташувння компонента заново. Щоб позбавити програміста від цієї кропіткої роботи, в бібліотеку AWT внесені два інтерфейси: `LayoutManager` і породжений від нього інтерфейс `LayoutManager2`, а також п'ять реалізацій цих інтерфейсів: класи `BorderLayout`, `CardLayout`, `FlowLayout`, `GridLayout` і `GridBagLayout`. Ці класи назовані менеджерами розміщення (*layout manager*) компонентів. Кожний програміст може створити свої менеджери розміщення, реалізувавши інтерфейси `LayoutManager` або `LayoutManager2`. Подивимося, як розміщують компоненти ці класи.

11.1. Менеджер FlowLayout

Найбільш просто поступає менеджер розміщення `FlowLayout`. Він укладає в контейнер один компонент за другим зліва направо як цеглини, переходячи від верхніх рядів до нижніх. При зміні розміру контейнера "цеглини" перестрояються, як показано на рис. 11.1. Компоненти поступають в тому порядку, в якому вони задані в методах `add()`. В кожному ряді компоненти можуть притискуватися до лівого краю, якщо в конструкторі аргумент `align` рівний `FlowLayout.LEFT`, до правого краю, якщо цей аргумент `FlowLayout.RIGHT`, або збиратися в середині ряду, якщо `FlowLayout.CENTER`. Між компонентами можна залишити проміжки (`gap`) по горизонталі `hgap` і вертикальні `vgap`. Це задається в конструкторі:

`FlowLayout(int align, int hgap, int vgap)`

Другий конструктор задає проміжки розміром 5 пікселів: `FlowLayout(int align)`

Третій конструктор визначає вирівнювання по центру і проміжки 5 пікселів: `FlowLayout()`

Після формування об'єкта ці параметри можна змінити методами: `setHgap(int hgap)`, `setVgap(int vgap)`, `setAlignment(int align)`

В лістингі 11.1 створюється кнопка `Button`, мітка `Label`, кнопка вибора `Checkbox`, розкриваючийся список `Choice`, поле введення `TextField` і все це розміщується в контейнері `Frame`. Рис. 11.1 містить вигляд цих компонентів при різних розмірах контейнера.

Лістинг 11.1. Менеджер розміщення FlowLayout

```
import java.awt.*;
import java.awt.event.*;
class FlowTest extends Frame{
```

```

FlowTest(String s) {
super(s);
setLayout (new FlowLayout (FlowLayout.LEFT, 10, 10));
add(new Button("Button"));
add(new Label("Label"));
add(new Checkbox("Choice"));
add(new Choice());
add(new TextField("Help", 6));
setSize(300, 100); setVisible(true);
}
public static void main(String[] args){
Frame f= new FlowTest(" Менеджер FlowLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
}

```

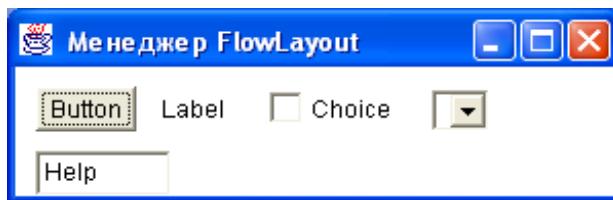


Рис. 11.1. Розміщення компонентів за допомогою `FlowLayout`

11.2. Менеджер `BorderLayout`

Менеджер разміщення `BorderLayout` ділить контейнер на п'ять нерівних областей, повністю заповнюючи кожну область одним компонентом, як показано на рис. 11.2. Області отримали географічні назви `NORTH`, `SOUTH`, `WEST`, `EAST` і `CENTER`. Метод `add()` у випадку застосування `BorderLayout` має два аргументи: посилку на компонент `comp` і область `region`, в яку поміщається компонент — одну із перечислених вище констант: `add(Component comp, String region)`. Метод `add (Component comp)` з одним аргументом поміщає компонент в область `CENTER`. В класі два конструктори:

- `BorderLayout()` — між областями немає проміжків;
- `BorderLayout(int hgap int vgap)` — між областями залишається горизонтальні `hgap` і вертикальні `vgap` проміжки, задані в пікселях.

Якщо в контейнер поміщається менше пяти компонентів, то деякі області не використовуються і не займають місце в контейнері, як можна помітити на рис. 11.3. Якщо не зайнята область `CENTER`, то компоненти притискаються до границь контейнера.

В лістингі 11.2 створюються п'ять кнопок, розміщених в контейнері. Замітьте відсутність установки менеджера в контейнері `setLayout()` — менеджер `BorderLayout` установлений в контейнер `Frame` по замовчуванню. Результат розміщення показаний на рис. 11.2.

Лістинг 11.2. Менеджер розміщення `BorderLayout`

```

import java.awt.*;
import java.awt.event.*;
class BorderTest extends Frame{
BorderTest(String s){ super(s);
add(new Button("North"), BorderLayout.NORTH);
add(new Button("South"), BorderLayout.SOUTH);
add(new Button("West"), BorderLayout.WEST);
}
}

```

```

add(new Button("East"), BorderLayout.EAST);
add(new Button("Center"));
setSize(300, 200);
setVisible(true);
}
public static void main(String[] args){
Frame f= new BorderTest(" Менеджер BorderLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
}

```

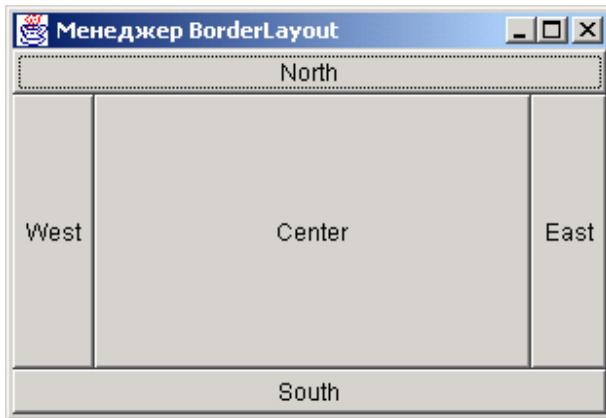


Рис. 11.2. Області розміщення BorderLayout

Менеджер розміщення [BorderLayout](#) здається незручним: він має не більше пяти компонентів, останні розтікаються по всій області, області мають дивний вигляд. Але справа в тому, що в кожну область можна помістити не компонент, а панель, і розміщувати компоненти на ній, як зроблено в лістингі 11.3 і показано на рис. 11.3. Нагадаємо, що на панелі [Panel](#) менеджер розміщення по замовчуванню [FlowLayout](#).

Лістинг 11.3. Складне компонування

```

import java.awt.*;
import java.awt.event.*;
class BorderPanelTest extends Frame{
BorderPanelTest(String s){
super(s);
// Створюємо панель p2 з трьома кнопками
Panel p2 = new Panel();
p2.add(new Button("Execute"));
p2.add(new Button("Cancel"));
p2.add(new Button("Exit"));
Panel p1 = new Panel();
p1.setLayout(new BorderLayout());
// Поміщаємо панель p2 з кнопками на "півдні" панелі p1
p1.add(p2, BorderLayout.SOUTH);
// Поле введення поміщаємо на "півночі"
p1.add(new TextField("Input Field", 20), BorderLayout.NORTH);
// Область введення поміщається в центрі
p1.add(new TextArea("Input Area", 5, 20, TextArea.SCROLLBARS_NONE),
BorderLayout.CENTER);

```

```

add(new Scrollbar(Scrollbar.HORIZONTAL), BorderLayout.SOUTH);
add(new Scrollbar(Scrollbar.VERTICAL), BorderLayout.EAST);
// Панель p1 поміщаємо в "центрі" контейнера
add(p1, BorderLayout.CENTER);
setSize(200, 200);
setVisible(true);
}

public static void main(String[] args) {
Frame f= new BorderPanelTest(" Складне компонування");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
}

```

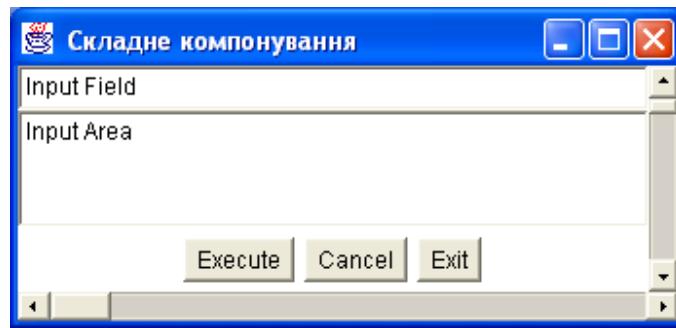


Рис. 11.3. Компонування за допомогою `FlowLayout` і `BorderLayout`

11.3. Менеджер `GridLayout`

Менеджер розміщення `GridLayout` розставляє компоненти в таблицю із заданим в конструкторі числом рядків `rows` і стовпців `columns`:

`GridLayout(int rows, int columns).`

Всі компоненти отримують одинаковий розмір. Проміжків між компонентами немає. Другий конструктор дозволяє задати проміжки між компонентами в пікселях по горизонталі `hgap` і вертикальні `vgap`:

`GridLayout(int rows, int columns, int hgap, int vgap)`

Конструктор по заочуванню `GridLayout()` задає таблицю розміром 0x0 без проміжків між компонентами. Компоненти будуть розташовуватися в одному рядку. Компоненти розміщаються менеджером `GridLayout` зліва направо по рядкам створеної таблиці в тому порядку, в якому вони задані в методах `add()`. Нульова кількість рядків або стовпців означає, що менеджер сам створить потрібне їх число. В лістингі 11.4 розміщаються кнопки для калькулятора, а рис. 11.4 показує, як виглядає це розміщення.

Лістинг 11.4. Менеджер `GridLayout`

```

import java.awt.*;
import java.awt.event.*;
import java.util.*;
class GridTest extends Frame{
GridTest(String s){ super(s);
setLayout(new GridLayout(4, 4, 5, 5));
StringTokenizer st =
new StringTokenizer("7 8 9 / 4 5 6 * 1 2 3 - 0 . = + ");

```

```
while(st.hasMoreTokens())
add(new Button(st.nextToken()));
setSize(200, 200); setVisible(true);
}
public static void main(String[] args){
Frame f= new GridLayout(" Менеджер GridLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

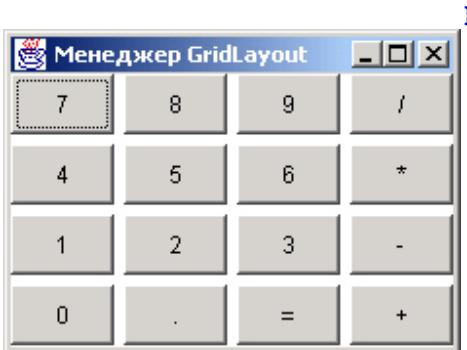


Рис. 11.4. Розміщення кнопок менеджером GridLayout

11.4. Менеджер Card Layout

Менеджер розміщення `CardLayout` своєрідний - він показує в контейнері тільки один, перший (`first`), компонент. Решта компонентів лежить під першим у певному порядку як гральні карти в колоді. Їх розташування визначається порядком, в якому написані методи `add()`. Наступний компонент можна показати методом `next (Container c)`, попередній - методом `previous (Container c)`, останній - методом `last (Container c)`, перший - методом `first (Container c)`. Аргумент цих методов - посилка на контейнер, в який поміщені компоненти, звичайно `this`. В класі два конструктори:

- `CardLayout()` — не відділяє компонент від границь контейнера;
 - `CardLayout(int hgap, int vgap)` — задає горизонтальне `hgap` і вертикальне `vgap` поля.

Менеджер `CardLayout` дозволяє організувати і довільний доступ до компонентів. Метод `add()` для менеджера `CardLayout` має своєрідний вигляд:

add(Component comp, Object constraints)

Тут аргумент `constraints` повинен мати тип `String` і містити ім'я компонента. Потрібний компонент з іменем `name` можна показати методом:

show(Container parent, String name)

В лістинзі 11.5 менеджер розміщення `c1` працює з панеллю `p`, поміщеної в "центр" контейнера `Frame`. Панель `p` указується як аргумент `parent` в методах `next()` і `show()`. На "північ" контейнера `Frame` відправлена панель `p2` з міткою в розкриваючому списку `ch`. Рис. 11.5 демонструє результат роботи програми.

Лістинг 11.5. Менеджер CardLayout

```
import java.awt.*;
import java.awt.event.*;
class CardTest extends Frame{ CardTest(String s){
```

```
super(s);
Panel p = new Panel();
CardLayout cl = new CardLayout();
p.setLayout(cl);
p.add(new Button("Ukrainian page"), "page1");
p.add(new Button("English page"), "page2");
p.add(new Button("German page"), "page3");
add(p);
cl.next(p);
cl.show(p, "page1");
Panel p2 = new Panel();
p2.add(new Label("Choice language:"));
Choice ch = new Choice();
ch.add("Ukrainian");
ch.add("English");
ch.add("German");
p2.add(ch);
add(p2, BorderLayout.NORTH);
setSize(400, 300);
setVisible(true); }
public static void main(String[] args){
Frame f= new CardTest(" Менеджер CardLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

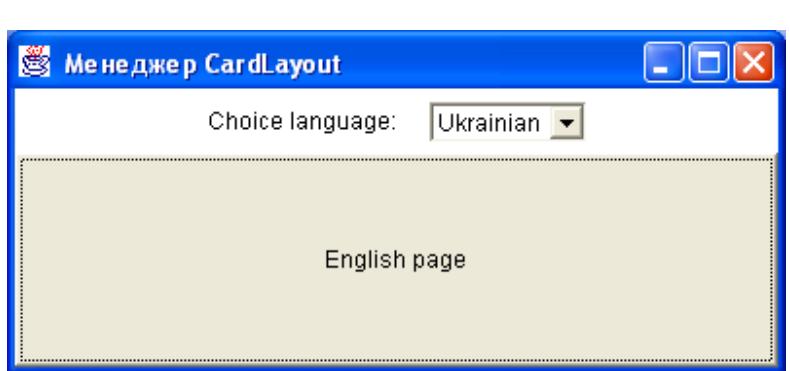


Рис. 11.5. Менеджер розміщення CardLayout

11.5. Менеджер GridBagLayout

Менеджер розміщення [GridBagLayout](#) розставляє компоненти найбільш гнучко, дозволяючи задавати розміри і положення кожного компонента. Але він виявився досить складним і застосовуються рідко. В класі [GridBagLayout](#) є тільки один конструктор по замовчуванню, без аргументів. Менеджер класу [GridBagLayout](#), на відміну від інших менеджерів розміщення, не містить правил розміщення. Він відіграє тільки організуючу роль. Йому передаються посилання на компонент і правила розташування цього компонента, а сам він поміщає даний компонент по вказаним правилам у контейнер. Всі правила розміщення компонентів задаються в обєкті іншого класу, [GridBagConstraints](#). Менеджер розміщує компоненти в таблиці з невизначеним задалегідь числом рядків і стовпців. Один компонент може займати декілька клітинок цієї таблиці, заповнювати клітинку цілком, розташовуватися в її центрі, куті або притискується до краю клітинки. Клас [GridBagConstraints](#) містить одинадцять полів, визначаючих розміри компонентів, їх положення в контейнері і взаємне положення, і декілька констант - значень деяких полів. Вони перечислені в табл. 11.1. Ці параметри визначаються конструктором, що має одинадцять аргументів. Другий конструктор — конструктор по замовчуванню — присвоює параметрам значення,

задані по замовчуванню.

Таблиця 11.1. Поля класу *GridBagConstraints*

Поле	Значення
anchor	Напрям розміщення компонента в контейнері. Константи: CENTER, NORTH, EAST, NORTHEAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, і NORTHWEST; по замовчуванню CENTER
fill	Розтягування компонента для заповнення клітинки. Константи: NONE, HORIZONTAL, VERTICAL, BOTH; по умолчанию NONE
gridheight	Кількість клітинок в колонці, зайнятих компонентом. Ціле типу int, по замовчуванню 1. Константа REMAINDER означає, що компонент займе решту колонки, RELATIVE — буде наступним по порядку в колонці
gridwidth	Кількість клітинок і рядку, зайнятих компонентом. Ціле типу int, по замовчуванню 1. Константа REMAINDER означає, що компонент займе решту рядка, RELATIVE — буде наступним в рядку по порядку
gridx	Номер клітинки в рядку. Сама ліва клітинка має номер 0. По замовчуванню константа RELATIVE, що означає: наступна по порядку
gridy	Номер клітинки в стовпці. Сама верхня клітинка має номер 0. По замовчуванню константа RELATIVE, що означає: наступна по порядку
insets	Поля в контейнері. Об'єкт класу insets; по замовчуванню об'єкт з нулями
ipadx, ipady	Горизонтальные и вертикальные поля вокруг компонентов; по замовчуванню 0
weightx, weighty	Пропорційний розтяг компонентів при зміні розміру контейнера; по замовчуванню 0,0

Як правило, об'єкт класу *GridBagConstraints* створюється конструктором по замовчуванню, потім значення потрібних полів змінюються простим присвоєнням нових значень, наприклад:

```
GridBagConstraints gbc = new GridBagConstraints();
gbc.weightx = 1.0;
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.gridheight = 2;
```

Після створення об'єкта *gbc* класу *GridBagConstraints* менеджеру розміщення указується, що при поміщенні компонента *comp* в контейнер належить застосовувати правила, занесені в об'єкт *gbc*. Для цього застосовується метод

```
add(Component comp, GridBagConstraints gbc)
```

Отже, схема застосування менеджера *GridBagLayout* така:

Лістинг 11.6. Менеджер *GridBagLayout*

```
import java.awt.*;
import java.awt.event.*;
class GridBagTest extends Frame{ GridBagTest(String s){
super(s);
GridBagLayout gbl = new GridBagLayout();
```

```
setLayout(gbl);
GridBagConstraints c = new GridBagConstraints();
Button b1 = new Button("Button1");
c.gridwidth = 2;
add(b1,c);
Button b2 = new Button("Button2");
c.gridwidth = 1;
add(b2,c);
setSize(400, 300);
setVisible(true); }

public static void main(String[] args){
Frame f= new GridBagTest(" Менеджер GridBagLayout");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
}
```

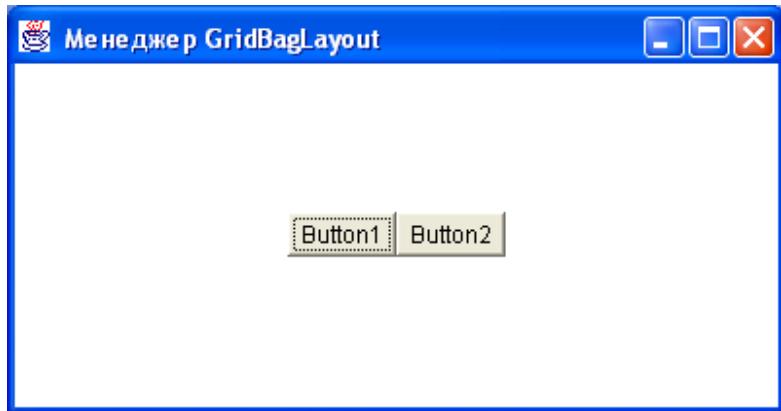


Рис. 11.6. Менеджер GridBagLayout

В документації до класу [GridBagLayout](#) приведений хороший приклад використання цього менеджера розміщення.

Заключення

Всі менеджери розміщення написані повністю мовою **Java**, в склад **SUN J2SDK** входять їх вихідні тексти. Якщо ви вирішили написати свій менеджер розміщення, реалізувавши інтерфейс **LayoutManager** або **LayoutManager2**, то подивітесь ці вихідні тексти.

Лабораторна робота 10. Створення повноцінної Java програми з графічним інтерфейсом.

1. Візьміть за основу програму лістингу 11.4. Додайте в форму текстове вікно і запрограмуйте повноцінний калькулятор.
 2. Додайте у форму ще декілька кнопок і запрограмуйте обчислення тригонометричних, обернених тригонометричних і інших елементарних функцій. Будьте готові до того, що викладач при прийомі роботи запропонує додати ще якусь кнопку - функцію.

Програмування у Java

Урок 12. Обробка подій

- Подія **ActionEvent**
 - Обробка дій миші
 - Класи-адаптери
 - Обробка дій клавіатури
 - Подія **TextEvent**
 - Обробка дій з вікном
 - Подія **ComponentEvent**
 - Подія **ConainerEvent**
 - Подія **FocusEvent**
 - Подія **ItemEvent**
 - Подія **AdjustmentEvent**
 - Декілька слухачів одного джерела
 - Диспетчеризація подій
 - Створення власної події

В двох попередніх главах ми написали багато програм, створюючих інтерфейси, але, власне, інтерфейса, тобто взаємодії з користувачем, ці програми не забезпечують. Можна клікнути по кнопці на екрані, вона буде "вдавлюватися" в площину екрана, але більше нічого не буде відбуватися. Можна ввести текст в поле введення, але він не стане сприйматися і оброблятися програмою. Все це відбувається із-за того, що ми не задали обробку дій користувача, обробку подій.

12.1. Подія

Подія ([event](#)) в бібліотеці [AWT](#) виникає при дії на компонент якими-небудь маніпуляціями мишею, при введенні з клавіатури, при переміщенні вікна, зміні його розмірів. Об'єкт, в якому відбулася подія, називається джерелом ([source](#)) події. Всі події в [AWT](#) класифіковані. При виникненні події виконуюча система [Java](#) автоматично створює об'єкт відповідного події класу. Цей об'єкт не виконує ніяких дій, він тільки зберігає всі дані про події.

На чолі ієрархії класів-подій стоїть клас `EventObject` із пакета `java.util` — безпосереднє розширення класу `Object`. Його розширяє абстрактний клас `AWTEvent` із пакета `java.awt` — глава класів, описуючих події бібліотеки `AWT`. Подальша ієрархія класів-подій показана на рис. 12.1. Всі класи, відображені на рисунку, крім класу `AWTEvent`, зібрані в пакет `java.awt.event`. Події типу `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` виникають у всіх компонентах. А події типу `ContainerEvent` — тільки в контейнерах: `Container`, `Dialog`, `FileDialog`, `Frame`, `Panel`, `ScrollPane`, `Window`.

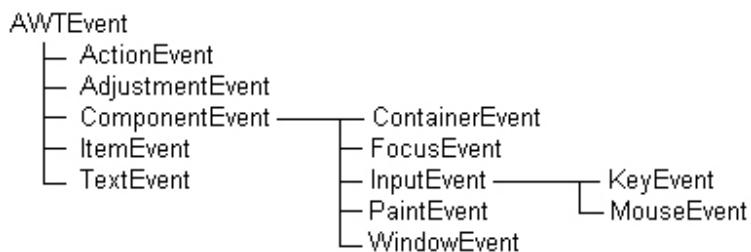


Рис. 12.1. Ієрархія класів, описуючих події AWT

Події типу `WindowEvent` виникають тільки у вікнах: `Frame`, `Dialog`, `FileDialog`, `Window`.

Події типу `TextEvent` генеруються тільки в контейнерах `Textcomponent`, `TextArea`, `TextField`.

Події типу `ActionEvent` проявляються тільки в контейнерах `Button`, `List`, `TextField`.

Події типу `ItemEvent` виникають тільки в контейнерах `Checkbox`, `Choice`, `List`.

Нарешті, події типу `AdjustmentEvent` виникають тільки в контейнері `Scrollbar`.

Парашт, подп. титу [Аддоменов](#) вийджає в Україні в комп'ютер [Согласія](#).

знати, в якому обєкт відбулася подія, можна методом `getTarget()` класу `EventTarget`. Цей метод

повертає тип **Object**. В кожному з цих класів-подій визначений метод **paramstring()**, повертуючий вміст обєкта даного класу у вигляді рядка **String**. Крім того, в кожному класі є свої методи, посталяючі ті чи інші відомості про події. Зокрема, метод **get()** повертає ідентифікатор (**identifier**) події — ціле число, що означає тип події. Ідентифікатори подій визначені в кожному класі-події як константи.

Методи обробки подій описані в інтерфейсах-слухачах (**listener**). Для кожного показаного на рис. 12.1 типу подій, крім **inputEvent** (ця подія рідко використовується самостійно), єсть свій інтерфейс. Імена інтерфейсів складаються із імені події і слова **Listener**, наприклад, **ActionListener**, **MouseListener**. Методи інтерфейса "слухають", що відбувається в потенційному джерелі події. При виникненні події ці методи автоматично виконуються, одержуючи в якості аргумента обєкт-подію і використовуючи при обробці відомості про події, що містяться в цьому обєкті.

Щоб задати обробку події певного типу, треба реалізувати відповідний інтерфейс. Класи, реалізуючі такий інтерфейс, класи-оброблювачі (**handlers**) події, називаються **слухачами** (**listeners**): вони "слухають", що відбувається в обєкті, щоб відслідкувати виникнення події і опрацювати її. Щоб звязати з обробчиком події, класи-джерела події повинні отримати посилку на екземпляр **eventHandler** класу-обробчика події одним із методів **addXxxListener(XxxEvent eventHandler)**, де **Xxx** — ім'я події. Такий спосіб реєстрації, при якому слухач залишає "візитну карточку" джерелу для свого виклику при виникненні події, називається зворотний виклик (**callback**). Ним часто користуються студенти, які, звонячи батькам і не бажаючи платити за телефонну розмову, говорять: "Передзвони мені по такому-то номеру". Зворотна дія — відмова від обробчика, переривання прослуховування — виконується методом **removeXxxListener()**. Таким чином, компонент-джерело, в якому відбулася подія, не займається його обробкою. Він звертається до екземпляра класу-слухача, уміючого обробляти події, **делегує** (**delegate**) йому повноваження по обробці.

Така схема отримала назву схеми **делегування** (**delegation**). Вона зручна тим, що ми можемо легко змінити клас-обробчик і опрацювати подію по-другому або призначити декілька оброблювачів однієї тієї ж події. З іншого боку, ми можемо один оброблювач призначити на прослуховування декількох обєктів-джерел подій. Ця схема здається занадто складною, але ми нею часто користуємося в житті. Допустимо, ми вирішили облаштувати квартиру. Ми поміщаємо в ній, як в контейнер, різні компоненти: меблі, сантехніку, електроніку, антикваріат. Ми вважаємо, що може відбутися неприємна подія — квартиру відвідають грабіжники, — і хочемо її опрацювати. Ми знаємо, що класи-оброблювачі цієї події — охоронні агентства, — і звертаємося до деякого екземпляра такого класу. Компонент-джерела події, тобто речі, які можуть бути украдені, приєднують до себе датчики методом **addXxxListener()**. Потім екземпляр оброблювача "слухає", що відбувається в обєктах, до яких він підключений. Він реагує на виникнення тільки однієї події — викрадення прослуховуваного обєкта, — інші події, наприклад, коротке замикання або прорив водопровідної труби, його не цікавлять. При виникненні "своєї" події він діє по контракту, записаному в методі опрацювання.

Зауваження

В JDK 1.0 була прийнята інша модель обробки подій. Не дивуйтесь, читаючи старі книги і проглядаючи вихідні тексти старих програм, але і не користуйтесь старою моделлю. Наведемо приклад. Нехай в контейнері типу **Frame** поміщено поле введення **tf** типу **TextField**, не редактувана область введення **ta** типу **TextArea** і кнопка **b** типу **Button**. В поле **tf** вводиться рядок, після натискання клавіші **<Enter>** або кліку кнопкою миші по кнопці **b** рядок переноситься в область **ta**. Після цього можна знову вводити рядок в поле **tf** і т. д.

Тут і при натисканні клавіші **<Enter>** і при кліку кнопкою миші виникає подія класу **ActionEvent**, причому вона може виникнути у двох компонентах-джерела: полі **tf** або кнопці **b**. Обробка події в обох випадках заключається в отриманні рядка тексту із поля **tf** (наприклад, методом **tf.getText()**) і розміщення його в область **ta** (скажімо, методом **ta.append()**). Значить, можна написати один оброблювач події **ActionEvent**, реалізувавши відповідний інтерфейс, який називається **ActionListener**. В цьому інтерфейсі всього один метод **actionPerformed()**. Отже, пишемо:

```
class TextMove implements ActionListener{
    private TextField tf;
    private TextArea ta;
    TextMove(TextField tf, TextArea ta) {
        this.tf = tf; this.ta = ta;
```

```

}
public void actionPerformed(ActionEvent ae) {
ta.append(tf.getText()+"\n");
}
}
}

```

Оброблювач події готовий. При виникненні події типу `ActionEvent` буде створений екземпляр класу-оброблювача `TextMove`, конструктор отримає посилання на конкретні поля об'єкта-джерела, метод `actionPerformed()`, автоматично включившись в роботу, перенесе текст із одного поля в друге.

Тепер напишемо клас-контейнер, в якому знаходяться джерела `tf` і `b` події `ActionEvent`, і підключим до них слухача цієї події `TextMove`, передавши їм посилки на нього методом `addActionListener()`, як показано в лістингі 12.1.

Лістинг 12.1. Обробка події `ActionEvent`

```

import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame{
MyNotebook(String title) {
super(title);
TextField tf = new TextField("Input a text", 50);
add(tf, BorderLayout.NORTH);
TextArea ta = new TextArea();
ta.setEditable(false);
add(ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Transfer");
p.add(b);
tf.addActionListener(new TextMove(tf, ta));
b.addActionListener(new TextMove(tf, ta));
setSize(300, 200); setVisible(true);
}
public static void main(String[] args){
Frame f = new MyNotebook(" Обробка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
class TextMove implements ActionListener{
private TextField tf;
private TextArea ta;
TextMove(TextField tf, TextArea ta){
this.tf = tf; this.ta = ta;
}
public void actionPerformed(ActionEvent ae){
ta.append(tf.getText()+"\n");
}
}

```

На рис. 12.2 показано результат роботи з цією програмою. В лістингі 12.1 в методах `addActionListener()` створюються два екземпляри класу `TextMove` — для прослуховування поля `tf` і для прослуховування кнопки `b`. Можна створити один екземпляр класу `TextMove`, він буде прослуховувати обидва компоненти:

```
TextMove tml = new TextMove(tf, ta);
```

```
tf.addActionListener(tml);
b.addActionListener(tml);
```

Але в першому випадку екземпляри створюються після виникнення події у відповідному компоненті, а в другому — незалежно від того, наступила подія чи ні, що приводить до витрат пам'яті, навіть якщо подія не відбулася. Вирішуйте самі, що краще.

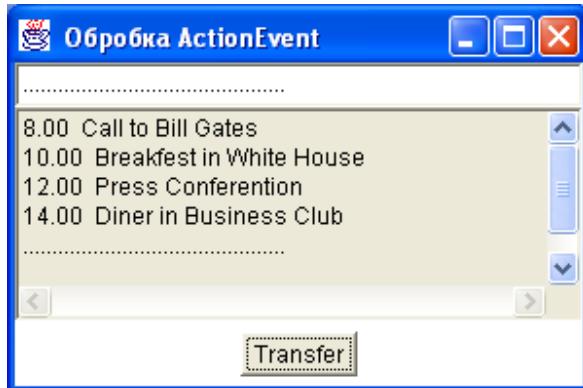


Рис. 12.2. Обробка події ActionEvent

Клас, що містить джерело події, може сам її опрацьовувати. Ви можете самостійно прослухувати компоненти в своїй квартирі, установивши пульт сигналізації біля ліжка. Для цього достатньо реалізувати відповідний інтерфейс прямо в класі-контейнері, як показано в лістингі 12.2.

Лістинг 12.2. Самообробка події ActionEvent

```
import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame implements ActionListener{
private TextField tf;
private TextArea ta;
MyNotebook(String title){
super(title) ;
tf = new TextField ("Input Text **", 50) ;
add(tf, BorderLayout.NORTH) ;
ta = new TextArea();
ta.setEditable(false);
add(ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Transfer");
p.add(b);
tf.addActionListener(this) ;
b.addActionListener(this) ;
setSize(300, 200); setVisible(true) ; }
public void actionPerformed(ActionEvent ae) {
ta.append(tf.getText()+"\n"); }
public static void main(String[] args){
Frame f = new MyNotebook(" Обробка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}});
```

}{
}

Тут `tf` і `ta` уже не локальні змінні, а змінні экземпляру, оскільки вони використовуються і в конструкторі, і в методі `actionPerformed()`. Цей метод тепер - один із методів класу `MyNotebook`. Клас `MyNotebook` став класом-оброблювачем події `ActionEvent` - він реалізує інтерфейс `ActionListener`. В методі `addActionListener()` указується аргумент `this` - клас сам слухає свої компоненти.

Розглянута схема, здається, простіша і зручніша, але вона представляє менше можливостей. Якщо ви захочете змінити обробку, наприклад заносити записи в поле `ta` по алфавіту або по часу виконання завдань, то прийдеться переписати і перекомпілювати клас `MyNotebook`. Ще один варіант - зробити оброблювач вкладеним класом. Це дозволяє обійтися без змінних екземпляра і конструктора в класі-обробчика `TextMove`, як показано в лістингі 12.3.

Лістинг 12.3. Обробка вкладеним класом

```
import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame{
private TextField tf;
private TextArea ta;
MyNotebook(String title){
super(title);
tf = new TextField("Input Text", 50);
add(tf, BorderLayout.NORTH);
ta = new TextArea();
ta.setEditable(false);
add(ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Transfer");
p.add(b);
tf.addActionListener(new TextMove());
b.addActionListener(new TextMove());
setSize(200, 200);
setVisible(true);
}
public static void main(String[] args){
Frame f = new MyNotebook(" Обробка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit (0);
}
});
}
// Вкладений клас
class TextMove implements ActionListener{
public void actionPerformed(ActionEvent ae){
ta.append(tf.getText()+"\n");
}
}
}
```

Нарешті, можна створити безіменний вкладений клас, що ми і робили в цьому і попередньому уроках, обробляючи натискування комбінації клавиш `<Alt>+<F4>` або клік кнопкою миші по кнопці закриття вікна. При цьому виникає подія типу `windowEvent`, для її обробки ми зверталися до методу `windowClosing()`, реалізуючи його звернення до методу завершення додатку `System.exit(0)`. Але для цього треба мати супер клас визначеного безіменного класу, такий як `windowAdapter`. Такими супер класами можуть бути класи-адаптери, про них мова піде чуть далі. Перейдемо до детального розгляду різних типів

подій.

12.2. Подія ActionEvent

Ця проста подія означає, що треба виконати якусь дію. При цьому не має значення, що викликало подію: клік миші, натискання клавіші чи щось інше. В класі `ActionEvent` єсть два корисні методи:

- метод `getActionCommand()` повертає у вигляді рядка `String` напис на кнопці `Button`, точніше, те, що установлено методом `setActionCommand(String s)` класу `Button`, выбраний пункт списку `List`, або щось інше, залежно від компонента;
- метод `getModifiers()` повертає код клавіш `<Alt>`, `<Ctrl>`, `<Meta>` або `<Shift>`, якщо якась одна або декілька з них були натиснені, у вигляд числа типу `int`; узнати, які саме клавіші були натиснуті, можна порівнянням зі статичними константами цього класу `ALT_MASK`, `CTRL_MASK`, `META_MASK`, `SHIFT_MASK`.

Примітка

Клавіші `<Meta>` на PC-клавіатурі неє, її дія часто призначається на клавішу `<Esc>` або ліву клавішу `<Alt>`. Наприклад:

```
public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand() == "Open" &&
        (ae.getModifiers() | ActionEvent.ALT_MASK) != 0) {
        // Якісь дії
    }
}
```

12.3. Обробка дій миши

Подія `MouseEvent` виникає в компоненті по одній із семи причин:

- натискання кнопки миши — ідентифікатор `MOUSE_PRESSED`;
- відпускання кнопки миши — ідентифікатор `MOUSE_RELEASED`;
- клік кнопкою миши — ідентифікатор `MOUSE_CLICKED` (натискання і відпускання не відрізняються);
- переміщення миши — ідентифікатор `MOUSE_MOVED`;
- переміщення миши з натисненою кнопкою — ідентифікатор `MOUSE_DRAGGED`;
- поява курсора миши в компоненті — ідентифікатор `MOUSE_ENTERED`;
- вихід курсора миши із компонента — ідентифікатор `MOUSE_EXITED`.

Для їх обробки єсть сім методів у двох інтерфейсах:

```
public interface MouseListener extends EventListener{
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
}

public interface MouseMotionListener extends EventListener{
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e);
}
```

Ці методи можуть одержати від аргумента `e` координати курсора миши в системі координат компонента методами `e.getx()`, `e.gety()`, або одним методом `e.getPoint()`, повертаючим екземпляр класу `Point`. Подвійний клік кнопкою миши можна відслідкувати методом `e.getClickCount()`, повертаючим кількість кліків. При переміщенні миши повертається 0. Узнати, яка кнопка була натиснута, можна за допомогою методу `e.getModifiers()` класу `inputEvent` порівнянням з наступними статичними константами класу

inputEvent:

- **BUTTON1_MASK** — натиснута перша кнопка, звичайно ліва;
- **BUTTON2_MASK** — натиснута друга кнопка, звичайно середня, або одночасно обидві кнопки на двохкнопковій миші;
- **BUTTON3_MASK** — натиснута третя кнопка, звичайно права.

Приведемо приклад, уже ставший класичним. В лістингі 12.4 представлений найпростіший варіант "рисувалки" — клас **scribble**. При натисканні першої кнопки миші методом **mousePressed()** запам'ятовуються координати курсора миші. При протягуванні миші викреслюються відрізки прямих між поточним і попереднім положенням курсора миші методом **mouseDragged()**. На рис. 12.3 показано приклад роботи з цією програмою.

Лістинг 12.4. Найпростіша программа рисування

```
import java.awt.*;
import java.awt.event.*;
public class ScribbleTest extends Frame{
public ScribbleTest(String s){
super(s);
ScrollPane pane = new ScrollPane();
pane.setSize(300, 300);
add(pane, BorderLayout.CENTER);
Scribble scr = new Scribble(this, 500, 500);
pane.add(scr);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b1 = new Button("Red");
p.add(b1);
b1.addActionListener(scr);
Button b2 = new Button("Green");
p.add(b2);
b2.addActionListener(scr) ;
Button b3 = new Button("Blue");
p.add(b3);
b3.addActionListener(scr) ;
Button b4 = new Button("Black");
p.add(b4);
b4.addActionListener(scr);
Button b5 = new Button("Clean");
p.add(b5);
b5.addActionListener(scr);
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e){
System.exit(0);
}
});
pack();
setVisible(true);
}
public static void main(String[] args){
new ScribbleTest(" \"Рисувалка\"");
}
}
class Scribble extends Component implements ActionListener, MouseListener,
MouseMotionListener{
protected int lastX, lastY, w, h;
protected Color currColor = Color.black;
protected Frame f;
```

```

public Scribble(Frame frame, int width, int height){
f = frame;
w = width;
h = height;
enableEvents(AWTEvent.MOUSE_EVENT_MASK | AWTEvent.MOUSE_MOTION_EVENT_MASK);
addMouseListener(this);
addMouseMotionListener(this); }
public Dimension getPreferredSize(){
return new Dimension(w, h); }
public void actionPerformed(ActionEvent event){
String s = event.getActionCommand();
if (s.equals ("Clean")) repaint();
else if (s.equals ("Red")) currColor = Color.red;
else if (s.equals("Green")) currColor = Color.green;
else if (s.equals("Blue")) currColor = Color.blue;
else if (s.equals("Black")) currColor = Color.black; }
public void mousePressed(MouseEvent e){
if ( (e.getModifiers() & MouseEvent.BUTTON1_MASK) == 0) return;
lastX = e.getX(); lastY = e.getY(); }
public void mouseDragged(MouseEvent e){
if ((e.getModifiers() & MouseEvent.BUTTON1_MASK) == 0) return;
Graphics g = getGraphics();
g.setColor(currColor);
g.drawLine(lastX, lastY, e.getX(), e.getY());
lastX = e.getX(); lastY = e.getY(); }
public void mouseReleased(MouseEvent e){}
public void mouseClicked(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}
public void mouseMoved(MouseEvent e){}
}

```

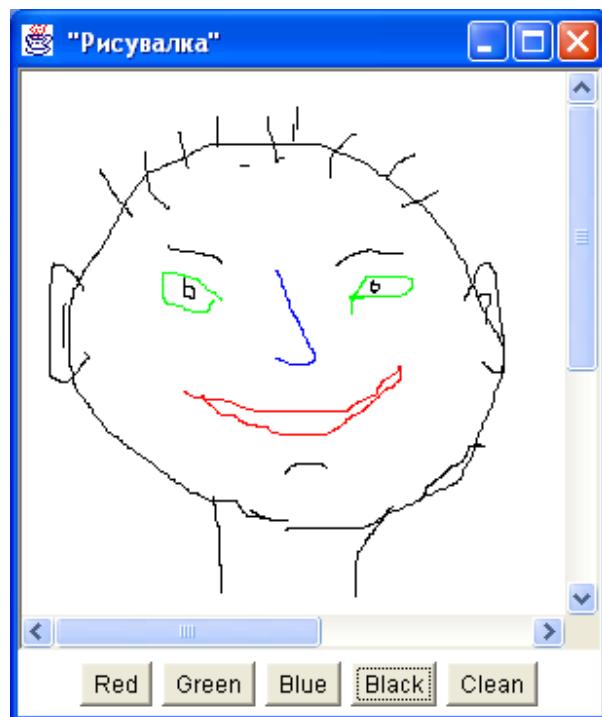


Рис. 12.3. Приклад роботи з програмою рисування

При створенні класу-слухача `scribble` і реалізації інтерфейсів `MouseListener` і `MouseMotionListener` прийшлося реалізувати всі їх сім методів, хоча ми відслідкували тільки натискання і переміщення миші, і нам потрібні були тільки методи `mousePressed()` і `mouseDragged()`. Для решти методів ми задали пусті реалізації. Щоб полегшити задачу реалізації інтерфейсів, що мають більше одного метода, створені класи-адаптери.

12.4. Класи-адаптери

Класи-адаптери представляють собою пусту реалізацію інтерфейсів-слухачів, що мають більше одного методу. Їх імена складаються із імені події і слова `Adapter`. Наприклад, для дії з мишою є два класи-адаптери. Виглядають вони дуже просто:

```
public abstract class MouseAdapter implements MouseListener{
    public void mouseClicked(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}

public abstract class MouseMotionAdapter implements MouseMotionListener{
    public void mouseDragged(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}
}
```

Замість того щоб реалізувати інтерфейс, можна розширяти ці класи. Не бог знає що, але корисно для створенні безіменного вкладеного класу, як у нас і робилося для закриття вікна. Там ми використовували клас-адаптер `WindowAdapter`. Класів-адаптерів всього сім. Крім уже згадуваних трьох класів, це класи `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter` і `KeyAdapter`.

12.5. Обробка дій клавіатури

Подія `KeyEvent` відбувається в компоненті по любій із трьох причин:

- натиснута клавіша — ідентифікатор `KEY_PRESSED`;
- відпущенна клавіша — ідентифікатор `KEY_RELEASED`;
- введений символ — ідентифікатор `KEY_TYPED`.

Остання подія виникає із-за того, що деякі символи вводяться натисканням декількох клавіш, наприклад, заглавні літери вводяться комбінацією клавіш `<Shift>+<літера>`. Згадайте ще `<Alt>`-введення в `MS Windows`. Натискання функціональних клавіш, наприклад `<F1>`, не викликає подію `KEY_TYPED`. Обробляються ці події трьома методами, описаними в інтерфейсі:

```
public interface KeyListener extends EventListener{
    public void keyTyped(KeyEvent e);
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
}
```

- Метод `e.getKeyChar()` повертає символ `Unicode` типу `char`, звязаний з клавішею. Якщо з клавішею не звязаний ніякий символ, то повертається константа `CHAR_UNDEFINED`.
- Метод `e.getKeyCode()` повертає код клавіши у вигляді цілого числа типу `int`.

В класі `KeyEvent` визначені коди всіх клавіш у вигляді констант, названих *віртуальними кодами* клавіш (*virtual key codes*), наприклад, `VK_F1`, `VK_SHIFT`, `VK_A`, `VK_B`, `VK_PLUS`. Вони перечислені в документації до класу `KeyEvent`. Фактичне значення віртуального коду залежить від мови і розкладки клавіатури. Щоб узнати, яка клавіша була натиснута, треба порівняти результат виконання методу `getKeyCode()` з цими константами. Якщо коду клавіші немає, як відбувається при настанні події `KEY_TYPED`, то повертається значення `VK_UNDEFINED`. Щоб узнати, чи не натиснута одна або декілька клавіш-модифікаторів `<Alt>`,

<Ctrl>, <Meta>, <Shift>, треба скористатися унаслідуванням від класу `inputEvent` методом `getModifiers()` і порівняти його результат з константами `ALT_MASK`, `CTRL_MASK`, `META_MASK`, `SHIFT_MASK`. Другий спосіб — застосувати логічні методи `isAltDown()`, `isControlDown()`, `isMetaDown()`, `isShiftDown()`. Додамо в лістинг 12.3 можливість очистки поля введення `tf` після натискання клавіші <Esc>. Для цього перепишемо вкладений клас-слухач `TextMove`:

```
class TextMove implements ActionListener, KeyListener{
public void actionPerformed(ActionEvent ae) {
ta.append(tf.getText() + "\n");
}
public void keyPressed(KeyEvent ke) {
if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) tf.setText("");
}
public void keyReleased(KeyEvent ke) {}
public void keyTyped(KeyEvent ke) {}
}
```

12.6. Подія TextEvent

Подія `TextEvent` настає тільки по одній причині — зміні тексту — і позначається ідентифікатором `TEXT_VALUE_CHANGED`. Відповідний інтерфейс має тільки один метод:

```
public interface TextListener extends EventListener{
public void textValueChanged(TextEvent e);
}
```

Від аргументу `e` цього методу можна одержати посилку на об'єкт-джерело події методом `getSource()`, унаслідуванням від класу `EventObject`, наприклад, так:

```
TextComponent tc = (TextComponent)e.getSource();
String s = tc.getText();
// Подальша обробка
```

12.7. Обробка дій з вікном

Подія `windowEvent` може настати по семи причинах:

- вікно відкрилось — ідентифікатор `WINDOW_OPENED`;
- вікно закрилось — ідентифікатор `WINDOW_CLOSED`;
- спроба закриття вікна — ідентифікатор `WINDOW_CLOSING`;
- вікно отримало фокус — ідентифікатор `WINDOW_ACTIVATED`;
- вікно втратило фокус — ідентифікатор `WINDOW_DEACTIVATED`;
- вікно звернулося в ярлик — ідентифікатор `WINDOW_ICONIFIED`;
- вікно розвернулося — ідентифікатор `WINDOW_DEICONIFIED`.

Відповідний інтерфейс містить сім методів:

```
public interface WindowListener extends EventListener {
public void windowOpened(WindowEvent e);
public void windowClosing(WindowEvent e);
public void windowClosed(WindowEvent e);
public void windowIconified(WindowEvent e);
public void windowDeiconified(WindowEvent e);
public void windowActivated(WindowEvent e);
public void windowDeactivated(WindowEvent e); }
```

Аргумент `e` цих методів дає посилку типу `Window` на вікно-джерело методом `e.getWindow()`. Частіше всього ці події використовуються для перерисування вікна методом `repaint()` при зміні його розмірів і для зупинки додатку при закртті вікна.

12.8. Подія ComponentEvent

Дана подія настає в компоненті по чотирьох причинах:

- компонент переміщається — ідентифікатор **COMPONENT_MOVED**;
- компонент змінює розмір — ідентифікатор **COMPONENT_RESIZED**;
- компонент видалений з екрана — ідентифікатор **COMPONENT_HIDDEN**;
- компонент появився на екране — ідентифікатор **COMPONENT_SHOWN**.
-

Відповідний інтерфейс містить опис чотирьох методів:

```
public interface ComponentListener extends EventListener{
    public void componentResized(ComponentEvent e);
    public void componentMoved(ComponentEvent e);
    public void componentShown(ComponentEvent e);
    public void componentHidden(ComponentEvent e);
}
```

Аргумент **e** методів цього інтерфейса представляє посилання на компонент-джерело події методом **e.getComponent()**.

12.9. Подія ContainerEvent

Ця подія настає по двох причинах:

- в контейнер додано компонент — ідентифікатор **COMPONENT_ADDED**;
- із контейнера видалено компонент — ідентифікатор **COMPONENT_REMOVED**.

Цим причинам відповідають методи інтерфейса:

```
public interface ContainerListener extends EventListener{
    public void componentAdded(ContainerEvent e);
    public void componentRemoved(ContainerEvent e);
}
```

Аргумент **e** представляє посилку на компонент, чиє додавання або видалення із контейнера викликало подію, методом **e.getChild()**, і посилку на контейнер - джерело події методом **e.getContainer()**. Звичайно при настанні даної події контейнер переміщає свої компоненти.

12.10. Подія FocusEvent

Подія настає в компоненті, коли він отримує фокус введення — ідентифікатор **FOCUS_GAINED**, або втрачає фокус - ідентифікатор **FOCUS_LOST**. Відповідний інтерфейс:

```
public interface FocusListener extends EventListener{
    public void focusGained(FocusEvent e);
    public void focusLost(FocusEvent e);
}
```

Звичайно при втраті фокусу компонент перерисовується блідим кольором, для цього застосовується метод **brighter()** класу **Color**, при отриманні фокусу становиться яскравіше, що досягається застосуванням методу **darker()**. Це приходиться робити самостійно при створенні свого компонента.

12.11. Подія ItemEvent

Ця подія настає при виборі або відмові від вибору елемента в списку **List**, **Choice** або прапорця **Checkbox** і позначається ідентифікатором **ITEM_STATE_CHANGED**. Відповідний інтерфейс дуже простий:

```
public interface ItemListener extends EventListener{
void itemStateChanged(ItemEvent e);
}
```

Аргумент `e` представляє посилку на джерело методом `e.getItemSelectable()`, посилку на вибраний пункт методом `e.getItem()` у вигляді `Object`. Метод `e.getStateChange()` дозволяє уточнити, що відбулося: значення `SELECTED` указує на те, що елемент був вибраний, значення `DESELECTED` — відбулася відмова від вибору. В наступному уроці ми розглянемо приклади використання цієї події.

12.12. Подія AdjustmentEvent

Ця подія настає для смуги прокрутки `Scrollbar` при всякій зміні її бігунка і позначається ідентифікатором `ADJUSTMENT_VALUE_CHANGED`. Відповідний інтерфейс описує один метод:

```
public interface AdjustmentListener extends EventListener{
public void adjustmentValueChanged(AdjustmentEvent e);
}
```

Аргумент `e` цього метода представляє посилку на джерело події методом `e.getAdjustable()`, поточне значення положення бігунка смуги прокрутки методом `e.getvalue()`, і спосіб зміни його значення методом `e.getAdjustmentType()`, повертаючим наступні значення:

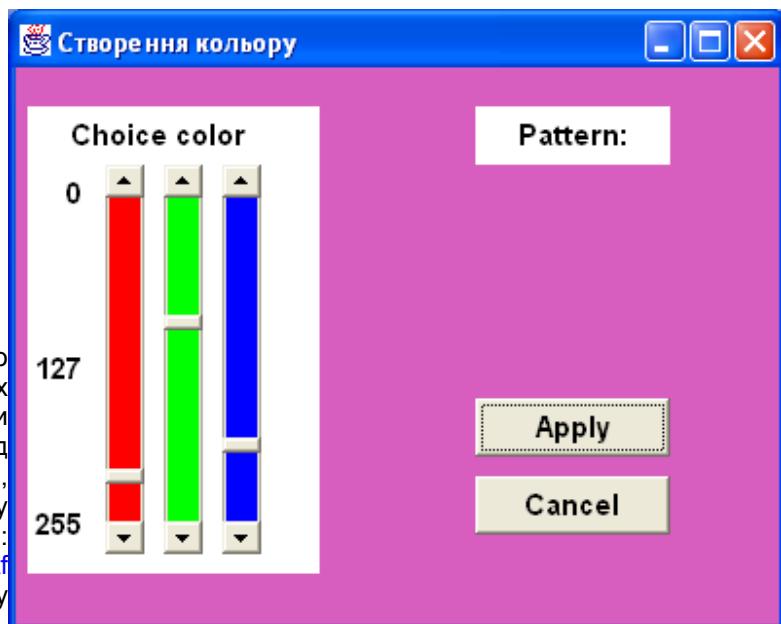
- `UNIT_INCREMENT` — збільшення на одну одиницю;
- `UNIT_DECREMENT` — зменшення на одну одиницю;
- `BLOCK_INCREMENT` — збільшення на один блок;
- `BLOCK_DECREMENT` — зменшення на один блок;
- `TRACK` — процес переміщення бігунка смуги прокрутки.

"Оживимо" програму створення кольору, приведену в лістингі 10.4, додавши необхідні дії. Результат цього приведений в лістингі 12.5.

Лістинг 12.5. Програма створення кольору

```
import java.awt.*;
import java.awt.event.*;
class ScrollTest extends Frame{
Scrollbar sbRed = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
Scrollbar sbGreen = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
Scrollbar sbBlue = new Scrollbar(Scrollbar.VERTICAL, 127, 10, 0, 255);
Color c = new Color(127, 127, 127);
Label lm = new Label();
Button b1 = new Button("Apply");
Button b2 = new Button("Cancel");
ScrollTest(String s){ super(s);
setLayout(null);
setFont(new Font("Serif", Font.BOLD, 15));
Panel p = new Panel();
p.setLayout(null);
p.setBounds(10,50, 150, 240); add(p);
Label lc = new Label("Choice color");
lc.setBounds(20, 0, 120, 30); p.add(lc);
Label lmin = new Label("0", Label.RIGHT);
lmin.setBounds(0, 30, 30, 30); p.add(lmin);
Label lmiddle = new Label("127", Label.RIGHT);
lmiddle.setBounds(0, 120, 30, 30); p.add(lmiddle);
Label lmax = new Label("255", Label.RIGHT);
lmax.setBounds(0, 200, 30, 30); p.add(lmax);
sbRed.setBackground(Color.red);
sbRed.setBounds(40, 30, 20, 200); p.add(sbRed);
}
```

```
sbRed.addAdjustmentListener(new ChColor());
sbGreen.setBackground(Color.green);
sbGreen.setBounds(70, 30, 20, 200); p.add(sbGreen);
sbGreen.addAdjustmentListener(new ChColor());
sbBlue.setBackground(Color.blue);
sbBlue.setBounds(100, 30, 20, 200); p.add(sbBlue);
sbBlue.addAdjustmentListener(new ChColor());
Label lp = new Label("Pattern:", Label.CENTER);
lp.setBounds(240, 50, 100, 30); add(lp);
lm.setBackground(new Color(127, 127, 127));
lm.setBounds(220, 80, 120, 80); add(lm);
b1.setBounds(240, 200, 100, 30); add(b1);
b1.addActionListener(new ApplyColor());
b2.setBounds(240, 240, 100, 30); add(b2);
b2.addActionListener(new CancelColor());
setSize(400, 320); setVisible(true);
}
class ChColor implements AdjustmentListener{
public void adjustmentValueChanged(AdjustmentEvent e){
int red = c.getRed(), green = c.getGreen(), blue = c.getBlue();
if (e.getAdjustable() == sbRed) red = e.getValue();
else if (e.getAdjustable() == sbGreen) green = e.getValue();
else if (e.getAdjustable() == sbBlue) blue = e.getValue();
c = new Color(red, green, blue);
lm.setBackground(c);
}
}
class ApplyColor implements ActionListener {
public void actionPerformed(ActionEvent ae){
setBackground(c);
}
}
class CancelColor implements ActionListener {
public void actionPerformed(ActionEvent ae){
c = new Color(127, 127, 127);
sbRed.setValue(127);
sbGreen.setValue(127);
sbBlue.setValue(127);
lm.setBackground(c);
setBackground(Color.white);
}
}
public static void main(String[] args){
Frame f = new ScrollTest("Створення кольору ");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```



12.13. Декілька слухачів одного джерела

На початку цього уроку, в лістингах 12.1—12.3, ми привели приклад класу `TextMove`, слухаючого зразу два компоненти: поле введення `tf` типу `TextField` у кнопку `b` типу `Button`.

Частіше зустрічається протилежна ситуація — декілька слухачів слідкують за одним компонентом. В тому ж прикладі кнопка `b` у відповідь на клік по ній кнопки миші здійснювала ще і власні дії — вона "вдавлювалась", а при відпусканні кнопки миші становилась "випуклою". В класі `Button` ці дії виконує реєр-об'єкт. В класі `FlowerButton` лістинга 10.6 такі ж дії виконує метод `paint()` цього класу. В даній моделі реалізований *design pattern* під назвою `observer`. До кожного компонента можна приєднати скільки завгодно слухачів однієї тієї ж події або різних типів подій. Однаке при цьому не гарантується якийсь певний порядок їх виклику, хоча частіше всього слухачі викликаються в порядку написання методів `addXxxListener()`. Якщо потрібно задати визначений порядок виклику слухачів для обробки подій, то прийдеться звертатися до них одинза одним або створювати об'єкт, викликаючий слухачів у потрібному порядку.

Посилки на приєднані методами `addxxxListener()` слухачі можна було б зберігати в будь-якому класі-коллекції, наприклад, `Vector`, але в пакет `java.awt` спеціально для цього введений клас `AWTEventMulticaster`. Він реалізує всі одинадцять інтерфейсів `xxxListener`, значить, сам являється слухачем будь-якої події. Основу класу складають своєрідні статичні методи `add()`, написані для кожного типу подій, наприклад:

`add(ActionListener a, ActionListener b)`

Своєрідність цих методів двояка: вони повертають посилку на той же інтерфейс, в даному випадку, `ActionListener`, і приєднують об'єкт `a` до об'єкту `b`, створюючи сукупність слухачів одного і того ж типу. Це дозволяє застосовувати їх подібно операціям `a +=`. Заглянувши у вихідний текст класу `Button`, ви

побачите, що метод `addActionListener()` дуже простий:

```
public synchronized void addActionListener(ActionListener l) {
    if (l == null) { return; }
    actionListener = AWTEventMulticaster.add(actionListener, l);
    newEventsOnly = true;
}
```

Він додає до сукупності слухачів `actionListener` нового слухача 1. Для подій типу `inputEvent`, а саме, `KeyEvent` і `MouseEvent`, є можливість зупинити подальшу обробку події методом `consume()`. Якщо записати виклик цього метода в клас-слухач, то ні `peer`-обєкти, ні наступні слухачі не будуть обробляти подію. Цим способом звичайно коістуються, щоб відмінить стандартні дії компонента, наприклад, "вдавлювання" кнопки.

12.14. Диспетчеризація подій

Якщо вам знадобиться обробити просто дію миші, не важливо, натискання це, переміщення або ще що-небудь, то прийдеться включати цю обробку у всі сім методів двох класів-слухачів подій миші. Цю роботу можна полегшити, виконавши обробку не в слухачі, а на більш ранній стадії. Справа в тому, що перш ніж подія діде до слухача, вона обробляється кількома методами. Щоб в компоненті настала подія `AWT`, повинна бути виконана хоча б одна із двох умов: до компонента приєднаний слухач або в конструкторі компонента визначена мрежливість появи події методом `enableEvents()`. В аргументі цього метода через операцію побітового додавання перечисляються константи класу `AWTEvent`, задаючі події, які можуть настати в компоненті, наприклад:

```
enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK |
AWTEvent.MOUSE_EVENT_MASK | AWTEvent.KEY_EVENT_MASK)
```

При настанні події створюється об'єкт ідповідного класу `xxxEvent`. Метод `dispatchEvent()` визначає, де з'явилася подія — в компоненті чи одному із його підкомпонентів, — і передає об'єкт-подію методу `processEvent()` компонента-джерела. Метод `processEvent()` визначає тип події і передає його спеціалізованому методу `processxxxEvent()`. Ось початок цього методу:

```
protected void processEvent(AWTEvent e) {
    if (e instanceof FocusEvent) {
        processFocusEvent((FocusEvent)e);
    } else if (e instanceof MouseEvent) {
        switch (e.getlDO) {
            case MouseEvent.MOUSE_PRESSED:
            case MouseEvent.MOUSE_RELEASED:
            case MouseEvent.MOUSE_CLICKED:
            case MouseEvent.MOUSE_ENTERED:
            case MouseEvent.MOUSE_EXITED:
                processMouseEvent((MouseEvent)e);
                break;
            case MouseEvent.MOUSE_MOVED:
            case MouseEvent.MOUSE_DRAGGED:
                processMouseEvent((MouseEvent)e);
                break;
        } } else if (e instanceof KeyEvent) {
        processKeyEvent((KeyEvent)e);
    }
    // ...
}
```

Потім включається спеціалізований метод, наприклад, `processKeyEvent()`. Він-то і передає об'єкт-подію слухачу. Ось вихідний текст цього методу:

```
protected void processKeyEvent(KeyEvent e) {
    KeyListener listener = keyListener;
    if (listener != null) { int id = e.getlDf);
        switch(id) {
            case KeyEvent.KEYJTTYPED: listener.keyTyped(e);
            case KeyEvent.KEYJTTRELEASED: listener.keyReleased(e);
            case KeyEvent.KEYJTTDOWN: listener.keyPressed(e);
        }
    }
}
```

```

break;
case KeyEvent.KEY_PRESSED: listener.keyPressed(e);
break;
case KeyEvent.KEY_RELEASED: listener.keyReleased(e);
break;
}
}
}
}

```

Із цього опису видно, що коли ви хочете обробити будь-яку подію типу `AWTEvent`, то вам потрібно перевизначити метод `processEvent()`, а якщо більш конкретну подію, наприклад, подію клавіатури, - перевизначити більш конкретний метод `processKeyEvent()`. Коли ви не перевизначаєте весь метод цілком, то не забудьте в кінці звернутися до методу суперкласу, наприклад, `super.processKeyEvent(e)`;

Зауваження

Не забувайте звертатися до методу `processXxxEvent()` суперкласу.

В наступному уроці ми застосуємо таке перевизначення в лістингі 13.2 для виклику спливаючого меню.

12.15. Створення власної події

Ви можете створити власну подію і визначити джерело та умови її настання. В лістингі 12.6 приведений приклад створення події `MyEvent`. Подія `MyEvent` говорить про початок роботи програми (`START`) і закінчення її роботи (`STOP`).

Лістинг 12.6. Створення власної події

```

// 1. Створюємо свій клас події:
public class MyEvent extends java.util.EventObject{ protected int id;
public static final int START = 0, STOP = 1;
public MyEvent(Object source, int id){
super(source);
this.id = id;
}
public int getID(){ return id; }
}
// 2. Описуємо Listener:
public interface MyListener extends java.util.EventListener{
public void start(MyEvent e);
public void stop(MyEvent e); }
// 3. В тілі потрібного класу створюємо метод fireEvent():
protected Vector listeners = new Vector();
public void fireEvent( MyEvent e){
Vector list = (Vector) listeners.clone();
for (int i = 0; i < list.size(); i++) {
MyListener listener = (MyListener) list.elementAt(i);
switch(e.getID() ) {
case MyEvent.START: listener.start(e); break;
case MyEvent.STOP: listener.stop(e); break;
}
}
}

```

Все, тепер при запуску програми робимо

```
fireEvent(this, MyEvent.START);
```

а по закінченню

```
fireEvent(this, MyEvent.STOP);
```

При цьому всі зареєстровані слухачі одержать екземпляри подій.

Лабораторна робота 11. Програмування реакції на натискання клавіш і маніпуляції з мишкою.

1. Побудова кривих другого порядку по означенню.

Події типу [WindowEvent](#) виникають тільки у вікнах: [Frame](#), [Dialog](#), [FileDialog](#), [Window](#).

Події типу [TextEvent](#) генеруються тільки в контейнерах [Textcomponent](#), [TextArea](#), [TextField](#).

Події типу [ActionEvent](#) проявляються тільки в контейнерах [Button](#), [List](#), [TextField](#).

Події типу [ItemEvent](#) виникають тільки в контейнерах [Checkbox](#), [Choice](#), [List](#).

Нарешті, події типу [AdjustmentEvent](#) виникають тільки в контейнерах [Scrollbar](#).

Лістинг 12.1. Обробка події ActionEvent

```
import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame{
MyNotebook(String title) {
super(title);
TextField tf = new TextField("Input a text", 50);
add(tf, BorderLayout.NORTH);
TextArea ta = new TextArea();
ta.setEditable(false);
add(ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Transfer");
p.add(b);
tf.addActionListener(new TextMove(tf, ta));
b.addActionListener(new TextMove(tf, ta));
setSize(300, 200); setVisible(true);
}
public static void main(String[] args){
Frame f = new MyNotebook(" Обробка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
class TextMove implements ActionListener{
private TextField tf;
private TextArea ta;
TextMove(TextField tf, TextArea ta){
this.tf = tf; this.ta = ta;
}
public void actionPerformed(ActionEvent ae){
ta.append(tf.getText()+"\n");
}
}
```

. В лістингі 12.1 в методах [addActionListener\(\)](#) створюються два екземпляри класу [TextMove](#) — для

прослуховування поля **tf** і для прослуховування кнопки **b**. Можна створити один екземпляр класу **TextMove**, він буде прослуховувати обидва компоненти:

```
TextMove tml = new TextMove(tf, ta);
tf.addActionListener(tml);
b.addActionListener(tml);
```

Лістинг 12.2. Самообробка події ActionEvent

```
import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame implements ActionListener{
private TextField tf;
private TextArea ta;
MyNotebook(String title){
super(title) ;
tf = new TextField ("Input Text **", 50) ;
add(tf, BorderLayout.NORTH);
ta = new TextArea();
ta.setEditable(false);
add(ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Transfer");
p.add(b);
tf.addActionListener(this) ;
b.addActionListener(this) ;
setSize(300, 200); setVisible(true) ;
}
public void actionPerformed(ActionEvent ae){
ta.append(tf.getText()+"\n");
}
public static void main(String[] args){
Frame f = new MyNotebook(" Обробка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
```

Лістинг 12.3. Обробка вкладеним класом

```
import java.awt.*;
import java.awt.event.*;
class MyNotebook extends Frame{
private TextField tf;
private TextArea ta;
MyNotebook(String title){
super(title);
tf = new TextField("Input Text", 50);
add(tf, BorderLayout.NORTH);
ta = new TextArea();
ta.setEditable(false);
add (ta);
Panel p = new Panel();
add(p, BorderLayout.SOUTH);
Button b = new Button("Transfer");
p.add(b);
```

```
tf.addActionListener(new TextMove());
b.addActionListener(new TextMove());
setSize(200, 200);
setVisible(true);
}
public static void main(String[] args){
Frame f = new MyNotebook(" Обробка ActionEvent");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit (0);
}
});
}
// Вкладений клас
class TextMove implements ActionListener{
public void actionPerformed(ActionEvent ae) {
ta.append(tf.getText()+"\n");
}
}
}
```

Програмування у Java

Урок 13. Створення меню

13.1. Меню

В контейнер типу [Frame](#) закладена можливість установки стандартного рядка меню ([menu bar](#)), розташованого нижче рядка заголовка, як показано на рис. 13.1. Цей рядок - об'єкт класу [MenuBar](#). Все, що потрібно зробити для установки рядка меню в контейнері [Frame](#) - це створити об'єкт класу [MenuBar](#) і звернутися до методу [setMenuBar\(\)](#):

```
Frame f = new Frame("Приклад меню");
MenuBar mb = newMenuBar();
f.setMenuBar(mb);
```

Якщо ім'я [mb](#) не знадобиться, можна сумістити два останніх звернення до методів:

```
f.setMenuBar(newMenuBar());
```

Розуміється, рядок меню ще пустий й пункти меню не створені. Кожний елемент рядка меню - *випадаюче меню* ([drop-down menu](#)) - це об'єкт класу [Menu](#). Створити ці об'єкти і занести їх в рядок меню не складніше, ніж створити рядок меню:

```
Menu mFile = new Menu("Файл");
mb.add(mFile);
Menu mEdit = new Menu("Правка");
mb.add(mEdit);
Menu mView = new Menu("Вид");
mb.add(mView);
Menu mHelp = new Menu("Справка");
mb.setHelpMenu(mHelp);
```

і т. д. Елементи розташовуються зліва направо в порядку звернень до методів [add\(\)](#), як показано на рис. 13.1. В багатьох графічних системах прийнято меню Справка ([Help](#)) притискати до правого краю рядка меню. Це досягається зверненням до методу [setHelpMenu\(\)](#), але фактичне положення меню Справка визначається графічною оболонкою.

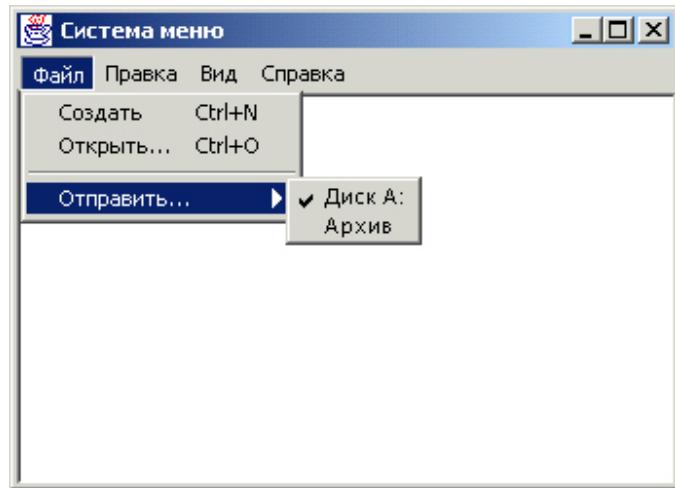


Рис. 13.1. Система меню

Потім визначаємо кожне випадаюче меню, створюючи його пункти. Кожний пункт меню — це об'єкт класу [MenuItem](#). Схема його створення і додавання до меню точно така ж, як і самого меню:

```
MenuItem create = new MenuItem("Створити");
mFile.add(create);
MenuItem open = new MenuItem("Відкрити...");
mFile.add(open);
```

і т. д.

Пункти меню будуть озташовані зверху вниз в порядку звернення до методів `add()`. Часто пункти меню об'єднуються в групи. Одна група від іншої відокремлюється горизонтальною рискою. На рис. 13.1 риска проведена між командами Відкрити і Відправити. Ця риска створюється методом `addSeparator()` класу `Menu` або визначається як пункт меню з написом спеціального виду — дефісом:

```
mFile.add(new MenuItem("-"));
```

Цікаво, що клас `Menu` розширює клас `MenuItem`, а не навпаки. Це означає, що меню само являється пунктом меню, і дозволяє задавати меню в якості пункта іншого меню, тим самим організуючи вкладені підменю:

```
Menu send = new Menu("Відправити");
mFile.add(send);
```

Тут меню `send` додається в меню `mFile` як один із його пунктів. Підменю `send` заповняється пунктами меню як звичайне меню. Часто команди меню створюються для вибору із них певних можливостей, подібно компонентам `Checkbox`. Такі пункти можна виділити кліком кнопки миші або відмінити виділення повторним кліком. Ці команди — об'єкти класу `CheckboxMenuItem`:

```
CheckboxMenuItem disk = new CheckboxMenuItem("Диск А:", true);
send.add(disk);
send.add(new CheckboxMenuItem("Архів"));
```

і т. д.

Все, що получилось в результаті перечислених дій, показано на рис. 13.1. Деякі графічні оболонки, але не MS Windows, дозволяють створювати відокремлювані (`tear-off`) меню, які можна переміщати по екрану. Це указується в конструкторі

`Menu(String label, boolean tearOff)`

Якщо `tearoff == true` і графічна оболочка уміє створювати відокремлюване меню, то воно буде створено. В протилежному випадку цей аргумент просто ігнорується.

Нарешті, треба призначити дії командам меню. Команди меню типу `MenuItem` породжують події типу `ActionEvent`, тому потрібно приєднати до них об'єкт класу-слухача як до звичайних компонентів, записавши щось подібне до

```
create.addActionListener(new SomeActionEventHandler())
open.addActionListener(new AnotherActionEventHandler())
```

Пункти типу `CheckboxMenuItem` породжують події типу `ItemEvent`, тому треба звертатися до об'єкту-слухача цієї події:

```
disk.addItemListener(new SomeItemEventHandler())
```

Дуже часто дії, записані в командах меню, викликаються не тільки кліком кнопки миші, але і "гарячими" клавішами-акселераторами (`shortcut`), діючими частіше всього при натисканні клавіші `<Ctrl>`. На екрані в пунктах меню, яким призначені "гарячі" клавіші, появляються підказки виду `Ctrl+N`, `Ctrl+O`, як на рис. 13.1. "Гаряча" клавіша визначається об'єктом класу `MenuShortcut` і вказується в його конструкторі константою класу `KeyEvent`, наприклад:

```
MenuShortcut keyCreate = new MenuShortcut(KeyEvent.VK_N);
```

Після цього "гарячою" буде комбінація клавіш **<Ctrl>+<N>**. Потім отриманий об'єкт указується в конструкторі класу **MenuItem**:

```
MenuItem create = new MenuItem("Створити", keyCreate);
```

Натискання **<Ctrl>+<N>** буде викликати вікно створення. Ці дії, зрозуіло, можна сумістити, наприклад,

```
MenuItem open = new MenuItem("Відкрити...",  
new MenuShortcut(KeyEvent.VK_O));
```

Можна додати ще натискання клавіші **<Shift>**. Дія пункта меню буде викликатися натисканням комбінації клавіш **<Shift>+<Ctrl>+<X>**, якщо скористуватися другим конструктором:

```
MenuShortcut(int key, boolean useShift) з аргументом useShift == true.
```

Програма рисування, створена в лістингі 12.4 і показана на рис. 12.3, явно перевантажена кнопками. Перенесемо їх дії в пункти меню. Додамо можливість маніпуляції файлами і команду завершення роботи. Це зроблено в лістингі 13.1. Клас **scribble** не змінився і в лістингі не наведений. Результат показаний на рис. 13.2.

Лістинг 13.1. Програма рисування з меню

```
import java.awt.*;  
import java.awt.event.*;  
public class MenuScribble extends Frame{  
  
    public MenuScribble(String s) { super(s);  
        setSize(400,150);  
        setVisible(true);  
  
        ScrollPane pane = new ScrollPane();  
        pane.setSize(300, 300);  
        add(pane, BorderLayout.CENTER);  
  
        Scribble scr = new Scribble(this, 500, 500);  
        pane.add(scr);  
  
        MenuBar mb = new MenuBar();  
        setMenuBar(mb);  
        Menu f = new Menu("File");  
        Menu v = new Menu("Draw");  
        mb.add(f); mb.add(v);  
  
        MenuItem open = new MenuItem("Open...", new MenuShortcut(KeyEvent.VK_O));  
        MenuItem save = new MenuItem("Save", new MenuShortcut(KeyEvent.VK_S));  
        MenuItem saveAs = new MenuItem("Save As...");  
        MenuItem exit = new MenuItem("Exit", new MenuShortcut(KeyEvent.VK_Q));  
        f.add(open); f.add(save); f.add(saveAs);  
        f.addSeparator(); f.add(exit);  
  
        open.addActionListener(new ActionListener(){  
            public void actionPerformed(ActionEvent e){  
                FileDialog fd = new FileDialog(new Frame(),  
                " Open File", FileDialog.LOAD);  
                fd.setVisible(true);  
            }  
        });
```

```

saveAs.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
FileDialog fd = new FileDialog(new Frame(),
" Save File As", FileDialog.SAVE);
fd.setVisible(true);
}
});

exit.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
System.exit(0);
}
});

addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit (0);
}
});
});

Menu c = new Menu("Color");
v.add(c);
MenuItem clear = new MenuItem("Clear",new MenuShortcut(KeyEvent.VK_D));
v.add(clear);

MenuItem red = new MenuItem("Red");
MenuItem green = new MenuItem("Green");
MenuItem blue = new MenuItem("Blue");
MenuItem black = new MenuItem("Black");
c.add(red); c.add(green); c.add(blue); c.add(black);

red.addActionListener(scr);
green.addActionListener(scr);
blue.addActionListener(scr) ;
black.addActionListener(scr) ;
clear.addActionListener(scr) ;
}

}

public static void main(String[] args){
Frame fr = new MenuScribble (" \"Рисовалка\" в меню");

}
}

class Scribble extends Component implements ActionListener, MouseListener,
MouseMotionListener{
protected int lastX, lastY, w, h;
protected Color currColor = Color.black;
protected Frame f;
public Scribble(Frame frame, int width, int height){
f = frame;
w = width;
h = height;
enableEvents(AWTEvent.MOUSE_EVENT_MASK | AWTEvent.MOUSE_MOTION_EVENT_MASK);
addMouseListener(this);
addMouseMotionListener(this); }
public Dimension getPreferredSize(){
return new Dimension(w, h); }
}

```

```

public void actionPerformed(ActionEvent event){
String s = event.getActionCommand();
if (s.equals ("Clear")) repaint();
else if (s.equals ("Red")) currColor = Color.red;
else if (s.equals("Green")) currColor = Color.green;
else if (s.equals("Blue")) currColor = Color.blue;
else if (s.equals("Black")) currColor = Color.black; }
public void mousePressed(MouseEvent e){
if ( (e.getModifiers() & MouseEvent.BUTTON1_MASK) == 0) return;
lastX = e.getX(); lastY = e.getY(); }
public void mouseDragged(MouseEvent e){
if ((e.getModifiers() & MouseEvent.BUTTON1_MASK) == 0) return;
Graphics g = getGraphics();
g.setColor(currColor);
g.drawLine(lastX, lastY, e.getX(), e.getY());
lastX = e.getX(); lastY = e.getY(); }
public void mouseReleased(MouseEvent e){}
public void mouseClicked(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}
public void mouseMoved(MouseEvent e){}
}

```

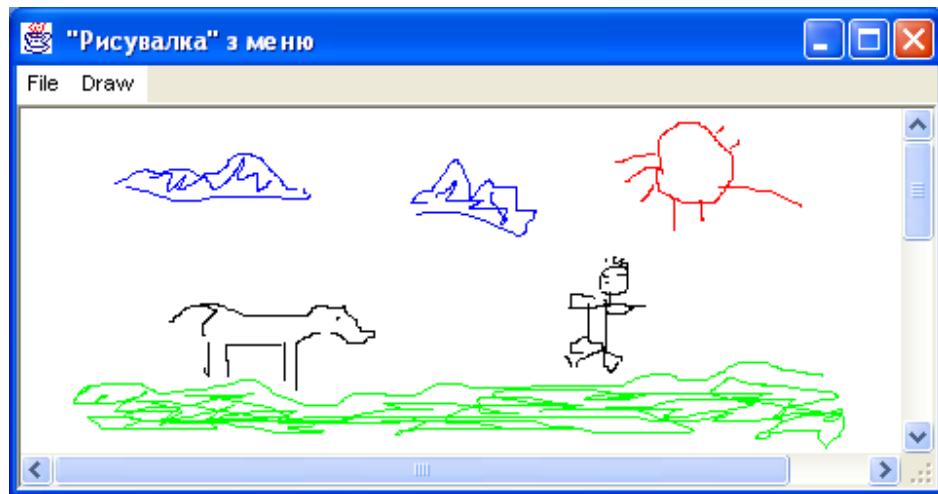


Рис. 13.2. Програма рисування з меню

13.2. Спливаюче меню

Спливаюче меню ([popup menu](#)) появляється звичайно при натисканні або відпусканні правої або середньої кнопки миші і являється контекстним ([context](#)) меню. Його команди залежать від компонента, на якому була натиснута кнопка миші. В мові Java спливаюче меню — об'єкт класу [PopupMenu](#). Цей клас розширяє клас [Menu](#), а значить, наслідує всі властивості меню і пункту меню [MenuItem](#). Спливаюче меню приєднується не до рядка меню типу [MenuBar](#) або до меню типу [Menu](#) в якості підменю, а до певного компонента. Для цього в класі [Component](#) є метод [add\(PopupMenu menu\)](#).

У деяких компонентів, наприклад [TextField](#) і [TextArea](#), уже існує спливаюче меню. Подібні меню не можна перевизначати.

Приєднати спливаюче меню можна тільки до одного компонента. Якщо треба використовувати спливаюче меню з кількома компонентами в контейнері, то його приєднують до контейнера, а потрібний компонент визначають за допомогою метода [getComponent\(\)](#) класу [MouseEvent](#), як показано в лістингі 13.2.

Крім унаслідуваних властивостей і методів, в класі `PopupMenu` єсть метод `show (Component comp, int x, int y)`, показуючий спливаюче меню на екрані так, що його лівий верхній кут роташовується в точці `(x, y)` в системі координат компонента `comp`. Частіше всього це компонент, на якому натиснута кнопка миші, і повертається методом `getComponent()`. Компонент `comp` повинен бути всередині контейнера, до якого приєднують меню, інакше виникне виключна ситуація.

Спливаюче меню появляється у [MS Windows](#) при відпусканні правої кнопки миші, а в інших графічних системах можуть бути іші правила. Щоб врахувати цю різницю, в клас `MouseEvent` ведено логічний метод `isPopupTrigger()`, показуючий, що наступивша подія миші викликає появу спливаючого меню. Його потрібно викликати при настанні всякої події миші, щоб перевіряти, чи не являється вона сигналом до появи спливаючого меню, тобто звернення до методу `show()`. Було б занадто незручно включати таку перевірку в усім методів класів-слухачів подій миші. Тому метод `isPopupTrigger()` краще викликати в методі `processMouseEvent()`.

Пропонуємо ще раз програму рисування із лістинга 12.4, ввівши в клас `scribble` спливаюче меню для вибора кольору рисування і очистки вікна і змінивши обробку подій миші. Для простоти видалимо рядок меню, хоча його можна було залишити. Результат показаний в лістингі 13.2, а на рис. 13.3 — вигляд спливаючого меню в [MS Windows](#).

Лістинг 13.2. Програма рисування з спливаючим меню

```
import java.awt.*;
import java.awt.event.*;
public class PopupMenuScribble extends Frame{
public PopupMenuScribble(String s){ super (s) ;
ScrollPane pane = new ScrollPane();
pane.setSize(300, 300);
add(pane, BorderLayout.CENTER);
Scribble scr = new Scribble(this, 500, 500);
pane.add(scr);
addWindowListener(new WinClose());
pack ();
setVisible(true);
}
class WinClose extends WindowAdapter{
public void windowClosing(WindowEvent e) {
System.exit(0);
}
}
public static void main(String[] args){
new PopupMenuScribble(" \"Рисувалка\" з спливаючим меню");
}
}
class Scribble extends Component implements ActionListener{
protected int lastX, lastY, w, h;
protected Color currColor = Color.black;
protected Frame f;
protected PopupMenu c;
public Scribble(Frame frame, int width, int height){
f = frame; w = width; h = height;
enableEvents(AWTEvent.MOUSE_EVENT_MASK|AWTEvent.MOUSE_MOTION_EVENT_MASK);
c = new PopupMenu ("Color") ;
add(c);
MenuItem clear = new MenuItem("Clear", new MenuShortcut(KeyEvent.VK_C));
MenuItem red = new MenuItem("Red");
MenuItem green = new MenuItem("Green");
MenuItem blue = new MenuItem("Blue");
MenuItem black = new MenuItem("Black");
c.add(red); c.add(green); c.add(blue);
```

```

c.add(black); c.addSeparator(); c.add(clear);
red.addActionListener(this);
green.addActionListener(this);
blue.addActionListener(this);
black.addActionListener(this);
clear.addActionListener(this);
}
public Dimension getPreferredSize()
{
return new Dimension(w, h);
}
public void actionPerformed(ActionEvent event){
String s = event.getActionCommand();
if (s.equals("Clear")) repaint();
else if (s.equals("Red")) currColor = Color.red;
else if (s.equals("Green")) currColor = Color.green;
else if (s.equals("Blue")) currColor = Color.blue;
else if (s.equals("Black")) currColor = Color.black;
}
public void processMouseEvent(MouseEvent e){
if (e.isPopupTrigger())
c.show(e.getComponent (), e.getX(), e.getY());
else if (e.getID() == MouseEvent.MOUSE_PRESSED){
lastX = e.getX(); lastY = e.getY(); }
else super.processMouseEvent(e); }
public void processMouseMotionEvent(MouseEvent e){
if (e.getID() == MouseEvent.MOUSE_DRAGGED){
Graphics g = getGraphics();
g.setColor(currColor);
g.drawLine(lastX, lastY, e.getX(), e.getY());
lastX = e.getX(); lastY = e.getY();
}
else super.processMouseMotionEvent(e);
}
}
}

```



Рис. 13.3. Програма рисування з спливаючим меню

Лабораторна робота 12. Аплікація з використанням меню

1. Оюеднайте програми побудови кривих другого порядку в одну з викликом процедур побудови кривої з меню.

Урок 14

Аплети

- Передача параметрів
- Параметри тега `<applet>`
- Відомості про аплет
- Зображення і звук
- Слідкування за процесом завантаження
- Клас `MediaTracker`
- Захист від аплету
- Заключення

14.1. Аплет

До сих пір ми створювали додатки ([applications](#)), працюючі самостійно ([standalone](#)) в [JVM](#) під управлінням графічної оболонки операційної системи. Ці додатки мали власне вікно верхнього рівня типу [Frame](#), зареєстроване у віконному менеджері ([window manager](#)) графічної оболонки. Крім додатків, мова [Java](#) дозволяє створювати аплети ([applets](#)). Це програми, що працюють в середовищі іншої програми - браузера. Аплету не потрібне вікно верхнього рівня - ім служить вікно браузера. Вони не запускаються [JVM](#) — їх завантажує браузер, котрий сам запускає [JVM](#) для виконання аплету. Ці особливості відбуваються на написанні програми аплета.

З точки зору мови [Java](#), аплет — це всяке розширення класу [Applet](#), котрий, в свою чергу, розширяє клас [Panel](#). Таким чином, аплет - це панель спеціального виду, контейнер для розміщення компонентів з додатковими властивостями і методами. Менеджером розміщення компонентів по замовчуванню, як і в класі [Panel](#), служить [FlowLayout](#). Клас [Applet](#) знаходиться в пакеті [java.applet](#), в якому крім нього єсть тільки три інтерфейси, реалізовані в браузері. Треба відмітити, що не всі браузери реалізують ці інтерфейси повністю.

Оскільки [JVM](#) не запускає аплет, відпадає необхідність в методі [main\(\)](#), його немає в аплетах. В аплетах рідко зустрічається конструктор. Справа в тому, що при запуску першого створюється його контекст. Під час виконання конструктора контекст ще не сформований, тому не всі початкові значення вдається визначити в конструкторі. Поштовхі дії, зазвичай виконувані в конструкторі і методі [main\(\)](#), в аплеті записуються в метод [init\(\)](#) класу [Applet](#). Цей метод автоматично запускається виконуючою системою [Java](#) браузера зразу ж після завантаження аплета. Ось як він виглядає у вихідному коді класу [Applet](#):

```
public void init(){}
```

Не густо! Метод [init\(\)](#) не має аргументів, не повертає значення і повинен перевизначатися в кожному аплеті — підкласі класу [Applet](#). Зворотні дії — завершення роботи, звільнення ресурсів — записуються при необхідності в метод [destroy\(\)](#), також виконуваний автоматично при вивантаженні аплету. В класі [Applet](#) єсть пуста реалізація цього методу.

Крім методів [init\(\)](#) і [destroy\(\)](#) в класі [Applet](#) присутні ще два пустих методи, виконувані автоматично. Браузер повинен звертатися до методу [start\(\)](#) при кожній появлі аплету на екрані і звертатися до методу [stop\(\)](#), коли аплет зникає з екрана. В методі [stop\(\)](#) можна визначити дії, зупиняючі роботу аплета, в методі [start\(\)](#) — відновлюючі її. Треба зразу ж відмітити, що не всі браузери звертаються до цих методів як повинно. Так, перший із розглянутих нижче аплетів [HelloWorld.html](#) мені не вдалося запустити браузером [Internet Explorer](#), прийшлося скористатися власним [Java](#) браузером [appletviewer](#), як пояснено далі.

Роботу указаних методів можна пояснити простим житійським прикладом. Приїхавши весною на дачу, ви прокладаєте водопровідні труби, прикручуете крани, протягуете шланги - виконуєте метод [init\(\)](#) для своєї зрошувальної системи. Після цього, приходячи на ділянку, включаете крани — запускаєте метод [start\(\)](#), а виходячи, виключаете їх — виконуєте метод [stop\(\)](#). Нарешті, весни ви розбираєте зрошувальну систему, відкручуете крани, просушуєте і укладаєте водопровідні труби - виконуєте метод [destroy\(\)](#).

14.2. Найпростіший аплет

Перераховані вище методи `init()`, `start()`, `stop()`, `destroy()` не є обов'язковими при написанні простих аплетів, які не займають багато пам'яті, як свідчить наступний приклад. В лістингі 14.1 записаний простенький аплет, виконуючий вічну програму `HelloWorld`.

Лістинг 14.1. Аплет `HelloWorld`

```
import java.awt.*;
import java.applet.*;
public class HelloWorld extends Applet{
public void paint(Graphics g){
g.drawString("Hello, XXI century World!", 10, 30);
}
}
```

Ця програма записується в файл `HelloWorld.java` і компілюється зазвичай: `javac HelloWorld.java`.

14.3. Виконання аплета

Компілятор створює файл `HelloWorld.class`, але скористатися для його виконання інтерпретатором `java` тепер не можна, немає методу `main()`. Замість інтерпретації треба дати вказівку браузеру для запуску аплета. Всі вказівки браузеру даються помітками, тегами (`tags`), на мові **HTML (HyperText Markup Language)**. Зокрема, вказівка на запуск аплета дається в тезі `<applet>`. В ньому обов'язково задається ім'я файла з класом аплета параметром `code`, ширина `width` і висота `height` панелі аплета в пікселях. Повністю текст **HTML** для нашого аплета приведений в лістингі 14.2.

Лістинг 14.2. Файл **HTML** для завантаження аплета `HelloWorld`

```
<html>
<head><title> Applet</title></head> <body>
<br>
<applet code = "HelloWorld.class" width = "200" height = "100">
</applet>
</body>
</html>
```

Цей текст заноситься в файл з розширенням `html` або `htm`, наприклад `HelloWorld.html`. Ім'я файла довільне, ніяк не звязано з аплетом або класом аплета. Обидва файли — `HelloWorld.html` і `HelloWorld.class` — поміщаються в один каталог на сервері, і файл `HelloWorld.html` завантажується в браузер, який може знаходитися в будь-якому місці **Internet**. Браузер, проглядаючи **HTML**-файл, виконав тег `<applet>` і завантажить аплет. Після завантаження аплет зявиться у вікні браузера, як показано на рис. 14.1.

В цьому простому прикладі можна помітити ще дві особливості аплетів. По-перше, розмір аплета задається не в ньому, а в тезі `<applet>`. Це дуже зручно, можна змінювати розмір аплета, не компілюючи його заново. Можна організувати аплет невидимим, зробивши його розміром в один піксель. Крім того, розмір аплета дозволяється задати в процентах по відношенню до розміру вікна браузера, наприклад, `<applet code = "HelloWorld.class" width = "100%" height = "100%">`.

По-друге, як видно на рис. 14.1, у аплета сірий фон. Такий фон був у перших браузерів, і аплет не виділявся із тексту у вікні браузера. Тепер у браузерах прийнято білий фон, якого можна установити звичайним для компонентів методом `setBackground(Color.white)`, звернувшись до нього в метод `init()`. В склад **JDK** будь-якої версії входить програма `appletviewer`. Це найпростіший браузер, призначений для запуску аплетів з метою налаштування. Якщо під рукою немає **Internet**-браузера, можна скористуватися ним. `Appletviewer` запускається з командного рядка:

`appletviewer HelloWorld.html`



Рис. 14.2. appletviewer показывает аплет HelloWorld.

Ви звернули увагу, що внизу вікна знаходиться напис **Applet started**, якого ми ніде не програмували. Якби ми змінили колір вікна аплета, а це ми зробимо згодом, то побачили б, що напис **Applet started** знаходиться на окремій смужці, так званій **status bar**. В ній програма запуску аплета **appletviewer** і інформує користувача, про стан аплета, в даному випадку він стартував.

14.4. Приклад більш складного аплета

Як тільки що було сказано, у нижньому рядку браузера — рядку стану ([status bar](#)) — відображаються дані про завантаження файлів. Аплет може записати в нього будь-який рядок `str` методом `showStatus(String str)`. В лістингі 14.3 приведено аплет, записуючий в рядок стан браузера "біжучий рядок", а в лістингі 14.4 — відповідний [HTML](#)-файл. Перш ніж аналізувати наступну програму, ознайомтеся з методами `substring()` і `charAt()` класу `String`.

Лістинг 14.3. Біжуний рядок в рядку стану браузера

```
// Файл RunningString.Java
import java.awt.*;
import java.applet.*;
public class RunningString extends Applet{

private boolean go; //роздширяємо клас Applet полем go

public void start(){// реалізуємо пустий метод start() класу Applet
go = true;
sendMessage("This string is printed automatically by applet "); /* реалізацію
метода sendMessage(String s) дивись нижче*/
}

public void sendMessage(String s){// реалізуємо власний метод sendMessage(String s)
String s1 = s+" "; // До String s додається пробіл
while(go){ //Якщо аплет стартував, то String s висвічується у status bar
showStatus(s);

try{ /*блок try .. catch на випадок збою, при використанні класу Thread є
обов'язковим, бажаючі можуть прочитати про це в уроці 17, який ми, за браком часу,
цього 2008 року розглянути не зможемо*/
Thread.sleep(200); /*так робиться пауза - метод sleep() статичний, тому об'єкт класу
Thread не створюємо*/
}
}
```

```

catch(Exception e){}

s = s1.substring(1)+s.charAt(0); /*Перший символ рядка переноситься в його кінець – починається формуватися копія рядка позаду оригіналу – ефект рухомого рядка.*/
s1 = s; // з одержаним рядком операція повторюється в циклі while
}
} //Закінчується процедура формування копії рядка

public void stop(){ /* метод stop() зараз не рекомендують застосовувати, так як при роботі в мережі він може вплинути на стан і інших моніторів. Аплет спрацює і без цього методу – перевірте*/
go = false;
}

}

```

Лістинг 14.4. Файл RunningString.html

```

<html>
<head> <title> Applet</title></head>
<body>
<br>
<applet code = "RunningString.class" width = "1" height = "1">
</applet>
</body>
</html>

```

На жаль, немає строгого стандарту на виконання аплетів, і браузери можуть запускати їх по-різному. Програма `appletviewer` здатна показати аплет не так, як браузери. Приходиться перевіряти аплети на всіх наявних браузерах, добиваяючись однакового виконання.

14.5. Аплет, створюючий вікно

Приведемо більш складний приклад. Аплет `showWindow` створює вікно `someWindow` типу `Frame`, в якому розташоване поле введення типу `TextField`. В нього вводиться текст, і після натискання клавіші `<Enter>` переноситься в поле введення аплета. В аплеті присутня кнопка. Після кліку кнопкою миші по ній вікно `someWindow` то зникає з екрана, то знову з'являється на ньому. Програма приведена в лістингах 14.5 і 14.6, результат — на рис. 14.3.

Лістинг 14.5. Аплет, створюючий вікно

```

// Файл ShowWindow.java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ShowWindow extends Applet{
private SomeWindow sw = new SomeWindow();
private TextField tf = new TextField(30);
private Button b = new Button("Hide");
public void init(){
add(tf); add(b); sw.pack();
b.addActionListener(new ActShow());
sw.tf.addActionListener(new ActShow());
}
public void start(){ sw.setVisible(true);}
public void stop(){ sw.setVisible(false);}
public void destroy(){
sw.dispose(); sw = null; tf = null; b = null;
}

```

```

public class ActShow implements ActionListener{
public void actionPerformed(ActionEvent ae){
if (ae.getSource() = sw.tf)
tf.setText(sw.tf.getText());
else if (b.getActionCommand() == "Show"){
sw.setVisible(true);
b.setText("Hide");
}
else{
sw.setVisible(false);
b.setText("Show");
}
}
}
}
}
}
class SomeWindow extends Frame{
public TextField tf = new TextField(50);
SomeWindow(){
super(" Вікно введення");
add(new Label("Input, please, your name"), "North");
add(tf, "Center");
}
}

```

Лістинг 14.6. Файл ShowWindow.html

```

<html>
<head><title> ShowWindow Applet</title></head>
<body>
<br>
<applet code = "ShowWindow.class" width = "400" height = "50">
</applet>
</body>
</html>

```



Рис. 14.3. Апплет, створюючий вікно

Зауваження по налаштуванню

Браузери поміщають завантажені апплети в свій кеш, тому після кліку кнопкою миші по кнопці **Refresh** або

Reload запускається стара копія аплета із кеша. Для завантаження нової копії треба при кліку по кнопці Refresh в IE (Internet Explorer) тримати натиснутою клавішу **<Ctrl>**, а при кліку по кнопці Reload в NC (Netscape Communicator) — клавішу **<Shift>**. Інколи і це не допомагає. Не врятує навіть перезапуск браузера. Тоді треба очистити обидва кеші — і дисковий, і кеш в пам'яті. В IE це виконується кнопкою Delete Files у вікні, викликаному набором команди Tools | Internet Options. В NC необхідно відкрити вікно Cache командою Edit | Preferences | Advanced.

При запуску додатку інтерпретатором `java` із командного рядка в нього можна передати параметри у вигляді аргумента метода `main(String [] args)`. В аплети також передаються параметри, але іншим шляхом.

14.3. Передача параметрів

Передача параметрів в аплет відбувається за допомогою тегів `<param>`, розташованих між відкриваючим тегом `<applet>` і закриваючим тегом `</applet>` в HTML-файлі. В тегах `<param>` указується назва параметра `name` і його значення `value`. Передамо, наприклад, в наш аплет `HelloWorld` параметри шрифту. В лістингі 14.7 показано змінений файл `HelloWorld.html`.

Лістинг 14.7. Параметри для передачі в аплет

```
<html>
<head><title> Applet</title></head>
<body>
<br>
<applet code = "HelloWorld.class" width = "400" height = "50"> /*Далі йде те, що
буде передано в аплет програмою appletviewer*/
<param name = "fontName" value = "Serif">
<param name = "fontSize" value = "2">
<param name = "fontStyle" value = "30">
</applet>
</body>
</html>
```

В аплеті для прийому кожного параметра треба скористатися методом `getParameter (String name)` класу `Applet`, повертуючому рядок типу `String`. В якості аргумента цього методу задається значення параметра `name` у вигляді рядка, причому тут не розрізняється регистр літер, а метод повертає значення параметра `value` теж у вигляді рядка.

Зауваження по налаштуванню

Оператори `System.out.println()`, зазвичай записувані в аплет для налаштування, виводять указані в них аргументи в спеціальне вікно браузера `Java Console`. Спочатку треба установити можливість показу цього вікна. В Internet Explorer це робиться установкою пропорці `Java Console enabled` набором команд `Tools | Internet Options | Advanced`. Після перезапуску IE в меню `View` з'являється команда `Java Console`.

В лістингі 14.8 показано перероблений аплет `HelloWorld`. В ньому призначений білий фон, а шрифт установлюється з параметрами, добутими із HTML-файла.

Лістинг 14.8. Аплет, приймаючий параметри

```
import java.awt.*;
import java.applet.*;
public class HelloWorld extends Applet{ public void init(){
setBackground(Color.white);
String font = "Serif";
int style = Font.PLAIN, size = 10; /*наступні дані аплет отримає від програми
appletviewer з файлу HelloWorld.html*/
font = getParameter("fontName");
style = Integer.parseInt(getParameter("fontStyle"));
```

```

size = Integer.parseInt(getParameter("fontSize"));
setFont(new Font(font, style, size));
}
public void paint(Graphics g){
g.drawString("Hello, XXI century World!", 10, 30);
}
}

```

Порада

Сподіваючись на те, що параметри будуть задані в [HTML](#)-файлі, все-таки присвойте початкові значення змінним в аплеті, як це зроблено в лістингі 14.8. На рис. 14.4 показано працюючий аплет.

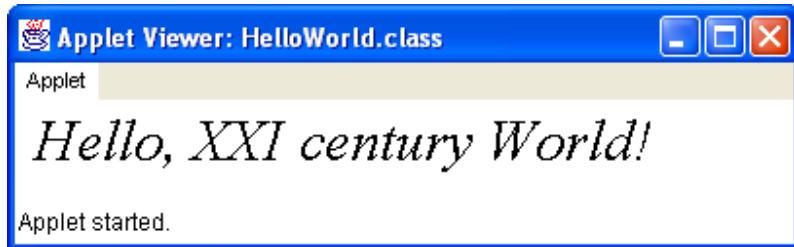


Рис. 14.4. Аплет із зміненим шрифтом

Правила хорошого тону рекомендують описати параметри, передавані аплету, у вигляді масиву, кожний елемент якого — масив із трьох рядків, відповідаючий одному параметру. Дано структура представляється у вигляді "ім'я", "тип", "опис". Для нашого прикладу можна написати:

```

String[][] pinfo = {
{"fontName", "String", "font name"},
 {"fontStyle", "int", "font style"},
 {"fontSize", "int", "font size"}
};

```

Потім перевизначається метод `getParameterInfo()`, повертуючий указаний масив. Це пустий метод класу `Applet`. Будь-який об'єкт, бажаючий узнати, що передавати аплету, може викликати цей метод. Для нашого прикладу перевизначення вигляє так:

```

public String[][] getParameterInfo() {
    return pinfo;
}

```

Крім того, правила хорошого тону приписують перевизначити метод `getAppletInfo()`, повертуючий рядок, в який записано ім'я автора, версія аплета і інші дані про аплет, котрі ви хотіли б представити всім бажаючим. Наприклад:

```

public String getAppletInfo() {
    return "MyApplet v.1.5 P.S. Ivanov";
}

```

Подивимось тепер, які ще параметри можна задати в тезі `<applet>`.

14.4. Параметри тега `<applet>`

Перечислимо всі параметри тега `<applet>`.

Обов'язкові параметри:

- `code` — URL-адреса файла з класом аплета або архівного файла;
- `width` і `height` — ширина і висота аплета в пікселях.

Необовязкові параметри:

- `codebase` — URL-адреса каталога, в якому розташований файл класу аплета. Якщо цей параметр відсутній, браузер буде шукати файл в тому ж каталозі, де розміщений відповідний HTML-файл;
- `archive` — файли всіх класів, складаючих аплет, можуть бути упаковані архіватором ZIP або спеціальним архіватором JAR в один або декілька архівних файлів. Параметр задає URL-адреси цих файлів через кому;
- `align` — вирівнювання аплета у вікні браузера. Цей параметр має одне із наступних значень: ABSBOTTOM, ABSMIDDLE, BASELINE, BOTTOM, CENTER, LEFT, MIDDLE, RIGHT, TEXTTOP, TOP;
- `hspace` і `vspace` — горизонтальні і вертикальні поля, відокремлюючі аплет від інших об'єктів у вікні браузера в пікселях;
- `download` — задає порядок завантаження зображень аплетом. Імена зображень перечисляються через кому в порядку завантаження;
- `name` - ім'я аплета. Параметр потрібний, якщо завантажуються декілька аплетів з однаковими значеннями `code` і `codebase`;
- `style` — інформація про стиль CSS (Cascading Style Sheet); `title` — текст, відображуваний в процесі виконання аплета;
- `alt` — текст, що виводиться замість аплета, якщо браузер не може завантажити його;
- `mayscript` — не має значення. Це слово указує на те, що аплет буде звертатися до тексту JavaScript.

Між тегами `<applet>` і `</applet>` можна написати текст, який буде виведений, якщо браузер не зможе зрозуміти тег `<applet>`. Ось повний приклад:

```
<applet name = "AnApplet" code = "AnApplet.class"
archive = "anapplet.zip, myclasses.zip"
codebase = "http://www.some.com/public/applets"
width = "300" height = "200" align = "TOP"
vspace = "5" hspace = "5" mayscript
alt = "If you have a Java-enabled browser, you would see an applet here.">
<hr>If your browser recognized the applet tag, you would see an applet here.<hr>
</applet>
```

Порада

Обовязково упаковуйте всі класи аплета в zip- і rar-архіви і вкажіть їх в параметрі `archive` в HTML-файлі. Це значно прискорить завантаження аплета. Слід ще сказати, що, починаючи з версії HTML 4.0, єсть тег `<object>`, призначений для завантаження і аплетів, і інших об'єктів, наприклад, ActiveX. Крім того, деякі браузери можуть використовувати для завантаження аплетів тег `<embed>`. Ми уже згадували, що при завантаженні аплета браузер створює контекст, в якому збирає всі дані, необхідні для виконання аплета. Деякі дані із контексту можна передати в аплет.

14.5. Дані про оточення аплета

Метод `getCodeBase()` повертає URL-адресу каталога, в якому лежить файл класу аплета.

Метод `getDocumentBase()` повертає URL-адресу каталога, в якому лежить HTML-файл, викликавший аплет. Браузер реалізує інтерфейс `AppletContext`, що знаходиться в пакеті `java.applet`. Аплет може отримати посилку на цей інтерфейс методом `getAppletContext()`. За допомогою методів `getApplet (String name)` і `getApplets()` інтерфейса `AppletContext` можна отримати посилку на указаний аргументом `name` аплет або на всі аплети, завантажені в браузер.

Метод `showDocument(URL address)` завантажує в браузер HTML-файл з адресою `address`.

Метод `showDocument (URL address, String target)` завантажує файл у фрейм, указаний другим аргументом `target`. Цей аргумент може приймати наступні значення:

- `_self` — те ж вікно і той же фрейм, в якому працює аплет;
- `_parent` — батьківський фрейм аплета;
- `_top` — фрейм верхнього рівня вікна аплета;
- `_blank` — нове вікно верхнього рівня;
- `name` — фрейм або вікно з іменем `name`, якщо воно не існує, то буде створено.

14.6. Зображення і звук

Зображення в Java — це об'єкт класу `Image`, представляючий прямокутний масив пікселів. Його можуть показати на екрані логічні методи `drawImage()` класу `Graphics`. Ми розглянемо їх детально в наступному уроці, а поки що нам знадобляться два логічні методи:

- `drawImage(Image img, int x, int y, ImageObserver obs)`
- `drawImage(Image img, int x, int y, int width, int height, ImageObserver obs)`

Методи починають рисувати зображення, не чекаючи закінчення завантаження зображення `img`. Більше того, завантаження не почнеться, поки не викликаний метод `drawImage()`. Методи повертають `false`, поки завантаження не закінчиться. Аргументи `(x, y)` задають координати лівого верхнього кута зображення `img`; `width` і `height` — ширину і висоту зображення на екрані; `obs` — посилання на об'єкт, реалізуючий інтерфейс `ImageObserver`, спідкуючий за процесом завантаження зображення. Останньому аргументу можна дати значення `this`.

Перший метод задає на екрані такі ж розміри зображення, як і у об'єкта класу `Image`, без змін. Одержані ці розміри можна методами `getWidth()`, `getHeight()` класу `Image`. Інтерфейс `ImageObserver`, реалізований класом `Component`, а значить, і класом `Applet`, описує тільки один логічний метод `imageUpdate()`, виконуваний при кожній зміні зображення. Саме цей метод побуджує перерисовувати компонент на екрані при кожній його зміні. Подивимося, як його можна використовувати в процесі завантаження файлів із Internet.

14.7. Відслідковування процесу завантаження

Якщо ви хоч раз бачили, як зображення завантажується із Internet, то помітили, що воно появляється на екрані по частинах по мері завантаження. Це відбувається в тому випадку, коли системна властивість `awt.image.incrementalDraw` має значення `true`. При поступанні кожної порції зображення браузер викликає логічний метод `imageUpdate()` інтерфейса `ImageObserver`. Аргументи цього методу містить інформацію про процес завантаження зображення `img`. Розглянемо їх:

`imageUpdate(Image img, int status, int x, int y, int width, int height);`

Аргумент `status` містить інформацію про завантаження у вигляді одного цілого числа, яке можна порівняти з наступними константами інтерфейса `ImageObserver`:

- `WIDTH` — ширина уже завантаженої частини зображення відома, і може бути одержана із аргументу `width`;
- `HEIGHT` — висота уже завантаженої частини зображення відома, і може бути одержана із аргументу `height`;
- `PROPERTIES` — властивості зображення уже відомі, їх можна одержати методом `getProperties()` класу `Image`;
- `SOMEBITS` — одержані пікселі, достатні для рисування маштабованої версії зображення; аргументи `x, y, width, height` визначені;
- `FRAMEBITS` — одержаний наступний кадр зображення, що містить декілька кадрів; аргументи `x, y, width, height` не визначені;
- `ALLBITS` — всезображення одержано, аргументи `x, y, width, height` не містять інформації;
- `ERROR` — завантаження зупинено, рисування перервано, визначений біт `ABORT`;
- `ABORT` — завантаження перервано, рисування призупинено до приходу наступної порції зображення.

Ви можете перевизначити цей метод в своєму аплеті і використовувати його аргументи для слідкування за процесом завантаження і визначення моменту повного завантаження.

Інший спосіб відслідкувати закінчення завантаження - скористатися методами класу [MediaTracker](#). Вони дозволяють перевірити, чи не закінчено завантаження, або призупинити роботу аплета до закінчення завантаження. Один екземпляр класу [MediaTracker](#) може слідкувати за завантаженням декількох зареєстрованих в ньому зображень.

14.8. Клас [MediaTracker](#)

Спочатку конструктором [MediaTracker](#) (`Component comp`) створюється об'єкт класу для указаного аргументом компонента. Аргумент конструктора частіше всього `this`. Потім методом `addImage(Image img, int id)` реєструється зображення `img` під порядковим номером `id`. Декілька зображень можна зареєструвати під одним номером. Після цього логічними методами `checkID(int id)`, `checkID(int id, boolean load)` і `checkAll()` перевіряється, чи завантажено зображення з порядковим номером `id` або всі зареєстровані зображення. Методи повертають `true`, якщо зображення уже завантажено, `false` — в протилежному випадку. Якщо аргумент `load` рівний `true`, то відбувається завантаження всіх ще не завантажених зображень. Методи `statusID(int id)`, `statusID(int id, boolean load)` і `statusAll` повертають ціле число, яке можна порівняти із статичними константами `COMPLETE`, `ABORTED`, `ERRORED`. Нарешті, методи `waitForID(int id)` і `waitForAll()` чекають закінчення завантаження зображення. В наступному уроці в лістинзі 15.5 ми застосуємо ці методи для очікування завантаження зображення.

Зображення, що знаходитьться в об'єкті класу `image` можна створити безпосередньо по пікселям, а можна одержати із графічного файла, типу [GIF](#) або [JPEG](#), одним із двох методів класу [Applet](#):

- `getImage(URL address)` — задається `URL`-адреса графічного файла;
- `getImage(URL address, String fileName)` — задається адреса каталога `address` і імя графічного файла `filename`.

Аналогічно, звуковий файл в аплетах представляється у вигляді об'єкта, реалізуючого інтерфейс [AudioClip](#), і може бути отриманий із файла типу [AU](#), [AIFF](#), [WAVE](#) або [MIDI](#) одним із трьох методів класу [Applet](#) з такими ж аргументами:

```
getAudioClip(URL address)
getAudioClip(URL address, String fileName)
newAudioClip(URL address)
```

Останній метод статичний, його можна використовувати не тільки в аплетах, але і в додатках. Інтерфейс [AudioClip](#) із пакета [java.applet](#) дуже простий. В ньому всього три методи без аргументів. Метод `play()` програє мелодію один раз. Метод `loop()` нескінчено повторяє мелодію. Метод `stop()` зупиняє програвання. Цей інтерфейс реалізується браузером. Звичайно, перед програвнням звукових файлів браузер повинен бути звязаний із звуковою системою комп'ютера.

В лістинзі 14.9 приведено простий приклад завантаження зображення і звуку із файлів, що знаходяться в тому ж каталозі, що і [HTML](#)-файл. На рис. 14.5 показано, як виглядає зображення, збільшене в два рази.

Лістинг 14.9. Звук і зображення в аплеті

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class SimpleAudioImage extends Applet{
private Image img;
private Audioclip ac;
public void init(){
img = getImage(getDocumentBase(), "gifIcon.gif");
ac = getAudioClip(getDocumentBase(), "spacemusic.au");
}
public void start (){ ac.loop();}
```

```

}
public void paint(Graphics g){
int w = img.getWidth(this), h = img.getHeight(this);
g.drawImage(img, 0, 0, 2 * w, 2 * h, this); }
public void stop() { ac.stop(); }
}

```



Рис. 14.5. Виведення зображення у звуковому супроводі

Жаль, що ви не можете з тексту лекції насолоджуватися звуками космічної музики.

Перед виведенням на екран зображення можна перетворити, але про це поговоримо в наступному уроці. Як бачите, аплету в браузері дозволено дуже мало. Це не випадково. Аплет, що зявився в браузері звідкись із Internet, може натворити немало бід. Він може бути викликаний із файла з цікавим текстом, невидимо обшукувати файлову систему і викрасти секретні дані, або, навпаки, відкрити вікно, таке ж, в яке ви вводите пароль, і перехватити його. Тому браузер повідомляє при завантаженні аплета: "Applet started", а в рядку стану вікна, відкритого аплетом, зявляється напись: "Warning: Applet Window". Але це не єдиний захист від аплета. Розглянемо цю проблему детальніше.

14.9. Захист від аплета

Браузер може взагалі відмовитися від завантаження аплетів. В *Netscape Communicator* це робиться за допомогою пропорції **Enable Java** у вікні, викликаному командою **Edit | Preferences | Advanced**, в *Internet Explorer* — у вікні після вибору команди **Tools | Internet Options | Security**. В такому випадку говорити більше немає про що. Якщо ж браузер завантажує аплет, то створює йому обмеження, так звану "пісочницю" (**sandbox**), в яку попадає аплет, але вийти з якої не може. Кожний браузер створює свої обмеження, але як правило вони заключаються в тому, що аплет:

- не може звертатися до файлової системи машини, на якій він виконується, навіть для читання файлів або перегляду каталогів;
- може звязатися па мережі тільки з тим сайтом, з якого він був завантажений;
- не може прочитати системні властивості, як це робить, наприклад, додаток в лістингі 6.4;
- не може друкувати на принтері, підключеному до того комп'ютера, на якому він виконується;
- не може користуватися буфером обміну (**clipboard**);
- не може запустити додаток методом **exec()**;
- не може використовувати "рідні" методи або завантажити бібліотеку методом **load()**;
- не може зупинити **JVM** методом **exit()**;
- не може створювати класи в пакетах **java.***, а класи пакетів **sun.*** не може навіть завантажувати.

Браузери можуть посилити або ослабити ці обмеження, наприклад, дозволити локальним аплетам, завантаженим з тієї ж машини, де вони виконуються, доступ до файлової системи. Найменшим обмеженням підлягають **довірені** (**trusted**) аплети, що мають електронний підпис за допомогою класів із пакетів **java.security.***.

При створені додатку, що завантажує аплети, необхідно забезпечити засоби перевірки аплета і задати обмеження. Їх дає в розпорядження клас **securityManager**. Екземпляр цього класу або його потомка

установлюється в **JVM** при запуску віртуальної машини статичним методом `setSecurityManager(SecurityManager sm)` класу `System`. Звичайні додатки не можуть використовувати даний метод. Кожний браузер розширяє клас `SecurityManager` по-своєму, установлюючи ті чи інші обмеження. Єдиний екземпляр цього класу створюється при запуску **JVM** в браузері і не може бути змінений.

Заключення

Аплети були спершу практичним застосуванням **Java**. За перші два роки існування **Java** були написані тисячі дуже цікавих і красивих аплетів, ожививших **WWW**. Маса аплетів розкидана по **Internet**, хороши приклади аплетів зібрані в **JDK** в каталозі `demo\applets`. В **JDK** увійшов цілий пакет `java.applet`, в який фірма **SUN** збиралась заносити класи, що розвивають і покращують аплети. Із збільшенням швидкості і покращенням якості комп'ютерних мереж значення аплетів сильно упало. Тепер вся обробка даних, раніше виконувана аплетами, переноситься на сервер, браузер тільки завантажує і показує результати цієї обробки, становиться "тонким клієнтом".

З іншого боку, зявилось багато спеціалізованих програм, в тому числі написаних на **Java**, завантажуючих інформацію із **Internet**. Така можливість єсть зараз у всіх музичних і відеопрограмах. Фірма **SUN** більше не розвивє пакет `java.applet`. В ньому так і залишився один клас і три інтерфейси. В бібліотеку **Swing** увійшов клас `JApplet`, розширяючий клас `Applet`. В ньому є додаткові можливості, наприклад, можна установити систему меню. Він здатний використовувати всі класи бібліотеки **Swing**. Але більшість браузерів ще не мають **Swing** в своєму складі, тому приходиться завантажувати класи **Swing** із сервера або включати їх в **jar**-архів разом з класами аплета.

Лабораторна робота 13. Створення аплетів.

1. Переробити програму побудови кривих другого порядку в аплет.

Лістинг 14.1. Апплет HelloWorld

```
import java.awt.*;
import java.applet.*;
public class HelloWorld extends Applet{
public void paint(Graphics g){
g.drawString("Hello, XXI century World!", 10, 30);
} }
```

Лістинг 14.3. Біжучий рядок в рядку стану браузера

```
// Файл RunningString.Java
import java.awt.*;
import java.applet.*;
public class RunningString extends Applet{
private boolean go; //розширюємо клас Applet полем go
public void start() {// реалізуємо пустий метод start() класу Applet
go = true;
sendMessage("This string is printed automatically by applet "); /* реалізацію метода
sendMessage(String s) дивись нижче*/
}
public void sendMessage(String s){// реалізуємо власний метод sendMessage(String s)
String s1 = s+" "; // До String s додається пробіл
while(go){ //Якщо апплет стартував, то String s висвічується у status bar
showStatus(s);
try{ /*блок try .. catch на випадок збою, при використанні класу Thread є
обовязковим, бажаючі можуть прочитати про це в уроці 17, який ми, за браком
часу, цього 2008 року розглянути не зможемо*/
Thread.sleep(200); /*так робиться пауза – метод sleep()статичний, тому об'єкт класу
Thread не створюємо*/
}
catch(Exception e){}
s = s1.substring(1)+s.charAt(0); /*Перший символ рядка переноситься в його кінець
– починається формуватися копія рядка позаду оригіналу – ефект рухомого
рядка.*/
s1 =s; // з одержаним рядком операція повторюється в циклі while
}
} //Закінчується процедура формування копії рядка

public void stop(){ /* метод stop() зараз не рекомендують застосовувати, так як при
роботі в мережі він може вплинути на стан і інших моніторів. Апплет спрацює і без
цього методу - перевірте*/
```

```
go = false;
}}
```

Лістинг 14.5. Аплет, створюючий вікно

```
// Файл ShowWindow.java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ShowWindow extends Applet{
private SomeWindow sw = new SomeWindow();
private TextField tf = new TextField(30);
private Button b = new Button("Hide");
public void init(){
add(tf); add(b); sw.pack();
b.addActionListener(new ActShow());
sw.tf.addActionListener(new ActShow());
}
public void start(){ sw.setVisible(true);}
public void stop(){ sw.setVisible(false);}
public void destroy(){
sw.dispose(); sw = null; tf = null; b = null;
}
public class ActShow implements ActionListener{
public void actionPerformed(ActionEvent ae){
if (ae.getSource() = sw.tf)
tf.setText(sw.tf.getText());
else if (b.getActionCommand() == "Show"){
sw.setVisible(true);
b.setLabel("Hide");
}
else{
sw.setVisible(false);
b.setLabel("Show");
}
}
}
class SomeWindow extends Frame{
public TextField tf = new TextField(50);
SomeWindow(){
super(" Вікно введення");
add(new Label("Input, please, your name"), "North");
add(tf, "Center");
}
}
```

Лістинг 14.8. Апплет, приймаючий параметри

```

import java.awt.*;
import java.applet.*;
public class HelloWorld extends Applet{ public void init(){
setBackground(Color.white);
String font = "Serif";
int style = Font.PLAIN, size = 10; /*наступні дані апплет отримає від програми
appletviewer з файлу HelloWorld.html*/
font = getParameter("fontName");
style = Integer.parseInt(getParameter("fontStyle"));
size = Integer.parseInt(getParameter("fontSize"));
setFont(new Font(font, style, size));
}
public void paint(Graphics g){
g.drawString("Hello, XXI century World!", 10, 30);
}
}

```

Лістинг 14.9. Звук і зображення в апплеті

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class SimpleAudioImage extends Applet{
private Image img;
private Audioclip ac;
public void init(){
img = getImage(getDocumentBase(), "gifIcon.gif");
ac = getAudioClip(getDocumentBase(), "spacemusic.au"); }
public void start (){ ac.loop(); }
public void paint(Graphics g){
int w = img.getWidth(this), h = img.getHeight(this);
g.drawImage(img, 0, 0, 2 * w, 2 * h, this); }
public void stop() { ac.stop(); }
}

```

Листинг 14.2. Файл HTML для завантаження аплета HelloWorId

```
<html>
<head><title> Applet</title></head> <body>
<br>
<applet code = "HelloWorld.class" width = "200" height = "100">
</applet> </body> </html>
```

Листинг 14.4. Файл RunningString.html

```
<html>
<head> <title> Applet</title></head>
<body>
<br>
<applet code = "RunningString.class" width = "1" height = "1">
</applet> </body> </html>
```

Листинг 14.6. Файл ShowWindow.html

```
<html>
<head><title> ShowWindow Applet</title></head>
<body>
<br>
<applet code = "ShowWindow.class" width = "400" height = "50">
</applet> </body> </html>
```

Листинг 14.7. Параметри для передачі в аплет

```
<html>
<head><title> Applet</title></head>
<body>
<br>
<applet code = "HelloWorld.class" width = "400" height = "50"> /*Далі йде те, що
буде передано в аплет програмою appletviewer*/
<param name = "fontName" value = "Serif">
<param name = "fontStyle" value = "2">
<param name = "fontSize" value = "30">
</applet> </body> </html>
```

Урок 15

Зображення і звук

- Модель обробки "поставщик-споживач"
- Класи-фільтри
- Як віділити фрагмент зображення
- Як змінити колір зображення
- Як переставити пікселі зображення
- Модель обробки прямим доступом
- Перетворення зображення в Java 2D
- Афінне перетворення зображення
- Зміна інтенсивності зображення
- Зміна складових кольору
- Створення різноманітних ефектів
- Анимація
- Покращення зображення подвійною буферизацією
- Звук
- Програвання звуку в Java 2
- Синтез і запис звуку Java 2

Як уже згадувалось в попередньому уроці, зображення в Java — це об'єкт класу `Image`. Там же показано, як в аплетах застосовуються методи `getImage()` для створення цих об'єктів із графічних файлів. Додатки теж можуть застосовувати аналогічні методи `getImage()` класу `Toolkit` із пакета `java.awt` з одним аргументом типу `String` або `URL`. Звернення до цих методів із компонента виконується через метод `getToolkit()` класу `Component` і виглядає так:

```
Image img = getToolkit().getImage("C:\images\ivanov.gif");
```

В загальному випадку звернення можна зробити через статичний метод `getDefaultToolkit()` класу `Toolkit`:

```
Image img = Toolkit.getDefaultToolkit().getImage("C:\images\ivanov.gif");
```

Але, крім цих методів, клас `Toolkit` містить п'ять методів `createImage()`, повертаючих посилання на об'єкт типу `Image`:

- `createImage (String fileName)` — створює зображення із вмісту графічного файлу `filename`;
- `createImage (URL address)` — створює зображення із вмісту графічного файлу по адресу `address`;
- `createImage (byte [] imageData)` — створює зображення із масиву байтів `imageData`, дані в якому повинні мати формат `GIF` або `JPEG`;
- `createImage (byte [] imageData, int offset, int length)` — створює зображення із частини масива `imageData`, що починається з індекса `offset` довжиною `length` байтів;
- `createImage (ImageProducer producer)` — створює зображення, одержане від поставщика `producer`.

Останній метод єсть і в класі `Component`. Він використовує модель "поставщик-споживач" і вимагає детального пояснення.

15.1. Модель обробки "поставщик-споживач"

Дуже часто зображення перед виведенням на екран підлягає обробці: змінюються кольори окремих пікселів або цілих частин зображення, виділяються і перетворюються якісь фрагменти зображення. В бібліотеці `AWT` застосовуються дві моделі обробки зображення. Одна модель реалізує давно відому в програмуванні спільну модель "поставщик-споживач" (`Producer-Consumer`). Згідно цієї моделі один об'єкт, "поставщик", генерує сам або перетворює отриману із іншого місця продукцію, в даному випадку, набір пікселів, і передає іншим об'єктам. Ці об'єкти, "споживачі", приймають продукцію і теж перетворюють її при необхідності. Тільки після цього створюється об'єкт класу `Image` і зображення виводиться на екран. У одного поставщика може бути декілька споживачів, котрі повинні бути зареєстровані поставщиком. Поставщик і споживач активно взаємодіють, звертаючись до методів один одного.

В AWT ця модель описана в двох інтерфейсах: [ImageProducer](#) і [ImageConsumer](#) пакета `java.awt.image`. Інтерфейс [ImageProducer](#) описує п'ять методів:

- `addConsumer(ImageConsumer ic)` - реєструє споживача `ic`; `removeConsumer (ImageConsumer ic)` - скасовує реєстрацію;
- `isConsumer(ImageConsumer ic)` — логічний метод, перевіряє, чи зареєстрований споживач `ic`;
- `startProduction (ImageConsumer ic)` — реєструє споживача `ic` і починяє поставку зображення всім зареєстрованим споживачам;
- `requestTopDownLeftRightResend (ImageConsumer ic)` — використовується споживачем для того, щоб затребувати зображення ще раз в порядку "зверху-вниз, зліва-направо" для методів обробки, застосовуючи саме такий порядок.

З кожним екземпляром класу `Image` звязаний об'єкт, реалізуючий інтерфейс [ImageProducer](#). Його можна отримати методом `getSource()` класу `Image`. Найпростіша реалізація інтерфейса [ImageProducer](#) - клас `MemoryImageSource` — створює пікселі в оперативній пам'яті по масиву байтів або цілих чисел. Спочатку створюється масив `pix`, що містить колір кожної точки. Потім одним із шести конструкторів створюється об'єкт класу `MemoryImageSource`. Він може бути оброблений споживачем або прямо перетворений у тип `Image` методом `createImage ()`.

В лістингі 15.1 наведена проста программа, що виводить на екран квадрат розміром 100x100 пікселів. Лівий верхній кут квадрата синій, лівий нижній — червоний, правий верхній — зелений, а до центру квадрата кольори змішуються.

Лістинг 15.1. Зображення, побудоване по точках

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
class InMemory extends Frame {
private int w = 100, h = 100;
private int[] pix = new int[w * h];
private Image img;
InMemory(String s)

{ super(s);
int i = 0;
for (int y = 0; y < h; y++){
int red = 255 * y / (h - 1);
for (int x = 0; x < w; x++){
int green = 255 * x / (w - 1) ;
pix[i++] = (255 << 24) | (red << 16) | (green << 8) | 128; } }
setSize(250, 200);
setVisible(true);
}

public void paint(Graphics gr){
if (img == null)
img = createImage(new MemoryImageSource(w, h, pix, 0, w));
gr.drawImage(img, 50, 50, this);
}

public static void main(String[] args){

Frame f= new InMemory(" Зображення в пам'яті");

f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){ System.exit (0); }
});
```

```

}
}

```

В лістингі 15.1 в конструктор класу-постачальника `MemoryImageSource (w, h, pix, o, w)` заноситься ширина `w` і висота `h` зображення, масив `pix`, зміщення в цьому масиві `o` і довжина рядка `w`. Споживачем служить зображення `img`, яке створюється методом `createImage ()` і виводиться на екран методом `drawImage(img, 50, 50, this)`. Лівий верхній кут зображення `img` розташовується в точці `(50, 50)` контейнера, а останній аргумент `this` показує, що роль `imageObserver` відіграє сам клас `InMemory`. Це заставляє включити в метод `paint()` перевірку `if (img == null)`, інакше зображення буде постійно перерисовуватися. Другий спосіб уникнути цього — перевизначити метод `imageupdate()`, про що говорилося в уроці 14, просто написавши в ньому `return true`. Рис. 15.1 демонструє виведення цієї програми.

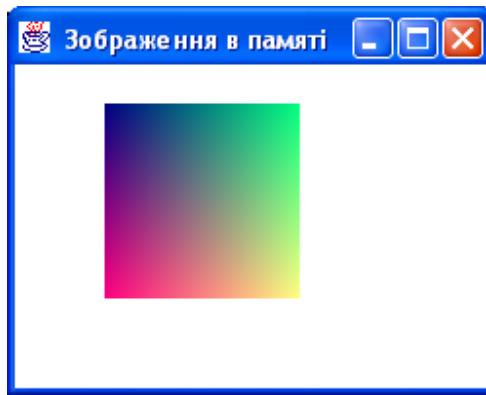


Рис. 15.1. Зображення, створене по точках

Інтерфейс `imageConsumer` описує сім методів, найважливішими із яких являються два методи `setPixels()`. Перший:

```
setPixels(int x, int y, int width, int height, ColorModel model, byte[] pix, int offset, int scansize);
```

Другий метод відрізняється тільки тим, що масив `pix` містить елементи типу `int`.

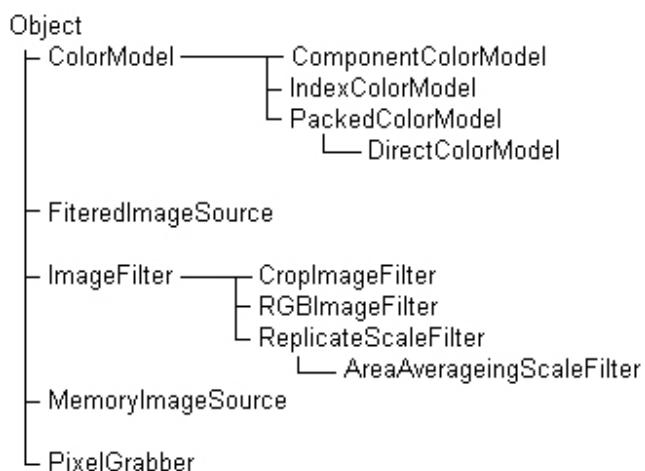


Рис. 15.2. Класи, реалізуючі модель "постачальник-споживач"

До цих методів звертається постачальник для передачі пікселів споживачу. Передається прямокутник шириною `width` і висотою `height` із заданим верхнім лівим кутом `(x, y)`, заповнений пікселями із масива `pix`, починаючи з індекса `offset`. Кожен рядок займає `scansize` елементів масиву `pix`. Колір пікселів визначається в кольоровій моделі `model` (звичайно це модель `RGB`). На рис. 15.2 показана ієархія класів,

реалізуючих модель "постачальник-споживач".

15.2. Класи-фільтри

Інтерфейс `imageConsumer` не має рациї реалізовувати, використовується його готова реалізація - клас `imageFilter`. Незважаючи на назву, цей клас не робить ніякої фільтрації, він передає зображення без зміни. Для перетворення зображення даний клас належить розширити, перевизначивши метод `setPixels()`. Результат перетворення належить передати споживачу, роль якого відіграє поле `consumer` цього класу. В пакеті `java.awt.image` є чотири розширення класу `ImageFilter`:

- `CropImageFilter (int x, int y, int w, int h)` — виділяє фрагмент зображення,ений в приведеному конструкторі;
- `RGBImageFilter` - дозволяє змінювати окрім пікселі; це абстрактний клас, він вимагає розширення і перевизначення свого методу `filterRGB()`;
- `ReplicateScaleFilter (int w, int h)` — змінює розміри зображення на указані в приведеному конструкторі, дублюючи рядки і/або стовпці при збільшенні розмірів або приираючи деякі із них при зменшенні;
- `AreaAveragingScaleFilter (int w, int h)` — розширення попереднього класу; використовує більш складний алгоритм зміни розмірів зображення, усереднюючий значення сусідніх пікселів.

Застосовуються ці класи разом із іншим класом-постачальником, реалізуючим інтерфейс `ImageProducer` - класом `FilteredImageSource`. Цей клас перетворює уже готову продукцію, отриману від другого постачальника `producer`, використовуючи для перетворення об'єкт `filter` класу-фільтра `imageFilter` або його підкласу. Обидва об'єкта задаються в конструкторі

`FilteredImageSource(ImageProducer producer, ImageFilter filter)`

Все це здається дуже заплутаним, але схема застосування фільтрів завжди одна і та ж. Вона показана в лістингах 15.2—15.4.

15.3. Як виділити фрагмент зображення

В лістингі 15.2 виділяється фрагмент зображення і виводиться на екран у збільшенному вигляді. Крім того, нижче виводяться зображення, збільшені зі допомогою класів `ReplicateScaleFilter` і `AreaAveragingScaleFilter`.

Лістинг 15.2. Приклади маштабування зображення

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
class CropTest extends Frame{
private Image img, cropimg, replimg, averimg;
CropTest(String s){ super (s) ;
// 1. Створюємо зображення - об'єкт класу Image
img = getToolkit().getImage("javalogos52x88.gif");
// 2. Створюємо об'єкти-фільтри:
// а) виділяємо лівий верхній кут розміром 30x30
CropImageFilter crp = new CropImageFilter(0, 0, 30, 30);
// б) збільшуємо зображення в два рази простим методом
ReplicateScaleFilter rsf = new ReplicateScaleFilter(104, 176);
// в) збільшуємо зображення в два рази з усередненням
AreaAveragingScaleFilter asf = new AreaAveragingScaleFilter(104, 176);
// 3. Створюємо зміну зображення
cropimg = createImage(new FilteredImageSource(img.getSource(), crp));
replimg = createImage(new FilteredImageSource(img.getSource(), rsf));
averimg = createImage(new FilteredImageSource(img.getSource(), asf));
setSize(400, 350); setVisible(true); }
public void paint(Graphics g){ g.drawImage(img, 10, 40, this);
```

```

g.drawImage(cropimg, 150, 40, 100, 100, this);
g.drawImage(replimg, 10, 150, this);
g.drawImage(averimg, 150, 150, this);
}
public static void main(String[] args){
Frame f= new CropTest(" Маштабування");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
}

```

На рис. 15.3 зліва зверху показано початкове зображення, справа — збільшений фрагмент, внизу — зображення, збільшене двома способами.

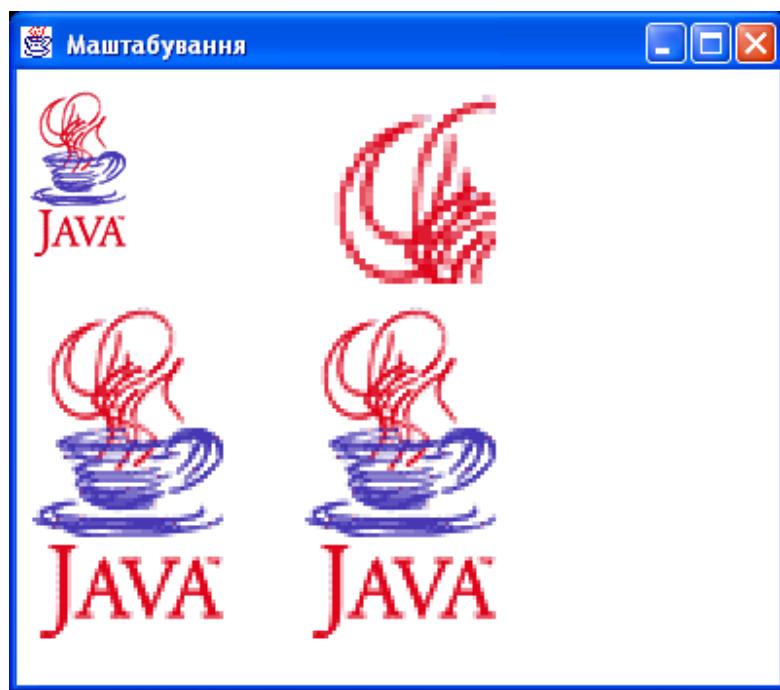


Рис. 15.3. Маштабування зображення

15.4. Як змінити колір зображення

В лістинзі 15.3 змінюються кольори кожного пікселя зображення. Це досягається просто зсувом `rgb » 1` вмісту пікселя на один біт вправо в методі `filterRGB()`. При цьому підсилюється червона складова кольору. Метод `filterRGB()` перевизначений в розширенні `colorFilter` класу `RGBImageFilter`.

Лістинг 15.3. Зміна кольору всіх пікселів

```

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
class RGBTest extends Frame{
private Image img, newimg;
RGBTest(String s){
super(s);
img = getToolkit().getImage("javalogo52x88.gif");

```

```

RGBImageFilter rgb = new ColorFilter();
newimg = createImage(new FilteredImageSource(img.getSource(), rgb));
setSize(400, 350);
setVisible(true);
public void paint(Graphics g){
g.drawImage(img, 10, 40, this);
g.drawImage(newimg, 150, 40, this); }
public static void main(String[] args){
Frame f= new RGBTest(" Зміна кольору");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}
class ColorFilter extends RGBImageFilter{ ColorFilter(){
canFilterIndexColorModel = true; }
public int filterRGB(int x, int y, int rgb){
return rgb >> 1;
}
}
}

```



15.5. Як переставити пікселі зображення

В лістингі 15.4 визначається перетворення пікселів зображення. Створюється новий фільтр - розширення `shiftFilter` класу `imageFilter`, здвигаючий зображення циклічно вправо на указане в конструкторі число пікселів. Все, що для цього потрібно, - це перевизначити метод `setPixels()`.

Лістинг 15.4. Циклічний зсув зображення

```

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
class ShiftImage extends Frame{ private Image img, newimg;
ShiftImage(String s){ super(s);
// 1. Одержано зображення із файла
img = Toolkit.getDefaultToolkit().getImage("javalogo52x88.gif");
// 2. Створюємо екземпляр фільтра
ImageFilter imf = new ShiftFilter(26);
// Зсув на 26 пікселів
// 3. Одержано нові пікселі за допомогою фільтра
ImageProducer ip = new FilteredImageSource(img.getSource(), imf);
// 4. Створюємо нове зображення
newimg = createImage(ip);
setSize(300, 200);
}
}

```

```

setVisible(true) ;
public void paint(Graphics gr){
gr.drawImage(img, 20, 40, this);
gr.drawImage(newimg, 100, 40, this);
public static void main(String[] args){
Frame f= new ShiftImage(" Циклічний зсув зображення");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
});
}
}
// Клас-фільтр
class ShiftFilter extends ImageFilter{
private int sh;
// Зсув на sh пікселів вправо.
public ShiftFilter(int shift){ sh = shift; }
public void setPixels(int x, int y, int w, int h,
ColorModel m, byte[] pix, int off, int size){
for (int k = x; k < x+w; k++){
if (k+sh <= w)
consumer.setPixels(k, y, 1, h, m, pix, off+sh+k, size);
else
consumer.setPixels(k, y, 1, h, m, pix, off+sh+k-w, size);
}
}
}
}

```

Як видно із листинга 15.4, перевизначення методу `setPixels()` заключається в тому, щоб змінити аргументи цього методу, переставивши, тим самим, пікселі зображення, і передати їх споживаю `consumer` — полю класу `imageFilter` методом `setPixels()` споживача. На рис. 15.4 показано результат виконання цієї програми. Інша модель обробки зображення введена в **Java 2D**. Вона названа моделлю **прямого доступу** (*immediate mode model*).

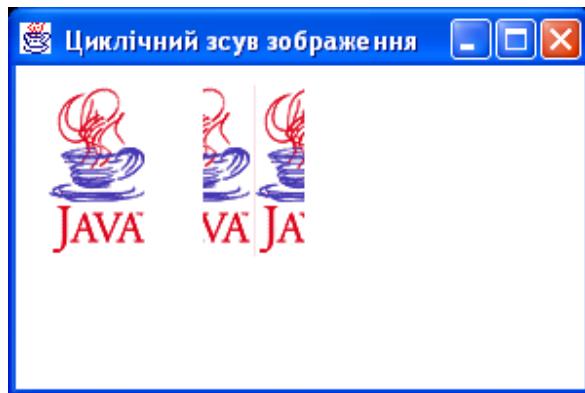


Рис. 15.4. Перестановка пікселів зображення

15.6. Модель обробки прямим доступом

Подібно до того, як замість класу `Graphics` система **Java 2D** використовує його розширення `Graphics2D`, описане в уроці 9, замість класу `Image` в **Java 2D** використовується його розширення — клас `BufferedImage`. В конструкторі цього класу

`BufferedImage(int width, int height, int imageType)`

задаються розміри зображення і спосіб зберігання точок — одна із констант:

TYPE_INT_RGB	TYPE_BYTE_GRAY	TYPE_3BYTE_BRG	TYPE USHORT_GRAY
TYPE_INT_ARGB	TYPE_BYTE_BINARY	TYPE_4BYTE_ABRG	TYPE USHORT_565_RGB
TYPE_INT_ARGB_PRE	TYPE_BYTE_INDEXED	TYPE_4BYTE_ABRG_PRE	TYPE USHORT_555_RGB
TYPE_INT_BRG			

Як бачите, кожний піксель може займати 4 байти — [INT, 4BYTE](#), або 2 байта — [USHORT](#), або 1 байт — [BYTE](#). Може використовуватися кольорова модель [RGB](#), або додана альфа-складова — [ARGB](#), або задани інший порядок розташування кольорових складових — [BRG](#), або задані градації сірого кольору — [GRAY](#). Кожна складова кольору може займати один байт, 5 бітів або 6 бітів. Екземпляри класу [BufferedImage](#) рідко створюються конструкторами. Для їх створення частіше звертаються до методів [createImage\(\)](#) класу [Component](#) з простим приведенням типу:

```
BufferedImage bi = (BufferedImage)createImage(width, height)
```

При цьому екземпляр [bi](#) отримує характеристики компонента: колір фону і колір рисування, спосіб зберігання точок. Розташування точок в зображенні регулюється класом [Raster](#) або його підкласом [WritableRaster](#). Ці класи задають систему координат зображення, представляють доступ до окремих пікселів методами [getPixel\(\)](#), дозволяють виділяти фрагменти зображення методами [getPixels\(\)](#). Клас [WritableRaster](#) додатково дозволяє змінювати окремі пікселі методами [getPixel\(\)](#) або цілі фрагменти зображення методами [setPixels\(\)](#) і [setRect\(\)](#). Початок системи координат зображення — лівий верхній кут — має координати ([minX, minY](#)), не обовязково рівні нулю.

При створенні екземпляра класу [BufferedImage](#) автоматично формується звязанный з ним екземпляр класу [WritableRaster](#). Точки зображення зберігаються в скритому буфері, вміщаючи одновимірний або двовимірний масив точок. Вся робота з буфером здійснюється методами одного із класів [DataBufferByte](#), [DataBufferInt](#), [DataBufferShort](#), [DataBufferUSHORT](#) в залежності від довжини даних. Загальні властивості цих класів зібрані в їх абстрактному суперкласі [DataBuffer](#). В ньому визначені типи даних, що зберігаються в буфері: [TYPE_BYTE](#), [TYPE_SHORT](#), [TYPE_INT](#), [TYPE_UNDEFINED](#).

Методи класу [DataBuffer](#) дозволяють прямий доступ до даних буфера, але зручніше і безпечніше звертатися до них методами класів [Raster](#) і [WritableRaster](#). При створенні екземпляра класу [Raster](#) або класу [WritableRaster](#) створюється екземпляр відповідного підкласу класу [DataBuffer](#). Щоб не враховувати спосіб зберігання точок зображення, [Raster](#) може звертатися не до буфера [DataBuffer](#), а до підкласів абстрактного класу [SampleModel](#), що розглядає не окремі байти буфера, а складові ([samples](#)) кольору. В моделі [RGB](#) — це червона, зелена і синя складові. В пакеті [java.awt.image](#) єсть п'ять підкласів класу [SampleModel](#):

- [ComponentSampleModel](#) — кожна складова кольору зберігається в окремому елементі масива [DataBuffer](#);
- [BandedSampleModel](#) — дані зберігаються по складовим, складов одного кольору зберігаються в одному масиві, а [DataBuffer](#) містить двовимірний масив: по масиву для кожної складової; даний клас розширяє клас [ComponentSampleModel](#);
- [PixelInterleavedSampleModel](#) — всі складові кольору одного пікселя зберігаються в сусідніх елементах єдиного масива [DataBuffer](#); даний клас розширяє клас [ComponentSampleModel](#);
- [MultiPixelPackedSampleModel](#) — колір кожного пікселя містить тільки одну складову, яка може бути упакована в один елемент масива [DataBuffer](#);
- [SinglePixelPackedSampleModel](#) — всі складові кольору кожного пікселя зберігаються в одному елементі масиву [DataBuffer](#).

На рис. 15.5 представлена ієархія класів [Java 2D](#), реалізуюча модель прямого доступу. Отже, [Java 2D](#) створює складну і розгалужену трьохшарову систему [DataBuffer](#) — [SampleModel](#) — [Raster](#) управління даними зображення [BufferedImage](#). Ви можете маніпулювати точками зображення, використовуючи їх координати в методах класів [Raster](#) або спуститися на рівень нижче і звертатися до складових кольору пікселя методами класів [SampleModel](#). Якщо ж вам треба працювати з окремими байтами, скористуйтесь класами [DataBuffer](#). Застосовувати цю систему приходиться рідко, тільки при створенні свого способу перетворення зображення. Стандартні ж перетворення виконуються дуже просто.

15.7. Перетворення зображення в Java 2D

Перетворення зображення `source`, що зберігається в обєкті класу `BufferedImage`, в нове зображення `destination` виконується методом `filter(BufferedImage source, BufferedImage destination)` описаним в інтерфейсі `BuffredImageOp`. Указаний метод повертає посилку на новий, змінений об'єкт `destination` класу `BuffredImage`, що дозволяє задати ланцюжок послідовних перетворень. Можна перетворювати тільки координатну систему зображення методом `filter(Raster source, WritableRaster destination)` повертаючим посилку на змінений об'єкт класу `WritableRaster`. Даний метод описано в інтерфейсі `RasterOp`. Спосіб перетворення визначається класом, реалізуючим ці інтерфейси, а параметри перетворення задаються в конструкторі класу. В пакеті `java.awt.image` є шість класів, реалізуючих інтерфейси `BuffredImageOp` і `RasterOp`:

- `AffineTransformOp` — виконує афінне перетворення зображення: зсув, поворот, відображення, стискання або розтягування по вісям;
- `RescaleOp` — змінює інтенсивність зображення;
- `LookupOp` — змінює окрім складові кольору зображення;
- `BandCombineOp` — мінює складові кольору в `Raster`;
- `ColorConvertOp` — змінює кольорову модель зображення;
- `ConvolveOp` — виконує згортання, що дозволяє змінити контраст і/або яскравість зображення, створити ефект "розмитості" і інші ефекти.

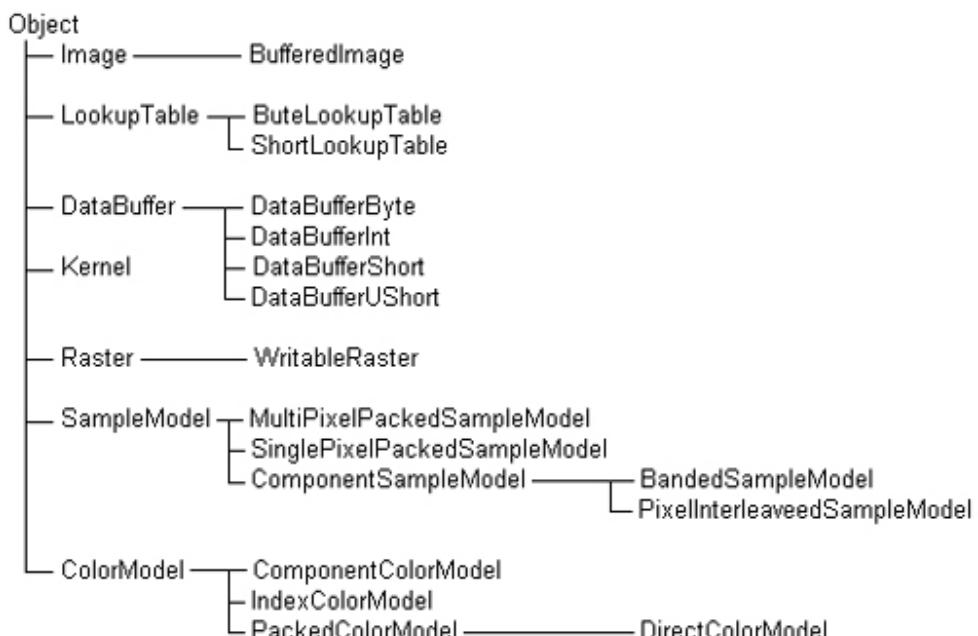


Рис. 15.5. Класи, реалізуючі модель прямого доступу

Розглянемо, як можна застосувати ці класи для перетворення зображення.

15.8. Афінне перетворення зображення

Клас `AffineTransform` і його використання детально розібрані в уроці 9, тут ми тільки застосуємо його для перетворення зображення. В конструкторі класу `AffineTransformOp` указується попередньо створене афінне перетворення `at` і спосіб інтерполяції `interp` і/або правила візуалізації `hints`:

```

AffineTransformOp(AffineTransform at, int interp);
AffineTransformOp(AffineTransform at, RenderingHints hints);
  
```

Спосіб інтерполяції — це одна із двох констант: `TYPE_NEAREST_NEIGHBOR` (по замовчуванню в

другому конструкторі) або `TYPE_BILINEAR`. Після створення об'єкту класу `AffineTransformOp` застосовується метод `filter()`. При цьому зображення перетворюється всередині нової області типу `BufferedImage`, як показано на рис. 15.6, справа. Сама область виділена чорним кольором. Другий спосіб афінного перетворення зображення — застосувати метод `drawImage(BufferedImage img, BufferedImageOp op, int x, int y)` класу `Graphics2D`. При цьому перетворюється вся область `img`, як продемонстровано на рис. 15.6, посередині. В лістингі 15.5 показано, як задаються перетворення, представлени на рис. 15.6. Зверніть увагу на особливості роботи з `BufferedImage`. Треба створити графічний контекст зображення і вивести в нього зображення. Ці дії здаються звичними, але зручні для подвійної буферизації, яка зараз стала стандартом перерисовування зображень, а в бібліотеці `Swing` виконується автоматично.

Лістинг 15.5. Афінне перетворення зображення

```

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.awt.event.*;
public class AffOp extends Frame{
private BufferedImage bi;
public AffOp(String s){ super (s) ;
// Завантажуємо зображення
Image img = getToolkit().getImage("javalogo52x88.gif");
// В цьому блоці організовано очікування завантаження
try{
MediaTracker mt = new MediaTracker(this);
mt.addImage(img, 0);
mt.waitForID(0);
// Чекаємо закінчення завантаження
}
catch(Exception e){}
// Розміри створюваної області bi співпадають з розмірами зображення img
bi = new BufferedImage(img.getWidth(this), img.getHeight(this),
BufferedImage.TYPE_INT_RGB);
// Створюємо графічний контекст big зображення bi
Graphics2D big = bi.createGraphics();
// Виводимо зображення img в графічний контекст
big.drawImage(img, 0, 0, this);
}
public void paint(Graphics g){
Graphics2D g2 = (Graphics2D)g;
int w = getSize().width;
int h = getSize().height;
int bw = bi.getWidth(this);
int bh = bi.getHeight(this);
// Створюємо афінне перетворення
AffineTransform at = new AffineTransform();
at.rotate(Math.PI/4); // Задаємо поворот на 45 градусів
// по годинниковій стрілці навколо лівого верхнього кута.
// Потім зсуваємо зображення вправо на величину bw
at.concatenate(new AffineTransform(1, 0, 0, 1, bw, 0));
// Визначаємо область зберігання bimg перетвореного
// зображення. Її розмір вдвічі більший попереднього
BufferedImage bimg = new BufferedImage(2*bw, 2*bh, BufferedImage.TYPE_INT_RGB);
// Створюємо об'єкт biop, що містить перетворення at
BufferedImageOp biop = new AffineTransformOp(at,
AffineTransformOp.TYPE_NEAREST_NEIGHBOR);
// Перетворюємо зображення, результат заносимо в bimg
biop.filter(bi, bimg);
// Виводимо початкове зображення.
g2.drawImage(bi, null, 10, 30);
}

```

```

// Виводимо змінене перетворенням віор область bi
g2.drawImage(bi, biop, w/4+3, 30);
// Виводимо перетворене всередині області bimg зображення
g2.drawImage(bimg, null, w/2+3, 30); }
public static void main(String[] args){
Frame f = new AffOp(" Афінне перетворення");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent e){
System.exit(0);
}
});
f.setSize(400, 200);
f.setVisible(true) ;
}
}

```

На рис. 15.6 показано початкове зображення, перетворена область і перетворене всередині області зображення.



Рис. 15.6. Афінне перетворення зображення

15.9. Зміна інтенсивності зображення

Зміна інтенсивності зображення виражається математично множенням кожної складової кольору на число `factor` і додаванням до результату множення числа `offset`. Результат приводиться до діапазону значень складової. Після цього інтенсивність кожної складової кольору лінійно змінюється в одному і тому ж масштабі. Числа `factor` і `offset` сталі для кожного пікселя і задаються в конструкторі класу разом з правилами візуалізації `hints`:

`RescaleOp(float factor, float^offset, RenderingHints hints)`

Після цього залишається застосувати метод `filter()`.

На рис. 15.7 інтенсивність кожного кольору зменшена вдвічі, в результаті білий фон став сірим, а кольори - темнішими. Потім інтенсивність збільшена на 70 одиниць. В лістингі 15.6 приведена программа, що виконує це перетворення.

Лістинг 15.6. Зміна інтенсивності зображення

```

import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
public class Rescale extends Frame{
private BufferedImage bi;
public Rescale(String s){

```

```

super (s) ;
Image img = getToolkit().getImage("javalogo52x88.gif");
try{
MediaTracker mt = new MediaTracker(this);
mt.addImage(img, 0);
mt.waitForID(0); }
catch(Exception e){}
bi = new BufferedImage(img.getWidth(this), img.getHeight(this),
BufferedImage.TYPE_INT_RGB);
Graphics2D big = bi.createGraphics();
big.drawImage(img, 0, 0, this);
}
public void paint(Graphics g){
Graphics2D g2 = (Graphics2D)g;
int w = getSize().width;
int bw = bi.getWidth(this);
int bh = bi.getHeight(this);
BufferedImage bimg = new BufferedImage(bw, bh, BufferedImage.TYPE_INT_RGB);
//_____ Початок визначення перетворення _____
RescaleOp rop = new RescaleOp(0.5f, 70.0f, null);
rop.filter(bi, bimg);
//_____ Кінець визначення перетворення _____
g2.drawImage(bi, null, 10, 30);
g2.drawImage(bimg, null, w/2+3, 30);
}
public static void main(String[] args){
Frame f = new Rescale("Зміна інтенсивності");
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent e) {
System.exit(0);
}
});
f.setSize(300, 200);
f.setVisible(true);
}
}

```

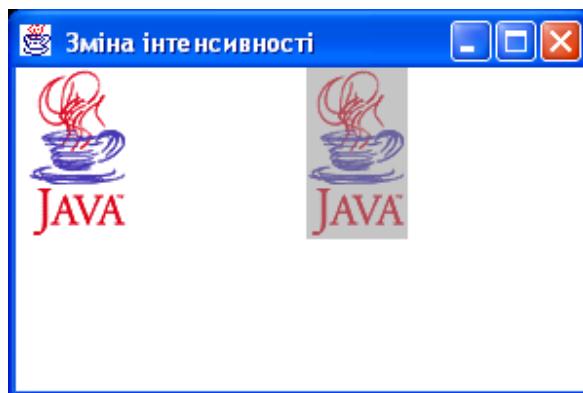


Рис. 15.7. Зміна інтенсивності зображення

15.10. Зміна складових кольору

Щоб змінити окремі складові кольору, треба перш за все продивитися тип зберігання елементів в `BufferedImage`, по замовчуванню це `TYPE_INT_RGB`. Тут три складові — червона, зелена і синя. Кожна складова кольору займає один байт, всі вони зберігаються в одному числі типу `int`. Потім треба скласти

таблицю нових значень складових. В лістингі 15.7 це двовимірний масив `samples`. Потім заповнюємо даний масив потрібними значеннями складових кожного кольору. В лістингі 15.7 задається яскраво-червоний колір рисування і білий колір фону. По отриманій таблиці створюємо екземпляр класу `ByteLookupTable`, який зв'яже цю таблицю з буфером даних. Цей екземпляр використовуємо для створення обєкта класу `LookupOp`. Нарешті, застосовуємо метод `filter()` цього класу.

В лістингі 15.7 приведений тільки фрагмент програми. Для одержання повної програми його треба вставити в лістинг 15.6 замість виділеного в ньому фрагмента. Логотип `Java` буде нарисований яскраво-червоним кольором.

Лістинг 15.7. Зміна складових кольору

```
//———— Вставити в лістинг 15.6 —————

byte samples[][] = new byte[3][256];
for (int j = 0; j < 255; j++){
    samples[0][j] = (byte) (255); // Червона складова
    samples[1][j] = (byte) (0); // Зелена складова
    samples[2][j] = (byte) (0); // Синя складова
}
samples[0][255] = (byte) (255); // Колір фону – білий
samples[1][255] = (byte) (255) ;
samples [2] [255] = (byte) (255) ;
ByteLookupTable blut=new ByteLookupTable(0, samples);
LookupOp lop = new LookupOp(blut, null);
lop.filter(bi, bimg);

//———— Кінець вставки —————
```



15.11. Створення різних ефектів

Операція згортання (`convolution`) задає значення кольору точки в залежності від кольорів сусідніх точок наступним способом. Нехай точка з координатами (x, y) має колір, вираженим числом $A(x, y)$. Складаємо масив із дев'яти дійсних чисел $w(0), w(1), \dots, w(8)$. Тоді нове значення кольору точки з координатами (x, y) буде рівним:

$$w(0)*A(x-l, y-l)+w(1)*A(x, y-l)+w(2)*A(x+l, y-l)+w(3)*A(x-l, y)+w(4)*A(x, y)+w(5)*A(x+l, y)+w(6)*A(x-l, y+l)+w(7)*A(x, y+l)+w(8)*A(x+l, y+1)$$

Задаючи різні значення ваговим коефіцієнтам $w(i)$, будемо отримувати різні ефекти, підсилюючи чи зменшуючи вплив сусідніх точок. Якщо сума всіх дев'яти чисел $w(i)$ рівна $1.0f$, то інтенсивність кольору залишиться попередня. Якщо при цьому всі ваги рівні між собою, тобто рівні $0.1111111f$, то одержимо ефект размитості, тумана, димки. Якщо вага $w(4)$ значно більше решти при загальній сумі їх $1.0f$, то зростає контрастність, виникає ефект графіки, штрихового рисунка.

Можно згорнути не тільки сусідні точки, але і наступні ряди точок, взявши масив вагових коєфіцієнтів із 15 елементов (3x5, 5x3), 25 елементів (5x5) і більше. В **Java 2D** згортання робиться так. Спочатку визначаємо масив ваг, наприклад:

```
float[] w = {0, -1, 0, -1, 5, -1, 0, -1, 0};
```

Потім створюємо екземпляр класу **Kernel** — ядра згортання:

```
Kernel kern = new Kernel(3, 3, w);
```

Потім обєкт класу **ConvolveOp** з цим ядром:

```
ConvolveOp conv = new ConvolveOp(kern);
```

Все готово, застосовуємо метод **filter()**: `conv.filter(bi, bimg);`

В лістинзі 15.8 записані дії, необхідні для створення ефекта "размитості".

Лістинг 15.8. Створення різних ефектів

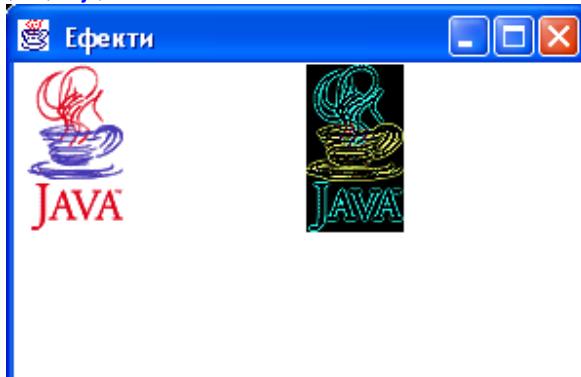
```
//———— Вставити в лістинг 15.6 —————
float[] wl = { 0.11111111f, 0.11111111f, 0.11111111f, 0.11111111f, 0.11111111f,
0.11111111f, 0.11111111f, 0.11111111f, 0.11111111f };
Kernel kern = new Kernel(3, 3, wl);
ConvolveOp cop = new ConvolveOp(kern, ConvolveOp.EDGE_NO_OP, null);
cop.filter(bi, bimg) ;

//———— Кінець вставки —————
```



На рис 15.8 представлена зліва направо початкове зображення і зображення, перетворені ваговими матрицями **w1**, **w2** і **w3**, де матриця **w1** показана в лістинзі 15.8, а матриці **w2** і **w3** виглядають так:

```
float[] w2 = { 0, -1, 0, -1, 4, -1, 0, -1, 0 };
```



```
float[] w3 = { -1, -1, -1,-1, 9, -1, -1, -1, -1 };
```



Рис. 15.8. Створення ефектів

15.12. Анімація

Єсть декілька способів створити анімацію. Самий простий із них — записати заздалегідь всі необхідні кадри в графічні файли, завантажити їх в оперативну пам'ять у вигляді об'єктів класу `Image` або `BufferedImage` і виводити по черзі на екран. Це зроблено в лістингі 15.9. Заготовлено десять кадрів у файлах `run1.gif`, `run2.gif`, , `run10.gif`. Вони завантажуються в масив `img[]` і виводяться на екран циклічно 100 раз, із затримкою в 0,1 сек.

Лістинг 15.9. Проста анімація

```
import java.awt.*;
import java.awt.event.*;
class SimpleAnim extends Frame{
private Image[] img = new Image[10];
private int count;
SimpleAnim(String s){ super(s);
MediaTracker tr = new MediaTracker(this);
for (int k = 0; k < 10; k++){
img[k] = getToolkit().getImage("run"+ String.valueOf(k) +".gif");
tr.addImage(img[k], 0);
}
try{
tr.waitForAll(); // Чекаємо завантаження всіх зображень
}
catch(InterruptedException e){}
}
```

```

setSize(400, 300);
setVisible(true);
}
public void paint(Graphics g){
g.drawImage(img[count % 10], 0, 0, this);
}
// public void update(Graphics g){ paint(g); }
public void go(){ while(count < 100){
repaint(); // Виводимо наступний кадр
try{ // Затримка в 0.1 сек
Thread.sleep(100);
}catch(InterruptedException e){count++;}}
}
}
public static void main(String[] args){
SimpleAnim f = new SimpleAnim("Проста анімація");
f.go();
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0); }
}); }
}

```

Якщо кадри анімації повністю перерисовують вікно, то його очистка методом `clearRect()` не потрібна. Більше того, вона часто викликає неприємне мерехтіння із-за появи на митті білого фону. В такому випадку треба зробити наступне перевизначення:

```

public void update(Graphics g) {
paint(g);
}

```

В лістингі 15.9 це перевизначення зроблено як коментар.

Для "легких" компонентів справа складніша. Метод `repaint()` послідовно звертається до методів `repaint()` обмежуючих "легких" контейнерів, поки не зустрінеться "важкий" контейнер, частіше всього це екземпляр класу `Container`. В ньому викликається метод `update()`, очищаючий і перерисовуючий контейнер. Після цього йде звернення до методів `update()` обмежуючих всі "легкі" компоненти в контейнері. Звідси слідує, що для усунення мерехтіння "легких" компонентів необхідно перевизначити метод `update()` першого обмежуючого "важкого" контейнера, звертаючись в ньому до методів `super.update(g)` або `super.paint(g)`. Якщо кадри покривають тільки частину вікна, причому кожний раз нову, то очистка вікна необхідна, інакше старі кадри залишаться у вікні, зявиться "хвіст". Щоб усунути мерехтіння, використовують прийом, одержавший назву "подвійна буферизація" (`double buffering`).

15.13. Покращення зображення подвійною буферизацією

Суть подвійної буферизації в тому, що в оперативній пам'яті створюється буфер — об'єкт класу `Image` або `BufferedImage`, і викликається його графічний контекст, в якому формується зображення. Там же відбувається очистка буфера, яка теж не відображається на екрані. Тільки після виконання всіх дій готове зображення виводиться на екран. Все це відбувається в методі `update()`, а метод `paint()` тільки звертається до `update()`. Лістинги 15.10—15.11 пояснюють даний прийом.

Лістинг 15.10. Подвійна буферизація за допомогою класу `Image`

```

public void update(Graphics g){
int w = getSize().width, h = getSize().height;
// Створюємо зображення-буфер в оперативній пам'яті
Image offImg = createImage(w, h);
// Одержано його графічний контекст

```

```

Graphics offGr = offImg.getGraphics();
// Змінюємо поточний колір буфера на колір фону
offGr.setColor(backgroundColor);
// і заповнюємо ним вікно компонента, очищуючи буфер
offGr.fillRect(0, 0, w, h);
// Повертаємо поточний колір буфера
offGr.setColor(foreground);
// Для лістингу 15.9 виводимо в контекст зображення
offGr.drawImage(img[count % 10], 0, 0, this);
// Рисуємо в графічному контексті буфера
// (необов'язкова дія)
paint(offGr);
// Виводимо зображення-буфер на екран
// (можна перенести в метод paint())
g.drawImage(offImg, 0, 0, this); }
// Метод paint() необов'язковий
public void paint(Graphics g).update(g); }

```

Лістинг 15.11. Подвійна буферизація за допомогою класу BufferedImage

```

public void update(Graphics g){
Graphics2D g2 = (Graphics2D),g;
int w = getSize().width, h = getSize().height;
// Створюємо зображення-буфер в оперативній пам'яті
BufferedImage bi = (BufferedImage)createImage(w, h);
// Створюємо графічний контекст буфера
Graphics2D big = bi.createGraphics();
// Установлюємо колір фону
big.setColor(backgroundColor);
// Очищаємо буфер кольором фону
big.clearRect(0, 0, w, h);
// Відновлюємо поточний колір
big.setColor(foreground);
// Виводимо що-небудь в графічний контекст big
// ...
// Виводимо буфер на екран
g2.drawImage(bi, 0, 0, this);
}

```

Метод подвійної буферизації став фактично стандартом виведення змінних зображень, а в бібліотеці **Swing** він застосовується автоматично. Даний метод зручний і при перерисуванні окремих частин зображення. В цьому випадку в зображені-буфері рисується незмінна частина зображення, а в методі **paint()** — те, що змінюється при кожному перерисуванні.

В лістингі 15.12 показано другий спосіб анімації — кадри зображення рисуються безпосередньо в програмі, в методі **update()**, по заданому закону зміни зображення. В результаті червоний мячик плигає на фоні зображення.

Лістинг 15.12. Анімація рисуванням

```

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;
class DrawAnim extends Frame{
private Image img;
private int count;
DrawAnim(String s) {
super(s);

```

```

MediaTracker tr = new MediaTracker(this);
img = getToolkit().getImage("IMG_0106.jpg");
tr.addImage(img, 0);
try{
tr.waitForID(0) ;
}catch(InterruptedException e) {}
setSize(1200, 800);
setVisible(true);
}
public void update(Graphics g){
Graphics2D g2 = (Graphics2D)g;
int w = getSize().width, h = getSize().height;
BufferedImage bi = (BufferedImage)createImage(w, h) ;
Graphics2D big = bi.createGraphics();
// Заповнюємо фон зображенням img
big.drawImage(img, 0, 0, this);
// Установлюємо колір рисування
big.setColor(Color.red);
// Рисуємо в графічному контексті буфера круг,
// переміщаючийся по синусоїде
big.fill(new Arc2D.Double(4*count, 50+30*Math.sin(count),
50, 50, 0, 360, Arc2D.OPEN));
// Змінюємо колір рисування
big.setColor(getForeground());
// Рисуємо горизонтальну пряму
big.draw(new Line2D.Double(0, 125, w, 125));
// Виводимо зображення-буфер на екран
g2.drawImage(bi, 0, 0, this); }
public void go(){
while(count < 1000){
repaint();
try{
Thread.sleep(10);
}catch(InterruptedException e){}
count++;
}
}
public static void main(String[] args){
DrawAnim f = new DrawAnim("Анімація");
f.go();
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){
System.exit(0);
}
});
}
}

```

Ефект мерехтіння, переливи кольору, затемнення і інші ефекти, отримані заміною окремих пікселів зображення, зручно створювати за допомогою класу [MemoryImageSource](#). Метод [newPixels\(\)](#) цього класу викликають негайне перерисування зображення навіть без звернення до методу [repaint\(\)](#), якщо перед цим виконаний метод [setAnimated\(true\)](#). Частіше всього застосовуються два методи:

- [newPixels\(int x, int y, int width, int height\)](#) — отримувачу посилається указаний аргументами прямокутний фрагмент зображення;
- [newPixels\(\)](#) — отримувачу посилається все зображення.

В лістингі 15.13 показано застосування цього способу. Квадрат, виведений на екран, переливається різними кольорами.

Лістинг 15.13. Анімація за допомогою `MemoryImageSource`

```

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
class InMemory extends Frame{
private int w = 100, h = 100, count;
private int[] pix = new int[w * h];
private Image img;
MemoryImageSource mis;
InMemory(String s){ super(s);
int i = 0;
for(int y = 0; y < h; y++){
int red = 255 * y / (h - 1);
for(int x = 0; x < w; x++){
int green = 255 * x / (w - 1);
pix[i++] = (255 << 24) | (red << 16) | (green << 8) | 128;
}
}
mis = new MemoryImageSource(w, h, pix, 0, w);
// Задаємо можливість анімації
mis.setAnimated(true);
img = createImage(mis);
setSize(350, 300);
setVisible(true);
}
public void paint(Graphics gr){
gr.drawImage(img, 10, 30, this);
}
public void update(Graphics g){ paint(g); }
public void go(){
while (count < 100){
int i = 0;
// Змінюємо масив пікселів по деякому закону
for(int y = 0; y < h; y++)
for (int x = 0; x < w; x++)
pix[i++] = (255 << 24) | (255 + 8 * count << 16) | (8*count <<8) | 255 + 8 * count;
// Повідомляємо споживача про зміну
mis.newPixels();
try{
Thread.sleep(100);
} catch(InterruptedException e){}
count++;
}
}
public static void main(String[] args){
InMemory f= new InMemory(" Зображення в пам'яті");
f.go();
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ev){ System.exit(0); }
});
}
}

```

Ось і всі засоби для анімації, решта — їх вміле застосування. Комбінуючи розглянуті способи, можна добитися цікавих ефектів. В документації SUN J2SDK, в каталогах `demo\applets` і `demo\jfc\Java2D\src`, наведено багато прикладів аплетів і додатків з анімацією.

15.14. Звук

Як було указано в попередньому уроці, в аплетах реалізується інтерфейс `AudioClip`. Екземпляр об'єкта, реалізуючого цей інтерфейс можна отримати методом `getAudioClip()`, котрий, крім того, завантажує звуковий файл, а потім користуватися методами `play()`, `loop()` і `stop()` цього інтерфейса для програвання музики. Для застосування даного ж прийому в додатках в клас `Applet` введено статичний метод `newAudioClip(URL address)`, завантажуючий звуковий файл, що знаходиться по адресу `address`, і повертаючий об'єкт, реалізуючий інтерфейс `AudioClip`. Його можна використовувати для програвання звуку в додатку, якщо звичайно звукова система комп'ютера уже настроєна.

В лістингі 15.14 приведено найпростіший консольний додаток, нескінчено програючий звуковий файл `doom.mid`, що знаходиться в поточному каталозі. Для завершення додатку треба застосувати засоби операційної системи, наприклад, комбінацію клавіш `<Ctrl>+<C>`.

Лістинг 15.14. Найпростіший аудіододаток

```
import java.applet.* ;
import java.net.*;
class SimpleAudio{
SimpleAudio () {
try{
AudioClip ac = Applet.newAudioClip(new URL("file:doom.mid"));
ac.loop();
}catch(Exception e){}
}
public static void main(String[] args){
new SimpleAudio();
}
}
```

Таким способом можна програвати звукові файли типів `AU`, `WAVE`, `AIFF`, `MIDI` без стискання. В склад віртуальної машини `Java`, що входить в `SUN J2SDK` починаючи з версії 1.3, включено пристрій, програваючий звук, записаний в одному із форматів `AU`, `WAVE`, `AIFF`, `MIDI`, перетворюючий, мікшуючий і записуючий звук в тих же форматах. Для роботи з цим пристроєм створені класи, зібрани в пакети `javax.sound.sampled`, `javax.sound.midi`, `javax.sound.sampled.spi` і `javax.sound.midi.spi`. Перечислений набір класів для роботи із звуком отримав назву `Java Sound API`.

15.15. Програвання звуку в Java 2

Програвач звуку, вбудований в `JVM`, розрахований на два способи запису звуку: моно і стерео оцифровку (`digital audio`) з частотою дискретизації (`sample rate`) від 8 000 до 48 000 Гц і аппроксимацією (`quantization`) 8 і 16 бітів, і `MIDI`-послідовності (`sequences`) типу 0 і 1. Оцифрований звук повинен зберігатися в файлах типу `AU`, `WAVE` і `AIFF`. Його можна програвати двома способами. Перший спосіб описаний в інтерфейсі `clip`. Він розрахований на відтворення невеликих файлів або неодноразового програвання файла і заключається в тому, що весь файл цілком завантажується в оперативну пам'ять, а потім проиграється. Другий спосіб описаний в інтерфейсі `SourceDataLine`. Згідно цьому способу файл завантажується в оперативну пам'ять по частинах в буфер, розмір якого можна задати довільно. Перед завантаженням файла треба задати формат запису звуку в об'єкті класу `AudioFormat`. Конструктор цього класу:

```
AudioFormat(float sampleRate, int sampleSize, int channels, boolean signed, boolean bigEndian)
```

вимагає знання частоти дискретизації `sampleRate` (по замовчуванню 44 100 Гц), аппроксимації `sampleSize`, заданої в бітах (по замовчуванню 16), числа каналів `channels` (1 — моно, по замовчуванню 2 — стерео), запис чисел із знаком, `signed == true`, або без знака, і порядка розташування байтів в числі `bigEndian`. Такі дані звичайно невідомі, тому їх отримують із самого файла. Це відбувається в два кроки. На першому кроці отримуємо формат файла статичним методом `getAudioFileFormat()` класу `AudioSystem`, на другому — формат запису звуку методом `getFormat()` класу `AudioFileFormat`. Це описано в лістингі 15.15. Після того як формат запису визначено і занесено в об'єкт класу `AudioFormat`, в об'єкті класу `DataLine.info` збирається інформація про вхідну лінію (`line`) і способі програвання `clip` або `SourceDataLine`.

Далі треба перевірити, чи зможе проигравач обслуговувати лінію з таким форматом. Потім треба звязати лінію з програвачем статичним методом `getLine()` класу `AudioSystem`. Потім створюємо потік даних із файла — об'єкт класу `AudioInputStream`. Із цього потоку теж можна отримати об'єкт класу `AudioFormat` методом `getFormat()`. Даний варіант вибраний в лістингі 15.16. Відкриваємо створений потік методом `open()`. У-фф! Все готово, тепер можна почати програвання методом `start()`, завершити методом `stop()`, "перемотати" в початок методом `setFramePosition(0)` або `setMillisecondPosition(0)`.

Можна задати програвання `n` раз підряд методом `loop(n)` або нескінчене число раз методом `loop(Clip.LOOP_CONTINUOUSLY)`. Перед цим необхідно установити початкову `n` і кінцеву `m` позиції повторення методом `setLoopPoints(n, m)`. По закінченню програвання треба закрити лінію методом `close()`. Вся ця послідовність дій показана в лістингі 15.15.

Лістинг 15.15. Програвання аудіокліпа

```
import javax.sound.sampled.*;
import java.io.*;
class PlayAudio{
PlayAudio(String s){
play(s);
}
public void play(String file){
Clip line = null;
try{
// Створюємо об'єкт, представляючий файл
File f = new File (file);
// Отримуємо інформацію про спосіб запису файла
AudioFileFormat aff = AudioSystem.getAudioFileFormat(f);
// Отримуємо інформацію про спосіб запису звуку
AudioFormat af = aff.getFormat();
// Збираємо всю інформацію вмісті, додаючи дані про клас
DataLine.Info info = new DataLine.Info(Clip.class, af) ;
// Перевіряємо, чи можна програвати такий формат
if (!AudioSystem.isLineSupported(info)){
System.err.println("Line is not supported");
System.exit(0);
}
// Отримуємо лінію зв'язку з файлом
line = (Clip)AudioSystem.getLine(info);
// Створюємо потік байтів із файла
AudioInputStream ais = AudioSystem.getAudioInputStream(f);
// Відкриваємо лінію
line.open(ais);
}catch(Exception e){
System.err.println(e);
}
// Начинаємо програвання
line.start();
// Тут треба зробити затримку до закінчення програвання або зупинити його наступним
// методом:
try{ // Затримка в 0.1 сек
Thread.sleep(60000);
}catch(InterruptedException e){ }
line.stop();
// По закінченні програвання закриваємо лінію
line.close();
}
public static void main(String[] args){
if (args.length != 1)
System.out.println("Usage: Java PlayAudio filename");
```

```

new PlayAudio(args[0]);
}
}
}

```

Як бачите, методи Java Sound API виконують елементарні дії, які треба повторяти із програми в программу. Як говорять, це методи "низького рівня" (*low level*). Другий спосіб, використовуючий методи інтерфейса `SourceDataLine`, вимагає попереднього створення буфера довільного розміру.

Лістинг 15.16. Програвання аудіофайла

```

import javax.sound.sampled.*;
import java.io.*;
class PlayAudioLine{
PlayAudioLine(String s){
play(s);
}
public void play(String file){
SourceDataLine line = null;
AudioInputStream ais = null;
byte[] b = new byte[2048]; // Буфер даних
try{
File f = new File(file);
// Створюємо вхідний потік байтів із файла f
ais = AudioSystem.getAudioInputStream(f);
// Дістаємо із потоку інформацію про спосіб запису звуку
AudioFormat af = ais.getFormat () ;
// Заносимо цю інформацію в обєкт info
DataLine.Info info = new DataLine.Info(SourceDataLine.class, af);
// Перевіряємо, чи підходить такий спосіб запису звуку
if (!AudioSystem.isLineSupported(info)){
System.err.println("Line is not supported");
System.exit(0);
}
// Отримуємо вхідну лінію
line = (SourceDataLine)AudioSystem.getLine(info);
// Відкриваємо лінію
line.open(af);
try{
Thread.sleep(1000);
}catch(InterruptedException e){}
// Починаємо програвання
int num = 0 ;
line.start(); // Чекаємо появи даних в буфері
// Роз за разом заповняємо буфер
while(( num = ais.read(b)) != -1)
line.write(b, 0, num);
// "Зливаемо" буфер, програючи залишок файла
line.drain();
// Закриваємо потік
ais.close();
} catch (Exception e) {
System.err.println (e);
}
// Зупиняємо програвання
line.stop();
// Закриваємо лінію
line.close();
}
public static void main(String[] args){

```

```

String s = "bark.aif";
if (args.length > 0) s = args[0];
new PlayAudioLine(s) ;
}
}

```

Управляти програванням файла можна за допомогою подій. Подія класу [LineEvent](#) настає при відкриті, [OPEN](#), і закритті, [CLOSE](#), потоку, при початку, [START](#), і закінченні, [STOP](#), програвання. Характер події відмічається указаними константами. Відповідний інтерфейс [LineListener](#) описує тільки один метод [update\(\)](#).

В [MIDI](#)-файлах зберігається [послідовність \(sequence\)](#) команд для [секвенсора \(sequencer\)](#) – пристрою для запису, програвання і редагування [MIDI](#)-послідовності, яким може бути фізичний пристрій або программа. Послідовність складається із декількох [дорожок \(tracks\)](#), на яких записані [MIDI-події \(events\)](#). Кожна дорожка завантажується в своєму [каналі \(channel\)](#). Звичайно дорожка містить звучання одного музичного інструмента або запис голосу одного виконавця або запис декількох виконавців, мікшовану [синтезатором \(synthesizer\)](#). Для програвання [MIDI](#)-послідовності в найпростішому випадку треба створити екземпляр секвенсора, відкрити його і направити в нього послідовність, отриману із файла, як показано в лістингі 15.17. Після цього треба почати програвання методом [start\(\)](#). Закінчити програвання можна методом [stop\(\)](#), "перемотати" послідовність на початок запису або на указаний час програвання - методами [setMicrosecondPosition\(long mcs\)](#) або [setTickPosition\(long tick\)](#).

Лістинг 15.17. Програвання [MIDI](#)-послідовності

```

import javax.sound.midi.*;
import java.io.*;
class PlayMIDK{
PlayMIDK(String s) {
play(s);
}
public void play(String file){
try{
File f = new File(file);
// Отримуємо секвенсор по замовчуванню
Sequencer sequencer = MidiSystem.getSequencer();
// Перевіряємо, чи отриманий секвенсор
if (sequencer == null) {
System.err.println("Sequencer is not supported");
System.exit(0);
}
// Відкриваємо секвенсор
sequencer.open();
// Отримуємо MIDI-послідовність із файла
Sequence seq = MidiSystem.getSequence(f);
// Направляємо послідовність в секвенсор
sequencer.setSequence(seq);
// Починаємо програвання
sequencer.start();
// Тут треба зробити затримку на час програвання, а потім зупинити:
try{
Thread.sleep(67000);
}catch(InterruptedException e){}
sequencer.stop();
}catch(Exception e){
System.err.println(e);
}
}
public static void main(String[] args){
String s = "doom.mid";

```

```

if (args.length > 0) s = args[0];
new PlayMIDK(s);
}
}

```

15.16. Синтез і запис звуку в Java 2

Синтез звуку заключається в створенні MIDI-послідовності - обєкта класу `sequence` — яким-небудь способом: з мікрофона, лінійного входа, синтезатора, із файла, або просто створити в програмі, як це робиться в лістингі 15.18. Спочатку створюється пуста послідовність одним із двох конструкторів:

```

Sequence(float divisionType, int resolution)
Sequence(float divisionType, int resolution, int numTracks)

```

Перший аргумент `divisionType` визначає спосіб відрахунків моментів (`ticks`) MIDI-подій — це одна із констант:

- `PPQ (Pulses Per Quarter note)` — відрахунки заміряються в долях від тривалості звуку в чверть;
- `SMPTE_24, SMPTE_25, SMPTE_so, SMPTE_30DROP` (Society of Motion Picture and Television Engineers) — відрахунки в долях одного кадра, при указаному числі кадрів в секунду.

Другий аргумент `resolution` задає кількість відрахунків у вказану одиницю, наприклад,

```
Sequence seq = new Sequence()Sequence.PPQ, 10);
```

задає 10 відрахунків у звуці тривалістю в чверть. Третій аргумент `numTracks` визначає конструктор дорожок в MIDI-послідовності. Потім, якщо застосовувався перший конструктор, в послідовності створюється одна або декілька дорожок:

```
Track tr = seq.createTrack();
```

Якщо застосовувався другий конструктор, то треба отримати уже створені конструктором дорожки:

```
Track[] trs = seq.getTracks();
```

Потім дорожки заповнюються MIDI-подіями за допомогою MIDI-подій. Єсть декілька типів повідомлень для різних типів подій. Найчастіше зустрічаються події типу `shortMessage`, які створюються конструктором по замовчуванню і потім заповнюються методом `setMessage()`:

```

ShortMessage msg = new ShortMessage();
rasg.setMessage(ShortMessage.NOTEJDN, 60, 93);

```

Перший аргумент указує тип повідомлення: `NOTE_ON` - почати звучання, `NOTE_OFF` - зупинити звучання і т. д. Другий аргумент для типу `NOTE_ON` показує висоту звуку, в стандарті MIDI це числа від 0 до 127, 60 - нота "до" першої октави. Третій аргумент означає "швидкість" натискання клавіші MIDI-інструмента і по-різному розуміється різними пристроями. Далі створюється MIDI-подія:

```
MidiEvent me = new MidiEvent{msg, ticks};
```

Перший аргумент конструктора `msg` — це повідомлення, другий аргумент `ticks` - час наступу події (в нашому прикладі програвання ноти "до") в одиницях послідовності `seq` (в нашому прикладі в десятих долях чверті). Час відраховується від початку програвання послідовності. Нарешті, подія заноситься на дорожку:

```
tr.add(me);
```

Указані дії продовжуються, поки всі дорожки не будуть заповнені всіма подіями. В лістингі 15.18 це робиться в циклі, але звичайно MIDI-події створюються в методах обробки натискання клавіш на звичайній або спеціальній MIDI-клавіатурі. Ще один спосіб — вивести на екран зображення клавіатури і

створювати MIDI-події в методах обробки натискання кнопки миші на цій клавіатурі. Після створення послідовності її можна програвати, як в лістинзі 15.17, або записати у файл або вихідний потік. Для цього замість методу `start()` треба застосовувати метод `startRecording()`, який одночасно і програє послідовність, і готує її до запису, яку здійснюють статичні методи:

```
write(Sequence in, int type, File out)
write(Sequence in, int type, OutputStream out)
```

Другий аргумент `type` задає тип MIDI-файла, який краще всього визначити для заданної послідовності `seq` статичним методом `getMidiFileTypes(seq)`. Даний метод повертає масив можливих типів. Треба скористатися нульовим елементом масиву. Все це показано в лістинзі 15.18.

Лістинг 15.18. Створення MIDI-послідовності нот звукоряду

```
import javax.sound.midi.*;
import java.io.*;
class SynMIDI {
SynMIDI() {
play(synth());
}
public Sequence synth(){
Sequence seq = null;
try{
// Послідовність буде відраховувати по 10 MIDI-подій на звук довжиною в чверть
seq = new Sequence(Sequence.PPQ, 10);
// Створюємо в послідовності одну дорожку
Track tr = seq.createTrack();
for (int k = 0; k < 100; k++){
ShortMessage msg = new ShortMessage();
// Пробігаємо MIDI-ноти від номера 10 до 109
msg.setMessage(ShortMessage.NOTE_ON, 10+k, 93);
// Будемо програвати ноти через кожні 5 відрахунків
tr.add(new MidiEvent(msg, 5*k));
msg = null;
}
} catch (Exception e) {
System.err.println("From synth(): "+e);
System.exit(0);
}
return seq;
}
public void play (Sequence seq) {
try{
Sequencer sequencer = MidiSystem.getSequencer();
if (sequencer == null){
System.err.println("Sequencer is not supported");
System.exit(0);
}
sequencer.open();
sequencer.setSequence(seq);
sequencer.startRecording();
int[] type = MidiSystem.getMidiFileTypes(seq);
MidiSystem.write(seq, type[0], new File("gammas.mid"));
} catch(Exception e) {
System.err.println("From play(): " + e);
}
}
}
public static void main(String[] args){
new SynMIDI();
```

```
}  
}
```

На жаль, із-за обмалі часу ми не можемо торкнутися теми про роботу з синтезатором (*synthesizer*), мікшування звуку, роботи з декільками інструментами і інших можливостей *Java Sound API*. В документації *SUN J2SDK*, в каталозі [docs\guide\sound\prog_guide](#), єсть детальне керівництво програміста, а в каталозі [demo\sound\src](#) лежать вихідні тексти синтезатора, використовуваного в *Java Sound API*.

Лабораторна робота 14. Робота з зображенням і анімація.

Програмування у Java

Урок 16. Обробка виключчих ситуацій

- **Блоки перехвату виключення**
- **Частина заголовку метода throws**
- **Оператор throw**
- **Ієрархія класів-виключень**
- **Порядок обробки виключень**
- **Створення власних виключень**
- **Заключення**

16.1. Виключчні ситуації

Виключчні ситуації ([exceptions](#)) можуть виникнути під час виконання ([runtime](#)) програми, перервавши її звичайний хід. До них відноситься ділення на нуль, відсутність завантажуваного файла, відємний або війшовши за верхню межу індекс масива, переповнення виділеної пам'яті і маса інших неприємностей, які можуть трапитися в самий непідходячий момент. Звичайно, можна передбачити такі ситуації і застрахуватися від них як-небудь так:

```
if (something == wrong) {
    // Робимо аварійні дії
} else {
    // Звичайний хід дій
}
```

Але при цьому багато часу йде на перевірки, і програма перетворюється в набір цих перевірок. Подивіться будь-яку солідну програму, написану мовою [C](#) або [Pascal](#), і побачите, що вона на 2/3 складається із таких перевірок. В об'єктно-орієнтованих мовах програмування прийнято інший підхід. При виникненні виключчної ситуації виконуюча система створює об'єкт певного класу, відповідний виникнувшій ситуації, який містить дані про те, що, де і коли трапилося. Цей об'єкт передається на обробку програмі, в якій виникло виключення. Якщо програма не обробляє виключення, то об'єкт повертається обробчику по замовчуванню виконуючої системи. Обробчик поступає дуже просто: виводить на консоль повідомлення про виникнувше виключення і зупиняє виконання програми.

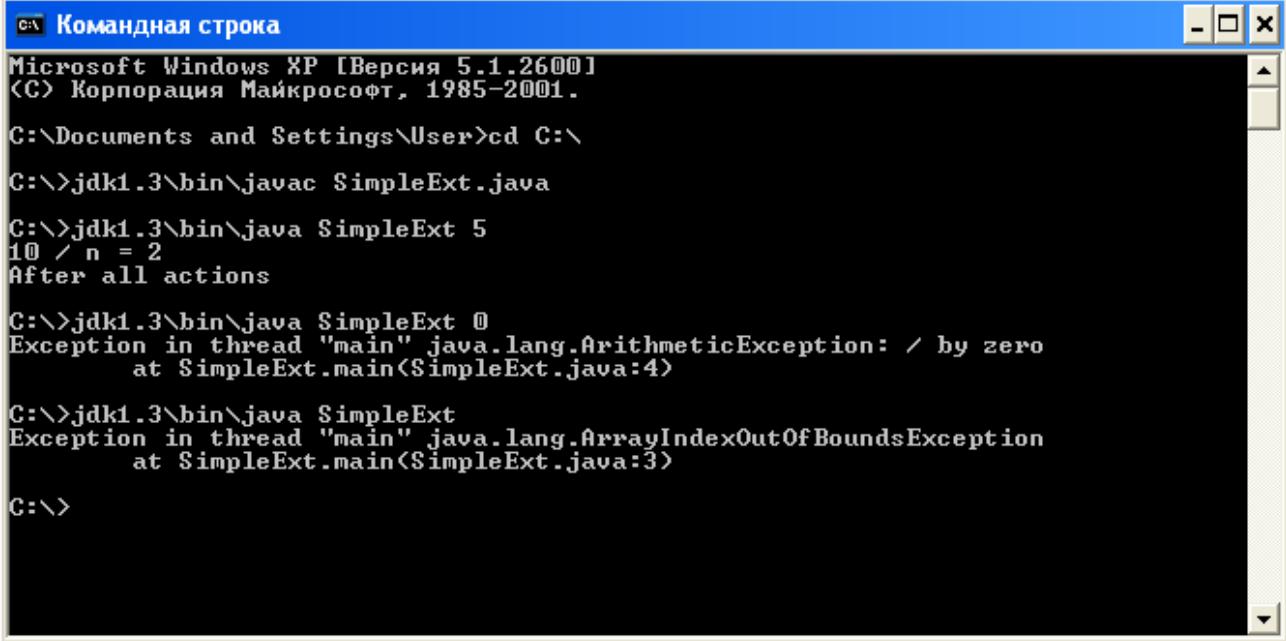
Наведемо приклад. В програмі лістинга 16.1 може виникнути ділення на нуль, якщо запустити її з аргументом 0. В програмі немає ніяких засобів обробки такої виключчної ситуації. Подивіться на рис. 16.1, які повідомлення виводить виконуюча система [Java](#).

Лістинг 16.1. Програма без обробки виключень

```
class SimpleExt {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        System.out.println("10 / n = " + (10 / n));
        System.out.println("After all actions");
    }
}
```

Програма [SimpleExt](#) запущена три рази. Перший раз аргумент [args\[0\]](#) рівний 5 і програма виводить результат: "10 / n = 2". Після цього появляється друге повідомлення: "After all actions". Другий раз аргумент рівний 0, і замість результата ми отримуємо повідомлення про те, що в підпроцесі "main" відбулося виключення класу [ArithmaticException](#) внаслідок ділення на нуль: "/ by zero". Далі уточнюється, що виключення виникло при виконанні метода [main](#) класу [SimpleExt](#), а в дужках указано, що дія, в результаті якої виникла виключча ситуація, записана в четвертому рядку файла [SimpleExt.java](#). Виконання програми зупиняється, заключне повідомлення не появляється. Третій раз програма запущена взагалі без аргумента. В масиві [args](#) немає елементів, його довжина рівна нулю, а ми намагаємося звернутися до елементу [args\[0\]](#). Виникає виключча ситуація класу [ArrayIndexOutOfBoundsException](#) внаслідок дії, записаної в третьому рядку файла [SimpleExt.java](#). Виконання програми зупиняється, звернення до методу

`println()` не відбувається.



```

Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\User>cd C:\

C:\>jdk1.3\bin\javac SimpleExt.java

C:\>jdk1.3\bin\java SimpleExt 5
10 / n = 2
After all actions

C:\>jdk1.3\bin\java SimpleExt 0
Exception in thread "main" java.lang.ArithmeticsException: / by zero
        at SimpleExt.main(SimpleExt.java:4)

C:\>jdk1.3\bin\java SimpleExt
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at SimpleExt.main(SimpleExt.java:3)

C:\>

```

Рис. 16.1. Повідомлення про виключні ситуації

16.2. Блоки перехвату виключення

Ми можемо перехватити і обробити виключення в програмі. При опису обробки застосовується бейсбольна термінологія. Говорять, що виконуюча система або програма "викидає" (`throws`) об'єкт-виключення. Цей об'єкт "пролітає" через всю програму, зявившись спочатку в тому методі, де відбулося виключення, а програма в одному або декількох місцях намагається (`try`) його "перехопити" (`catch`) і обробити. Обробку можна зробити повністю в одному місці, а можна обробити виключення в одному місці, викинути знову, перехопити в другому місці і обробляти далі.

Ми уже багато раз в цій книзі стикалися з необхідністю обробляти різні виключні ситуації, але не робили цього, тому що не хотіли відволікатися від основних конструкцій мови. Не вводьте це в звичку! Добре написані об'єктно-орієнтовані програми обовязково повинні обробляти всі виникаючі в них виключні ситуації. Для того щоб спробувати (`try`) перехватити (`catch`) об'єкт-виключення, треба весь код програми, в якому може виникнути виключна ситуація, охопити оператором `try {} catch() {}`. Кожний блок `catch(){}` перехоплює виключення тільки одного типу, того, котрий указано в його аргументі. Але можна написати декілька блоків `catch(){}` для перехоплення декількох типів виключень. Наприклад, ми знаємо, що в програмі лістинга 16.1 можуть виникнути виключення двох типів. Напишемо блоки їх обробки, як це зроблено в лістингі 16.2.

Лістинг 16.2. Програма з блоками обробки виключень

```

class SimpleExt1{
public static void main(String[] args){
try{
int n = Integer.parseInt(args[0]);
System.out.println("After parseInt()");
System.out.println(" 10 / n = " + (10 / n) );
System.out.println("After results output");
}catch(ArithmeticsException ae){
System.out.println("From Arithm.Exc. catch: "+ae);
}catch(ArrayIndexOutOfBoundsException arre){
}
}

```

```

System.out.println("From Array.Exc.catch: "+arre);
}finally{
System.out.println("From finally");
}
System.out.println("After all actions");
}
}

```

В програму лістинга 16.1 вставлено блок `try{}` і два блоки перехоплення `catch(){}` для кожного типу виключення. Обробка виключення тут заключається просто у виведенні повідомлення і вмісту обєкта-виключення, як воно представлено методом `toString()` відповідного класу-виключення. Після блоків перехоплення вставлено ще один, необов'язковий блок `finally()`. Він призначений для виконання дій, які треба виконати обов'язково, щоб там не трапилось. Все, що написано в цьому блоці, буде виконано і при виникненні виключення, і при звичайній ході програми, і навіть якщо вихід із блоку `try{}` здійснюється оператором `return`.

Якщо в операторі обробки виключень є блок `finally{}`, то блок `catch() {}` може бути відсутнім, тобто можна не перехоплювати виключення, але при його появі все-таки зробити якісь обов'язкові дії. Крім блоків перехоплення в лістингі 16.2 післяожної дії робиться трасуючий друк, щоб можна було прослідити за порядком виконання програми. Програма запущена три рази: з аргументом 5, з аргументом 0 і взагалі без аргумента. Результат показано на рис. 16.2.

```

С:\ Командная строка
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\User>cd C:\

C:\>jdk1.3\bin\javac SimpleExt1.java

C:\>jdk1.3\bin\java SimpleExt1 5
After parseInt()
10 / n = 2
After results output
From finally
After all actions

C:\>jdk1.3\bin\java SimpleExt1 0
After parseInt()
From Arithm.Exc. catch: java.lang.ArithmException: / by zero
From finally
After all actions

C:\>jdk1.3\bin\java SimpleExt1
From Array.Exc.catch: java.lang.ArrayIndexOutOfBoundsException
From finally
After all actions

C:\>_

```

Рис. 16.2. Повідомлення обробки виключень

Після першого запуску, при звичайній ході програми, виводяться всі дії. Після другого запуску, що веде до ділення на нуль, управління зараз же передається у відповідний блок `catch(ArithmException ae) {}`, потім виконується те, що написано в блоці `finally{}`. Після третього запуску управління після виконання методу `parseInt()` передається в другий блок `catch(ArrayIndexOutOfBoundsException arre){}`, потім в блок `finally{}`.

Зверніть увагу, що у всіх випадках — і при звичайній ході програми, і після цих обробок — виводиться повідомлення `"After all actions"`. Це свідчить про те, що виконання програми не зупиняється при виникненні виключної ситуації, як це було в програмі лістингу 16.1, а продовжується після обробки і виконання блоку `finally {}`. При запису блоків обробки виключень треба чітко уявляти собі, як буде передаватися управління у всіх випадках. Тому вивчіть уважно рис. 16.2. Цікаво, що пустий блок `catch (){}`,

в якому між фігурними дужками нічого немає, навіть пробілу, теж вважається обробкою виключення і приводить до того, що виконання програми не зупиниться. Саме так і "обробляли" виключення в попередніх уроках. Трохи вище було сказано, що викинуте виключення "пролітає" через всю програму. Що це означає? Змінimo програму лістингу 16.2, винісши ділення в окремий метод `f()`. Одержано лістинг 16.3.

Лістинг 16.3. Видалення виключень із методу

```
class SimpleExt2{
private static void f(int n){
System.out.println(" 10 / n = " + (10 / n));
}
public static void main(String[] args){
try{
int n = Integer.parseInt(args[0]);
System.out.println("After parseInt()");
f (n);
System.out.println("After results output");
}catch(ArithmetiсException ae){
System.out.println("From Arithm.Exc. catch: "+ae);
}catch(ArrayIndexOutOfBoundsException arre){
System.out.println("From Array.Exc. catch: "+arre);
}finally{
System.out.println("From finally");
}
System.out.println("After all actions");
}
}
```

Відкомпілювавши і запустивши програму лістингу 16.3, впевнимося, що виведення програми не змінилося, воно таке ж, як на рис. 16.2. Виключення, що виникло при діленні на нуль в методі `f()`, "пролетіло" через цей метод, "вилетіло" в метод `main()`, там перехоплено і оброблено.

16.3. Частина заголовка методу `throws`

Та обставина, що метод не обробляє виникаючого в ньому виключення, а викидає (`throws`) його, належить відмічати в заголовку методу службовим словом `throws` і указанням класу виключення:

```
private static void f(int n) throws ArithmetiсException{
System.out.println(" 10 / n = " + (10 / n));
}
```

Чому ж ми не зробили цього в лістингі 16.3? Справа в тому, що специфікація `JLS` ділить всі виключення на перевірючі (`checked`), тобто, котрі перевіряє компілятор, і неперевірючі (`unchecked`). При перевірці компілятор помічає необроблені в методах і конструкторах виключення і вважає за помилку відсутність в заголовку таких методів і конструкторів з поміткою `throws`. Саме для уникнення подібних помилок ми в попередніх уроках вставляли в лістинги блоки обробки виключень. Так от, виключення класу `RuntimeException` і його підкласів, одним із яких являється `ArithmetiсException`, неперевірючі, для них помітка `throws` необовязкова. Ще одно велике сімейство неперевірючих виключень складає клас `Error` і його розширення. Чому компілятор не перевіряє ці типи виключень? Причина в тому, що виключення класу `RuntimeException` свідчать про помилки в програмі, і 'єдино розумний метод їх обробки — виправити вихідний текст програми і перекомпілювати її. Що стосується класу `Error`, то ці виключення дуже важко локалізувати і на стадії компіляції неможливо визначити місце їх появи.

Напроти, виникнення перевірючого виключення показує, що програма недостатньо продумана, не всі можливі ситуації описані. Така програма повинна бути допрацьована, про що і нагадує компілятор. Якщо метод або конструктор викидає декілька виключень, то їх треба перечислити через кому після слова `throws`. Заголовок метода `main()` лістингу 16.1, якби виключення, які він викидає, не були б об'єктами підкласів класу `RuntimeException`, належало б написати так:

```

public static void main(String[] args)
throws ArithmeticException, ArrayIndexOutOfBoundsException{
// Зміст методу
}

```

Перенесемо тепер обробку ділення на нуль в метод `f()` і додамо трасуючий друк, як це зроблено в лістингі 16.4. Результат — на рис. 16.3.

Лістинг 16.4. Обробка виключення в методі

```

class SimpleExt3{
private static void f(int n){ // throws ArithmeticException{
try{
System.out.println(" 10 / n = " + (10 / n) ) ;
System.out.println("From f() after results output");
}catch(ArithmeticException ae){
System.out.println("From f() catch: " + ae) ;
// throw ae;
}finally{ System.out.println("From f() finally"); }
}
public static void main(String[] args){
try{
int n = Integer.parseInt(args[0]);
System.out.println("After parseInt()");
f (n);
System.out.println("After results output"); }
catch(ArithmeticException ae){
System.out.println("From Arithm.Exc. catch: "+ae);
}catch(ArrayIndexOutOfBoundsException arre){
System.out.println("From Array.Exc. catch: "+arre);
}finally{ System.out.println("From finally");}
System.out.println("After all actions");
}
}

```

```

Командная строка
C:\>jdk1.3\bin\javac SimpleExt3.java
C:\>jdk1.3\bin\java SimpleExt3 5
After parseInt()
 10 / n = 2
From f() after results output
From f() finally
After results output
From finally
After all actions

C:\>jdk1.3\bin\java SimpleExt3 0
After parseInt()
From f() catch: java.lang.ArithmeticException: / by zero
From f() finally
After results output
From finally
After all actions

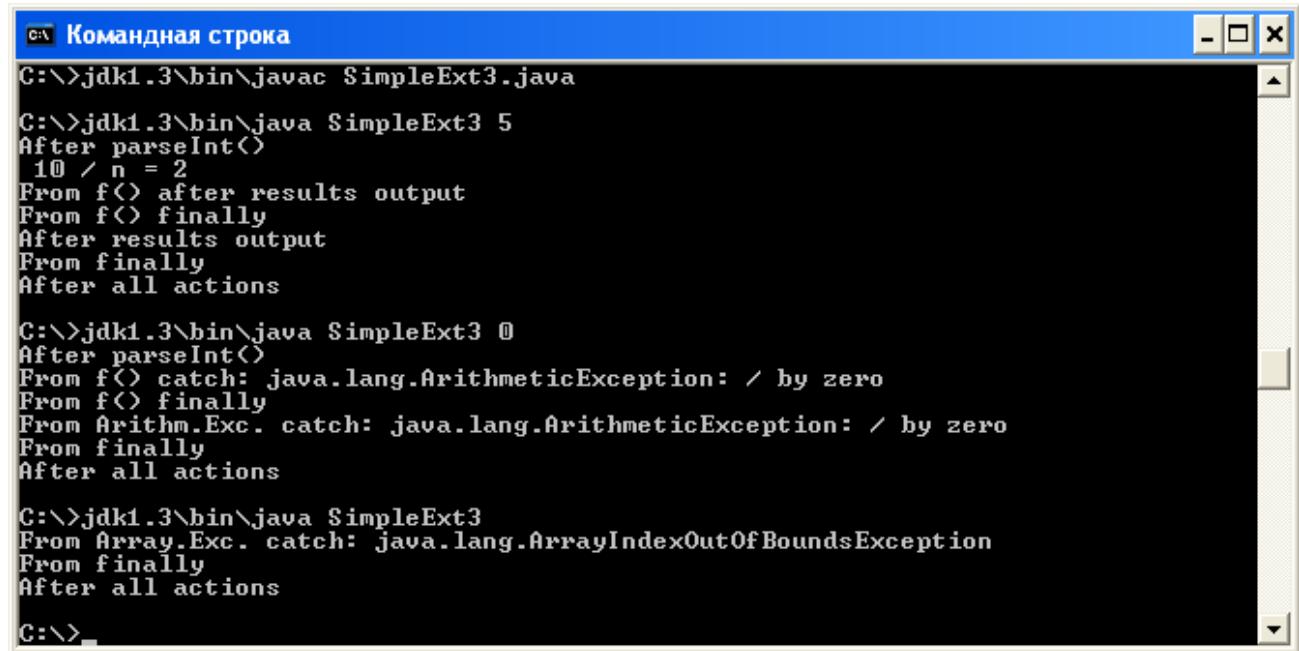
C:\>_

```

Рис. 16.3. Обробка виключення в методі

Уважно прослідкуйте за передачею управління і замітьте, що виключенні класу `ArithmeticException` уже не

викидається в метод `main()`. Оператор `try {} catch() {}` в методі `f()` можна розглядати як вкладений в оператор обробки виключень в методі `main()`. При необхідності виключення можна викинути оператором `throw()`. В лістингі 16.4 цей оператор показаний як коментар. Приберіть символ коментаря `//`, перекомпілюйте програму і подивіться, як зміниться її виведення.



```

C:\>jdk1.3\bin\javac SimpleExt3.java
C:\>jdk1.3\bin\java SimpleExt3 5
After parseInt()
 10 / n = 2
From f() after results output
From f() finally
After results output
From finally
After all actions

C:\>jdk1.3\bin\java SimpleExt3 0
After parseInt()
From f() catch: java.lang.ArithmeticsException: / by zero
From f() finally
From Arithm.Exc. catch: java.lang.ArithmeticsException: / by zero
From finally
After all actions

C:\>jdk1.3\bin\java SimpleExt3
From Array.Exc. catch: java.lang.ArrayIndexOutOfBoundsException
From finally
After all actions

C:\>_

```

16.4. Оператор `throw`

Цей оператор дуже простий: після слова `throw` через пробіл записується об'єкт класу-виключення. Досить часто він створюється прямо в операторі `throw`, наприклад:

```
throw new ArithmeticsException();
```

Оператор можна записати в будь-якому місці програми. Він негайно викидає записаний в ньому об'єкт-виключення і далі обробка цього виключення іде як звичайно, ніби-то тут відбулося ділення на нуль або інша дія, що викликала виключення класу `ArithmeticsException`. Отже, кожний блок `catch()` перехоплює один певний тип виключень. Якщо треба однаково обробити декілька типів виключень, то можна скористатися тим, що класи-виключення утворюють ієархію. Змінимо ще раз лістинг 16.2, одержавши лістинг 16.5.

Лістинг 16.5. Обробка декількох типів виключень

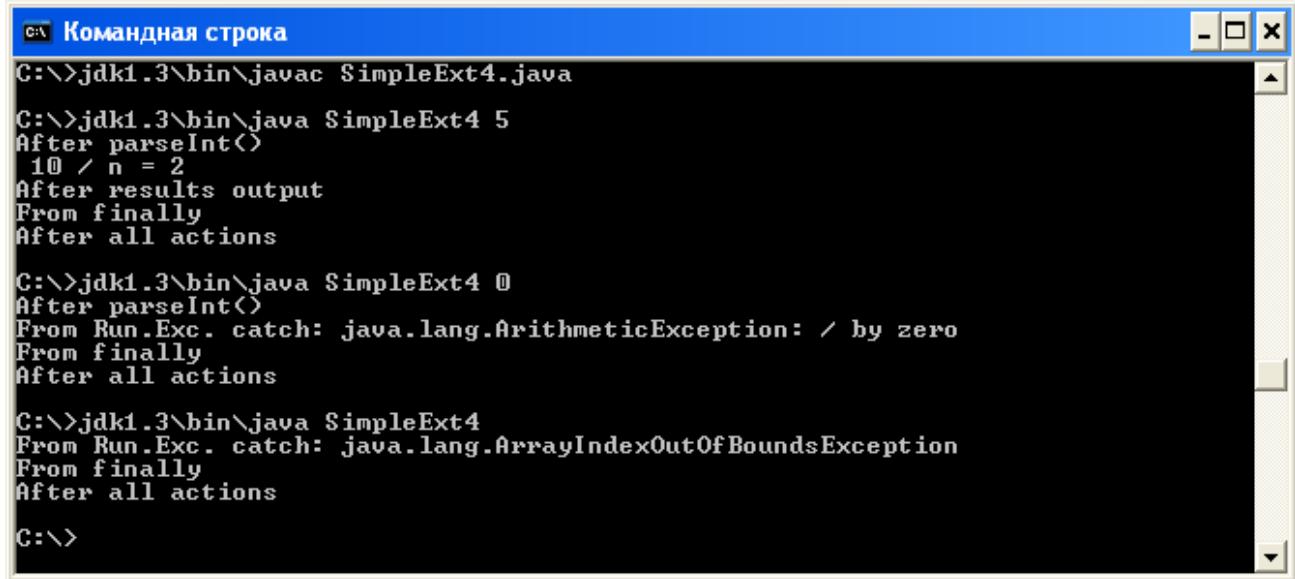
```

class SimpleExt4{
public static void main(String[] args){
try{
int n = Integer.parseInt(args[0]);
System.out.println("After parseInt()");
System.out.println(" 10 / n = " + (10 / n) );
System.out.println("After results output");
}catch(RuntimeException ae){
System.out.println("From Run.Exc. catch: "+ae);
}finally{
System.out.println("From finally");
}
System.out.println("After all actions");
}

```

}

В лістингі 16.5 два блоки `catch()` {} замінені одним блоком, перехоплюючим виключення класу `RuntimeException`. Як видно на рис. 16.4, цей блок перехоплює обидва виключення. Чому? Тому що це виключення підкласів класу `RuntimeException`.



```

C:\>jdk1.3\bin\javac SimpleExt4.java
C:\>jdk1.3\bin\java SimpleExt4 5
After parseInt()
10 / n = 2
After results output
From finally
After all actions

C:\>jdk1.3\bin\java SimpleExt4 0
After parseInt()
From Run.Exc. catch: java.lang.ArithmeticsException: / by zero
From finally
After all actions

C:\>jdk1.3\bin\java SimpleExt4
From Run.Exc. catch: java.lang.ArrayIndexOutOfBoundsException
From finally
After all actions

C:\>

```

Рис. 16.4. Перехват декількох типів виключень

Таким чином, переміщаючись по ієархії класів-виключень, ми можемо обробляти зразу більш-менш крупні сукупності виключень. Розглянемо детальніше ієархію класів-виключень.

16.5. Ієархія класів-виключень

Всі класи-виключення розширяють клас `Throwable` — безпосереднє розширення класу `Object`. У класі `Throwable` і у всіх його розширеннях два конструктори:

- `Throwable()` — конструктор по замовчуванню;
- `Throwable (String message)` — створюваний об'єкт буде містити довільне повідомлення `message`.

Записане в конструкторі повідомлення можна отримати потім методом `getMessage()`. Якщо об'єкт створювався конструктором по замовчуванню, то даний метод поверне `null`. Метод `toString()` повертає короткий опис події, саме він працював у попередніх лістингах. Три методи виводять повідомлення про всі методи, що зустрілися на шляху "польоту" виключення:

- `printStackTrace()` — виводить повідомлення в стандартне виведення, як правило, це консоль;
- `printStackTrace(PrintStream stream)` — виводить повідомлення в байтовий потік `stream`;
- `printStackTrace(PrintWriter stream)` — виводить повідомлення в символьний потік `stream`.

У класі `Throwable` два безпосередніх наслідники — класи `Error` і `Exception`. Вони не додають нових методів, а служать для розподілу класів-виключень на два великих сімейства - сімейство класів-помилок (`error`) і сімейство власне класів-виключень (`exception`). Класи-помилки, розширяючі клас `Error`, свідчать про виникнення складних ситуацій у віртуальній машині `Java`. Їх обробка вимагає глибокого розуміння всіх тонкощів роботи `JVM`. Її не рекомендується виконувати в звичайній програмі. Не радять навіть викидати помилки оператором `throw`. Не треба робити свої класи-виключення розширеннями класу `Error` або якогось його підкласу. Імена класів-помилок закінчуються словом `Error`. Класи-виключення, розширяючі клас `Exception`, відмічають виникнення звичайної нештатної ситуації, яку можна і навіть потрібно

обробити. Такі виключення належить викинути оператором `throw`. Класів-виключень дуже багато, більше двохсот. Вони розкидані буквально по всіх пакетах [J2SDK](#). В більшості випадків ви здатні підібрати готовий клас-виключення для обробки виключчих ситуацій в своїй програмі. При бажанні можна створити і свій клас-виключення, розширивши клас [Exception](#) або будь-який його підклас.

Серед класів-виключень виділяється клас [RuntimeException](#) — пряме розширення класу [Exception](#). В ньому і його підкласах відмічаються виключення, що виникли при роботі [JVM](#), але не такі серйозні, як помилки. Їх можна обробляти і викидати, розширяти своїми класами, але краще довірити це [JVM](#), оскільки частіше всього це просто помилка в програмі, яку треба виправити. Особливість виключень даного класу в тому, що їх не треба відмічати в заголовку метода поміткою `throws`. Імена класів-виключень закінчуються словом [Exception](#).

16.6. Порядок обробки виключень

Блоки `catch () {}` перехоплюють виключення в порядку написання цих блоків. Це правило приводить до цікавих результатів. В лістингі 16.2 ми записали два блоки перехоплення `catch()` і обидва блоки виконувались при виникненні відповідного виключення. Це відбувалося тому, що класи-виключення [ArithmetiException](#) і [ArrayIndexOutOfBoundsException](#) знаходяться на різних гілках ієархії виключень. Інакше буде, якщо блоки `catch() {}` перехоплюють виключення, розташовані на одній гілці. Якщо в лістингі 16.4 після блоку, перехоплюючого [RuntimeException](#), помістити блок, обробляючий вихід індекса за межі:

```
try{
    // Оператори, викликаючі виключення
} catch(RuntimeException re) {
    // Якась обробка
} catch(ArrayIndexOutOfBoundsException ae) {
    // Ніколи не буде виконано!
}
```

то він не буде виконуватися, оскільки виключення цього типу являється, до того ж, виключенням загального типу [RuntimeException](#) і буде перехоплюватися попереднім блоком `catch () {}`.

16.7. Створення власних виключень

Перш за все, треба чітко визначити ситуації, в яких буде виникати ваші власні виключення, і подумати, не чи не стане його перехоплення заразом перехоплювати також і інші, не враховані вами виключення. Потім треба вибрати суперклас створюваного класу-виключення. Ним може бути клас [Exception](#) або один із його чисельних підкласів. Після цього можна написати клас-виключення. Його ім'я повинне завершатися словом [Exception](#). Як правило, цей клас складається тільки із двох конструкторів і перевизначення методів `toString()` і `getMessage()`.

Розглянемо простий приклад. Нехай метод `handle(int cipher)` обробляє арабські цифри 0—9, які передаються йому в аргументі `cipher` типу `int`. Ми хочемо викинути виключення, якщо аргумент `cipher` виходить за діапазон 0—9. Перш за все, упевнімося, що такого виключення немає в ієархії класів [Exception](#). До всього іншого, не відслідковується і більш загальна ситуація попадання цілого числа в якийсь діапазон. Тому будемо розширяти наш клас. Назовемо його [cipherException](#), прямо від класу [Exception](#). Визначимо клас [CipherException](#), як показано в лістингі 16.6, і використаємо його в класі [ExceptDemo](#). На рис. 16.5 продемонстровано виведення цієї програми.

Лістинг 16.6. Створення класу-виключення

```
class CipherException extends Exception{
    private String msg;
    CipherException(){ msg = null; }
    CipherException(String s){ msg = s; }
    public String toString(){
        return "CipherException (" + msg + ")";
    }
}
```

```

class ExceptDemo{
static public void handle(int cipher) throws CipherException{
System.out.println("handle()'s beginning");
if (cipher < 0 || cipher > 9)
throw new CipherException("") + cipher);
System.out.println("handle()'s ending");
}
public static void main(String[] args){
try{
handle(1) ;
handle(10) ;
catch(CipherException ce){
System.out.println("caught " + ce) ;
ce.printStackTrace();
}
}
}

```

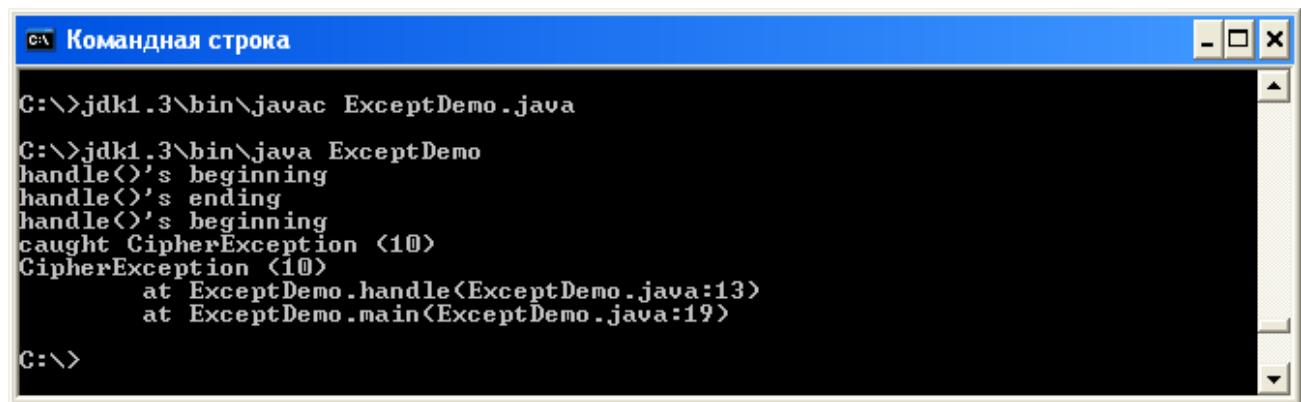


Рис. 16.5. Обробка власного виключення

16.8. Заключення

Обробка виключчів ситуацій стала тепер обов'язковою частиною об'єктно-орієнтованих програм. Застосовуючи методи класів **J2SDK** і інших пакетів, звертайте увагу на те, які виключення вони викидають, і обробляйте їх. Виключення різко змінюють хід виконання програми, роблять його заплутаним. Не займайтесь складною обробкою, памятайте про принцип **KISS**. Наприклад, із блоку **finally{}}** можна викинути виключення і обробити його в іншому місці. Подумайте, що відбудеться в цьому випадку з виключенням, що виникло в блоці **try{}**? Воно ніде не буде перехоплено і оброблено.

Урок 17

Підпроцеси

- Клас Thread
- Синхронізація підпроцесів
- Узгодження роботи декількох підпроцесів
- Пріоритети підпроцесів
- Підпроцеси-демони
- Групи підпроцесів
- Заключення

17.1. Процес

Основне поняття сучасних операційних систем — процес (process). Як і всі загальні поняття, процес важко визначити. Можна розуміти під процесом виконувану (runnable) програму, але треба памятати про те, що у процесу єсть декілька станів. Процес може в будь-який момент перейти до виконання машинного коду іншої програми, а також "заснути" (sleep) на деякий час, призупинивши виконання програми. Він може бути вивантажений на диск. Кількість станів процесу і їх особливості залежать від операційної системи.

Всі сучасні операційні системи багатозадачні (multitasking), вони запускають і виконують зразу декілька процесів. Одночасно може працювати браузер, текстовий редактор, музичний програвач. На екрані дисплея відкриваються декілька вікон, кожне з яких звязано із своїм працюючим процесом. Якщо на комп'ютері тільки один процесор, то він переключається з одного процесу на другий, створюючи видимість одночасної роботи. Переключення відбувається по закінченню одного або декількох "тиків" (ticks). Розмір тику залежить від тактової частоти процесора і звичайно має порядок 0,01 секунди. Процесам назначаються різні пріоритети (priority). Процеси з низьким пріоритетом не можуть перервати виконання процесу з більш високим пріоритетом, вони менше займають процесор, тому виконуються повільно, як говорять, "на фоні". Самий високий пріоритет у системних процесів, наприклад, у диспетчера (scheduler), який як раз і займається переключенням процесора з процесу на процес. Такі процеси не можна переривати, поки вони не закінчать роботу, інакше комп'ютер швидко прийде в хаотичний стан.

Кожному процесу виділяється певна область оперативної пам'яті для розміщення коду програми і її даних — його адресний простір. В цю ж область записується частина даних про процес, складаюча його контекст (context). Дуже важливо розділити адресний простір різних процесів, щоб вони не могли змінити код і дані один одного. Операційні системи по-різному відносяться до забезпечення захисту адресних просторів процесів. MS Windows NT/2000 ретельно розділяють адресні простори, витрачаючи на це багато ресурсів і часу. Це підвищує надійність виконання програми, але утруднює створення процесу. Такі операційні системи погано справляються з управлінням величного числа процесів.

Операційні системи сімейства UNIX менше турбуються про захист пам'яті, але легше створюють процеси і здатні управляти сотнею одночасно працюючих процесів. Крім управління роботою процесів операційна система повинна забезпечити засоби їх взаємодії: обмін сигналами і повідомленнями, створення спільних декільком процесам областей пам'яті і виконуваного коду програми. Ці засоби також вимагають ресурсів і уповільнюють роботу комп'ютера.

Роботу багатозадачної системи можна спростити і прискорити, якщо дозволити взаємодіючим процесам працювати в одному адресному просторі. Такі процеси називаються threads. Буквальний переклад — "нитка", але ми зупинимося на слові "підпроцес". Підпроцеси створюють нові труднощі для операційної системи — треба дуже уважно слідкувати за тим, щоб вони не заважали один одному при запису в спільні ділянки пам'яті, — але зате полегшують взаємодію підпроцесів. Створення підпроцесів і управління ними — це справа операційної системи, але в Java введені засоби для виконання цих дій. Оскільки програми, написані на Java, повинні працювати у всіх операційних системах, ці засоби дозволяють виконувати тільки самі загальні дії.

Коли операційна система запускає віртуальну машину Java для виконання додатку, вона створює один процес з декількома підпроцесами. Головний (main) підпроцес виконує байт-коди програми, а саме, він зразу ж звертається до методу main() додатку. Цей підпроцес може породити нові підпроцеси, які, в свою

чергу, здатні породити підпроцеси і т. д. Головним підпроцесом аплета являється один із підпроцесів браузера, в якому аплет виконується. Головний підпроцес не грає ніякої особливої ролі, просто він створюється першим.

Підпроцес в **Java** створюється і управляється методами класу **Thread**. Після створення обєкта цього класу одним із його конструкторів новий підпроцес запускається методом **start()**. Отримати посилку на поточний підпроцес можна статичним методом **Thread.currentThread()**;

Клас **Thread** реалізує інтерфейс **Runnable**. Цей інтерфейс описує тільки один метод **run()**. Новий підпроцес буде виконувати те, що записано в цьому методі. Між іншим, клас **Thread** містить тільки пусту реалізацію методу **run()**, тому клас **Thread** не використовується сам по собі, він завжди розширяється. При його розширенні метод **run()** перевизначається. **Метод run()** не містить аргументів, так як нікому передавати їх значення в метод. Він не повертає значення, його нікуди передавати. До методу **run()** не можна звернутися із програми, це завжди робиться автоматично виконуючою системою **Java** при запуску нового підпроцесу методом **start()**.

Отже, задати дії створюваного підпроцесу можна двома способами: розширити клас **Thread** або реалізувати інтерфейс **Runnable**. Перший спосіб дозволяє використовувати методи класу **Thread** для управління підпроцесом. Другий спосіб застосовується в тих випадках, коли треба тільки реалізувати метод **run()**, або клас, створюючий підпроцес, уже розширяє якийсь інший клас. Подивимося, які конструктори і методи містить клас **Thread**.

17.2. Клас Thread

В класі **Thread** сім конструкторів:

- **Thread(ThreadGroup group, Runnable target, String name)** — створює підпроцес з іменем **name**, належний групі **group** і виконуючий метод **run()** обєкта **target**. Це основний конструктор, всі інші звертаються до нього з тим чи іншим параметром, рівним **null**;
- **Thread()** — створюваний підпроцес буде виконувати свій метод **run()**;
- **Thread(Runnable target);**
- **Thread(Runnable target, String name);**
- **Thread(String name);**
- **Thread(ThreadGroup group, Runnable target);**
- **Thread(ThreadGroup group, String name).**

Ім'я підпроцесу **name** не має ніякого значення, воно не використовується віртуальною машиною **Java** і застосовується тільки для відрізняння підпроцесів в програмі. Після створення підпроцесу його треба запустити методом **start()**. Віртуальна машина **Java** почне виконувати метод **run()** цього обєкта-підпроцеса. Підпроцес завершить роботу після виконання метода **run()**. Для знищенння обєкта-підпроцеса вслід за цим він повинен присвоїти значення **null**.

Виконуваний підпроцес можна призупинити статичним методом **sleep (long ms)** на **ms** мілісекунд. Цей метод ми уже використовували в попередніх уроках. Якщо обчислювальна система може відраховувати наносекунди, то можна призупинити підпроцес з точністю до наносекунд методом **sleep(long ms, int nanosec)**. В лістингі 17.1 приведено найпростіший приклад. Головний підпроцес створює два підпроцеси з іменами **Thread1** і **Thread 2**, виконуючих один і той же метод **run()**. Цей метод просто виводить 20 раз текст на екран, а потім повідомляє про своє завершення.

Лістинг 17.1. Два підпроцеси, запущені із головного підпроцесу

```
class OutThread extends Thread{
private String msg;
OutThread(String s, String name){
super(name); msg = s;
}
public void run()
{
for(int i = 0; i < 20; i++) {
```

```

// try{
// Thread.sleep(100);
// }catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of " + getName());
}
} class TwoThreads{
public static void main(String[] args){
new OutThread("HIP", "Thread 1").start();
new OutThread("hop", "Thread 2").start();
System.out.println();
}
}

```

На рис. 17.1 показано результат двох запусків програми лістингу 17.1. Як бачите, в першому випадку підпроцес **Thread1** встиг відпрацювати повністю до переключення процесора на виконання другого підпроцесу. В другому випадку робота підпроцеса **Thread1** була перервана, процесор переключився на виконання підпроцеса **Thread 2**, встиг виконати його повністю, а потім переключився знову на виконання підпроцеса **Thread1** і завершив його.



Рис. 17.1. Два підпроцеси працюють без затримки

Це дуже повчальний приклад, але якщо у вас сучасний комп'ютер з більшою швидкістю дії, то запустивши на ньому програму лістингу 17.1 ви можете побачити зовсім іншу картину. Підпроцеси можуть спрацювати так швидко, що переключення не здійсниться.

Приберемо в лістингі 17.1 коментарі, затримавши тим самим виконання кожної ітерації циклу на 0,1 секунди. Пуста обробка виключення **InterruptedException** означає, що ми ігноруємо спробу переривання роботи підпроцеса. На рис. 17.2 показано результат двох запусків програми. Як бачите, процесор переключається з одного підпроцеса на інший, але в одному місці регулярність переключення порушується і раніше запущений підпроцес завершується пізніше.

Як же досягти узгодженості, як говорять, *синхронізації* (synchronization) підпроцесів? Обговоримо це нижче, а поки що покажемо ще два варіанти створення тієї ж самої програми. В лістингі 17.2 приведено другий варіант тієї ж програми: сам клас **TwoThreads2** являється розширенням класу **Thread**, а метод **run()** реалізується прямо в ньому.

Лістинг 17.2. Клас розширює Thread

```

class TwoThreads2 extends Thread{
private String msg;
TwoThreads2(String s, String name) {

```

```

super(name); msg = s;
}
public void run(){
for(int i = 0; i < 20; i++){
try{
Thread.sleep(100);
}catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of " + getName());
}
public static void main(String[] args){
new TwoThreads2("HIP", "Thread 1").start();
new TwoThreads2("hop", "Thread 2").start();
System.out.println();
}
}

```

Третій варіант: клас `TwoThreads3` реалізує інтерфейс `Runnable`. Цей варіант записаний в лістингі 17.3. Тут не можна використовувати методи класу `Thread`, але зате клас `TwoThreads3` може бути розширенням іншого класу. Наприклад, можна зробити його аплемтом, розширивши клас `Applet` або `JApplet`.

Лістинг 17.3. Реалізація інтерфейса `Runnable`

```

class TwoThreads3 implements Runnable{
private String msg;
TwoThreads3(String s){ msg = s; }
public void run(){
forint i = 0; i < 20; i++){
try{
Thread.sleep(100);
}catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of thread.");
}
public static void main (String[] args){
new Thread(new TwoThreads3("HIP"), "Thread 1").start ();
new Thread(new TwoThreads3("hop"), "Thread 2").start ();
System.out.println();
}
}

```



Рис. 17.2. Підпроцеси працюють із затримкою

Частіше всього в новому підпроцесі задаються нескінчені дії, виконувані на фоні основних дій: програється музика, на екрані крутиться анімований логотип фірми, біжить рекламний рядок. Для реалізації такого підпроцеса в методі `run()` задається нескінчений цикл, зупинюваний після того, як об'єкт-підпроцес отримає значення `null`. В лістингі 17.4 показано четвертий варіант тієї ж самої програми, в якій метод `run()` виконується до тих пір, доки поточний об'єкт-підпроцес `th` співпадає з об'єктом `go`, запустившим поточний підпроцес. Для переривання його виконання передбачений метод `stop()`, до якого звертається головний підпроцес. Ця стандартна конструкція, рекомендована документацією [J2SDK](#). Головний підпроцес в даному прикладі тільки створює об'єкти-підпроцеси, чекає одну секунду і зупиняє їх.

Лістинг 17.4. Зупинка роботи підпроцесів

```
class TwoThreadsS implements Runnable{
private String msg;
private Thread go;
TwoThreadsS(String s){
msg = s;
go = new Thread(this);
go.start();
}
public void run(){
Thread th = Thread.currentThread();
while(go == th){
try{
Thread.sleep(100);
}catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of thread.");
}
public void stop(){ go = null; }
public static void main(String[] args){
TwoThreadsS th1 = new TwoThreadsS("HIP");
TwoThreadsS th2 = new TwoThreadsS("hop");
try{
Thread.sleep(1000);
}catch(InterruptedException ie){}
th1.stop(); th2.stop();
System.out.println();
}
}
```

17.3. Синхронізація підпроцесів

Основна складність при написанні програм, в яких працюють декілька підпроцесів — це узгодити сумісну роботу підпроцесів із загальними комірками пам'яті. Класичний приклад — банківська трансакція, в якій змінюється залишок на рахунку клієнта з номером `numDep`. Припустимо, що для її виконання запрограмовані такі дії:

```
Deposit myDep = getDeposit(numDep); // Отримуємо рахунок з номером numDep
int rest = myDep.getRest(); // Отримуємо залишок на рахунку myDep
Deposit newDep = myDep.operate(rest, sum); // Змінюємо залишок на величину sum
myDep.setDeposit(newDep); // Заносимо новий залишок на рахунок myDep
```

Нехай на рахунку лежить 1000 гривнів. Ми вирішили зняти з рахунку 500 гривнів, а в той же час поступив поштовий переказ на 1500 гривнів. Ці дії виконують різні підпроцеси, але змінюють вони один і той же рахунок `myDep` з номером `numDep`. Подивившись ще раз на рис. 17.1 і 17.2, ви повірите, що послідовність дій може скластися так. Перший підпроцес проробить віднімання 1000 -500, в цей час другий підпроцес виконає всі три дії і запише на рахунок $1000+1500 = 2500$ гривнів, після чого перший підпроцес виконає свою останню дію і у нас на рахунку виявиться 500 гривнів. Навряд ли вам сподобається таке виконання

двох трансакцій.

В мові **Java** прийнятий виход із цього положення, названий в теорії операційних систем **монітором (monitor)**. Він заключається в тому, що підпроцес блокує об'єкт, з яким працює, щоб другі підпроцеси не могли звернутися до даного об'єкту, поки блокування не буде знято. В нашому прикладі перший підпроцес повинен спочатку заблокувати рахунок **myDep**, потім повністю виконати всю трансакцію і зняти блокування. Другий підпроцес призупиниться і стане чекати, поки блокування не буде знято, після чого почне працювати з об'єктом **myDep**. Все це робиться одним оператором **synchronized() {}**, як показано нижче:

```
Deposit myDep = getDeposit(numDep);
synchronized(myDep) {
    int rest = myDep.getRest();
    Deposit newDep = myDep.operate(rest, sum);
    myDep.setDeposit(newDep);
}
```

В заголовку оператора **synchronized** в дужках указується посилання на об'єкт, який буде заблокований перед виконанням блоку. Об'єкт буде недоступний для інших підпроцесів, поки виконується блок. Після виконання блоку блокування знімається. Якщо при написанні якогось методу виявилось, що в блок **synchronized** входять всі оператори цього методу, то можна просто помітити метод словом **synchronized**, зробивши його **синхронізованим (synchronized)**:

```
synchronized int getRest() {
    // Тіло методу
}
synchronized Deposit operate(int rest, int sum) {
    // Тіло методу
}
synchronized void setDeposit(Deposit dep) {
    // Тіло методу
}
```

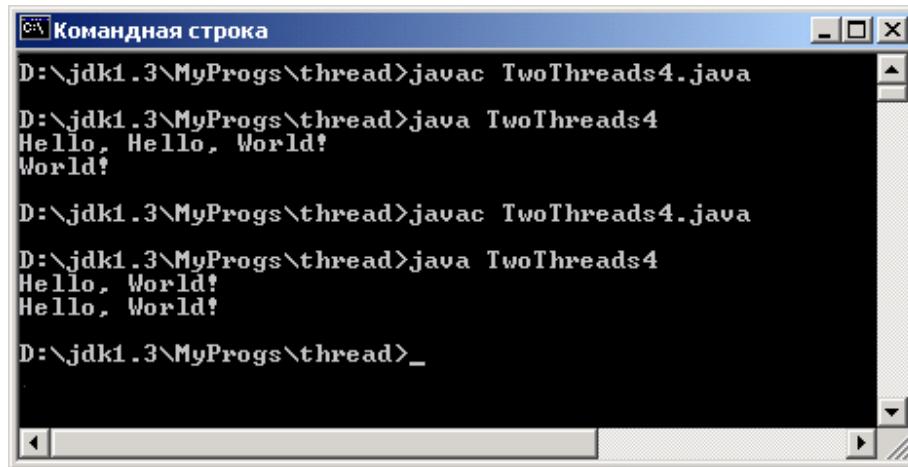
В цьому випадку блокується об'єкт, виконуючий метод, тобто **this**. Якщо всі методи, до яких не повинні одночасно звертатися декілька підпроцесів, помічені **synchronized**, то оператор **synchronized() {}** уже не потрібний. Тепер, якщо один підпроцес виконує синхронізований метод об'єкта, то інші підпроцеси уже не можуть звернутися до жодного синхронізованого методу того ж самого об'єкта.

Приведемо простий приклад. Метод **run()** в лістингі 17.5 виводить рядок **"Hello, World!"** із затримкою в 1 секунду між словами. Цей метод виконується двома підпроцесами, працюючими з одним об'єктом **th**. Програма виконується два рази. Перший раз метод **run()** не синхронізований, другий раз синхронізований, його заголовок показано в лістингі 17.4 як коментар. Результат виконання програми представлений на рис. 17.3.

Лістинг 17.5. Синхронізація методу

```
class TwoThreads4 implements Runnable{
public void run(){
    // synchronized public void run(){
    System.out.print("Hello, ");
    try{
        Thread.sleep(1000);
    }catch(InterruptedException ie){}
    System.out.println("World!");
}
public static void main(String[] args){
    TwoThreads4 th = new TwoThreads4();
    new Thread(th).start();
    new Thread(th).start();
}
```

```
}
```



```
D:\jdk1.3\MyProgs\thread>javac TwoThreads4.java
D:\jdk1.3\MyProgs\thread>java TwoThreads4
Hello, Hello, World!
World!
D:\jdk1.3\MyProgs\thread>javac TwoThreads4.java
D:\jdk1.3\MyProgs\thread>java TwoThreads4
Hello, World!
Hello, World!
D:\jdk1.3\MyProgs\thread>_
```

Рис. 17.3. Синхронізація методу

Дії, що входять в синхронізований блок або метод створюють *критичну ділянку* (*critical section*) програми. Декілька підпроцесів, що збираються виконувати критичну ділянку, стають в чергу. Це сповільнює роботу програми, тому для швидкого її виконання критичних ділянок повинно бути як можна менше, і вони повинні бути як можна коротші. Багато методів [Java 2 SDK](#) синхронізовані. Зверніть увагу, що на рис. 17.1 слова виводяться впереміжку, але кожне слово виводиться повністю. Це відоується тому, що метод [print\(\)](#) класу [Printstream](#) синхронізований, при його виконанні вихідний потік [System.out](#) блокується до тих пір, доки метод [print\(\)](#) не закінчить свою роботу.

Отже, ми можемо легко організувати послідовний доступ декількох підпроцесів до полів одного обєкта за допомогою оператора [synchronized\(\)](#). Синхронізація забезпечує *взаємно виключаюче* (*mutually exclusive*) виконання підпроцесів. Але що робити, якщо потрібний сумісний доступ декількох підпроцесів до спільних обєктів? Для цього в [Java](#) існує механізм очікування і сповіщення (*wait-notify*).

17.4. Узгодження роботи декількох підпроцесів

Можливість створення багатопотокових програм закладена в [Java](#) з самого її створення. В кожному обєкті є три методи [wait\(\)](#) і один метод [notify\(\)](#), дозволяючи призупиняти роботу підпроцесу з цим обєктом, дозволити іншому підпроцесу попрацювати з обєктом, а потім сповістити ([notify](#)) перший підпроцес про можливість продовження роботи. Ці методи визначені прямо в класі [Object](#) і наслідуються всіма класами. З кожним обєктом звязано багато підпроцесів, очікуючих доступу до обєкту ([wait set](#)). Спочатку цей "зал очікування" порожній.

Основний метод [wait \(long millisec\)](#) призупиняє поточний підпроцес [this](#), працюючий з обєктом, на [millisec](#) мілісекунд і переводить його в "зал очікування", в множину очікуючих підпроцесів. Звернення до цього методу допускається тільки в синхронізованому блоці або методі, щоб бути впевненими в тому, що з обєктом працює тільки один підпроцес. Через [millisec](#) або після того, як обєкт отримає сповіщення методом [notify\(\)](#), підпроцес готовий відновити роботу. Якщо аргумент [millisec](#) рівний 0, то час очікування не визначено і відновлення роботи підпроцесу можна тільки після того, як обєкт отримає сповіщення методом [notify\(\)](#). Відміна даного методу від методу [sleep\(\)](#) в тому, що метод [wait\(\)](#) знімає блокування з обєкта. З обєктом може працювати один із підпроцесів із "зала очікування", звичайно той, який чекав довше всіх, хоч це не гарантується специфікацією [JLS](#).

Другий метод [wait \(\)](#) еквівалентний [wait\(0\)](#). Третій метод [wait \(long millisec, int nanosec\)](#) уточнює затримку на наносес наносекунд, якщо їх зуміє відрахувати операційна система. Метод [notify\(\)](#) виводить із "зали очікування" тільки один, довільно вибраний підпроцес. Метод [notifyAll\(\)](#) виводить із стану очікування всі підпроцеси. Ці методи теж повинні виконуватися в синхронізованому блоці або методі. Як же застосувати все це для узгодженого доступу до обєкта? Як завжди, краще всього пояснити це на прикладі.

Звернемося знову до схеми "постачальник-споживач", уже використану в уроці 15. Один підпроцес, постачальник, робить обчислення, другий, споживач, очікує результати цих обчислень і використовує їх в міру поступання. Підпроцеси передають інформацію через спільний екземпляр `st` класу `store`. Робота цих підпроцесів повинна бути узгоджена. Споживач зобовязаний чекати, доки постачальник не занесе результат обчислення в об'єкт `st`, а постачальник повинен чекати, доки споживач не візьме цей результат. Для простоти постачальник просто заносить в спільний об'єкт класу `store` цілі числа, а споживач лише забирає їх. В лістингі 17.6 клас `store` не забезпечує узгодженості отримання і видачі інформацію. Результат роботи показаний на рис. 17.4.

Лістинг 17.6. Неузгоджені підпроцеси

```

class Store{
private int inform;
synchronized public int getInform(){ return inform; }
synchronized public void setInform(int n){ inform = n; }
}
class Producer implements Runnable{
private Store st;
private Thread go;
Producer(Store st){
this.st = st;
go = new Thread(this);
go.start();
}
public void run(){
int n = 0;
Thread th = Thread.currentThread();
while(go == th){
st.setInform(n);
System.out.print("Put: " + n + " ");
n++;
}
}
public void stop(){ go = null;
}
}
class Consumer implements Runnable{
private Store st;
private Thread go;
Consumer(Store st){
this.st = st;
go = new Thread(this);
go.start();
}
public void run(){
Thread th = Thread.currentThread();
while(go == th) System.out.println("Got: " + st.getInform());
}
public void stop(){ go = null; }
}
class ProdCons{
public static void main(String[] args){
Store st = new Store();
Producer p = new Producer(st);
Consumer c = new Consumer(st);
try{
Thread.sleep(30);
} catch(InterruptedException ie){}
}
}

```

```

p.stop(); c.stop();
}
}

```

Рис. 17.4. Неузгоджена робота двох підпроцесів

В лістингі 17.7 в клас `store` внесено логічне поле `ready`, що відмічає процес отримання і видачі інформації. Коли нова порція інформації отримана від постачальника `Producer`, в полі `ready` заноситься значення `true`, отримувач `consumer` може забирати цю порцію інформації. Після видачі інформації змінна `ready` становиться рівною `false`. Але цього мало. Те, що отримувач може забрати продукт, не означає, що він дійсно забере його. Тому в кінці методу `setinform()` отримувач сповіщається про поступанні продукту методом `notify()`. Поки поле `ready` не прийме потрібне значення, підпроцес переводиться в "залу очікування" методом `wait()`. Результат роботи програми з обновленим класом `store` показаний на рис. 17.5.

Лістинг 17.7. Узгодження отримання і видачі інформації

```

class Store{
private int inform = -1;
private boolean ready;
synchronized public int getinform(){
try{
if (! ready) wait();
ready = false;
return inform;
}catch(InterruptedException ie){
}finally{
notify();
}
return -1;
}
synchronized public void setinform(int n){
if (ready)
try{
wait ();
}catch(InterruptedException ie){}
inform = n;
ready = true;
notify();
}
}

```

Оскільки сповіщення поставщика в методі `getinform()` повинно відбуватися уже після відправки інформації оператором `return inform`, воно включено в блок `finally{}`

Зверніть увагу: повідомлення "Got: 0" відстає на один крок від дійсного отримання інформації.

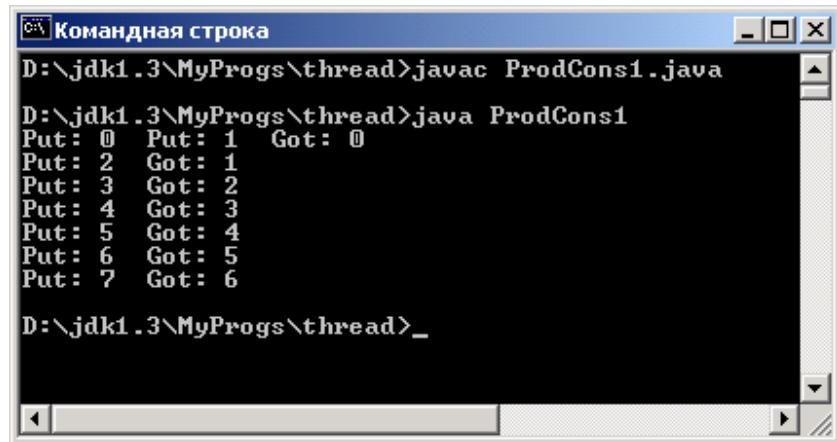


Рис. 17.5. Узгоджена робота підпроцесів

17.5. Пріоритети підпроцесів

Планувальник підпроцесів віртуальної машини [Java](#) призначає кожному підпроцесу одинаковий час виконання процесором, переключаючись з підпроцеса на підпроцес по закінченню цього часу. Інколи необхідно виділити якомусь підпроцесу більше або менше часу в порівнянні з іншим підпроцесом. В такому випадку можна задати підпроцесу більший або менший пріоритет. В класі [Thread](#) є три цілі статичні константи, що задають пріоритети:

- [NORM_PRIORITY](#) — звичайний пріоритет, який одержує кожний підпроцес при запуску, його числове значення 5;
- [MIN_PRIORITY](#) — найменший пріоритет, його значення 1;
- [MAX_PRIORITY](#) — найвищий пріоритет, його значення 10.

Крім цих значень можна задати будь-яке проміжне значення від 1 до 10, але треба памятати про те, що процесор буде переключатися між підпроцесами з однаковим вищим пріоритетом, а підпроцеси з меншим пріоритетом не стануть виконуватися, якщо тільки не призупинені всі підпроцеси з вищим пріоритетом. Тому для підвищення загальної продуктивності належить призупиняти час від часу методом [sleep\(\)](#) підпроцеси з високим пріоритетом.

Установити той чи інший пріоритет можна в будь-який час методом [setPriority\(int newPriority\)](#), якщо підпроцес має право змінювати свій пріоритет. Перевірити наявність такого права можна методом [checkAccess\(\)](#). Цей метод викидає виключення класу [SecurityException](#), якщо підпроцес не може змінити свій пріоритет. Породжені підпроцеси будуть мати той же пріоритет, що і підпроцес-батько. Отже, підпроцеси, як правило, повинні працювати з пріоритетом [NORM_PRIORITY](#). Підпроцеси більшу частину часу очікуючі настання якоїсь події, наприклад, натискання користувачем кнопки Вихід, можуть отримати більш високий пріоритет [MAX_PRIORITY](#). Підпроцеси, виконуючі тривалу роботу, наприклад, установку мережевого з'єднання або рисування зображення в пам'яті при подвійній буферизації, можуть працювати з нижчим пріоритетом [MIN_PRIORITY](#).

17.6. Підпроцеси-демони

Робота програми починається з виконання метода [main\(\)](#) головним підпроцесом. Цей підпроцес може породити інші підпроцеси, вони, в свою чергу, здатні породити свої підпроцеси. Після цього головний підпроцес нічим не буде відрізнятися від решти підпроцесів. Він не слідкує за породженими ним підпроцесами, не чекає від них ніяких сигналів. Головний підпроцес може завершитися, а програма буде продовжувати роботу, доки не закінчить роботу останній підпроцес. Це правило не завжди зручне. Наприклад, якийсь із підпроцесів може призупинитися, очікуючи мережевого з'єднання, яке ніяк не може наступити. Користувач, не дочекавшись з'єднання, зупиняє роботу головного підпроцесу, але програма продовжує працювати.

Такі випадки можна врахувати, оголосивши деякі підпроцеси **демонами** (**daemons**). Це поняття не співпадає з поняттям демона в **UNIX**. Просто програма завершується по закінченні роботи останнього **користувальського** (**user**) підпроцесу, не чекаючи закінчення роботи демонів. Демони будуть примусово завершенні виконуючою системою **Java**. Оголосити підпроцес демоном можна зразу після його створення, перед запуском. Це робиться методом **setDaemon(true)**. Даний метод звертається до методу **checkAccess()** і може викинути **SecurityException**. Змінити статус демона після запуску процесу уже неможна. Всі підпроцеси, породжені демоном, теж будуть демонами. Для зміни їх статусу необхідно звернутися до методу **setDaemon(false)**.

17.7. Групи підпроцесів

Підпроцеси об'єднуються в групи. На початку роботи програми виконуюча система **Java** створює групу підпроцесів з іменем **main**. Всі підпроцеси по замовчуванню попадають в цю групу. В будь-який час програма може створити нові групи підпроцесів і підпроцеси, що входять в ці групи. Спочатку створюється група — екземпляр класу **ThreadGroup**, конструктором

ThreadGroup(String name)

При цьому група отримує ім'я, задане аргументом **name**. Потім цей екземпляр указується при створенні підпроцесів в конструкторах класу **Thread**. Всі підпроцеси попадуть в групу з іменем, заданим при створенні групи. Групи підпроцесів можуть утворювати ієрархію. Одна група породжується від другої конструктором

ThreadGroup(ThreadGroup parent, String name)

Групи підпроцесів використовуються головним чином для задання пріоритетів підпроцесам всередині групи. Зміна пріоритетів всередині групи не буде впливати на пріоритети підпроцесів зовні ієрархії цієї групи. Кожна група має максимальний пріоритет, установлений методом **setMaxPriority(int maxPri)** класу **ThreadGroup**. Ні один підпроцес із цієї групи не може перевищити значення **maxPri**, але пріоритети підпроцесів, задані до установки **maxPri**, не змінюються.

17.8. Заключення

Технологія **Java** по своїй суті — багатозадачна технологія, основана на **threads**. Це одна із причин, по яких технологія **Java** так і не може розумним способом реалізовуватися в **MS-DOS** і **Windows 3.1**, незважаючи на багато спроб. Тому, конструкуючи програму для **Java**, належить весь час памятати, що вона буде виконуватися в багатозадачному середовищі. Треба ясно представляти собі, що буде, якщо програма почне виконуватися одночасно кількома підпроцесами, виділяти критичні ділянки і синхронізувати їх. З другого боку, якщо програма виконує декілька дій, треба подумати, чи не зробити їх виконання одночасним, створивши додаткові підпроцеси і розподіливши їх пріоритети.

В лістингі 17.1 приведено найпростіший приклад. Головний підпроцес створює два підпроцеси з іменами Thread1 і Thread 2, виконуючих один і той же метод run(). Цей метод просто виводить 20 раз текст на екран, а потім повідомляє про своє завершення.

Лістинг 17.1. Два підпроцеси, запущені із головного підпроцесу

```
class OutThread extends Thread{
private String msg;
OutThread(String s, String name){
super(name); msg = s;
}
public void run()
{
for(int i = 0; i < 20; i++){
// try{
// Thread.sleep(100);
// }catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of " + getName());
}
} class TwoThreads{
public static void main(String[] args){
new OutThread("HIP", "Thread 1").start();
new OutThread("hop", "Thread 2").start();
System.out.println();
}
}
```

В лістингі 17.2 приведено другий варіант тієї ж програми: сам клас TwoThreads2 являється розширенням класу Thread, а метод run() реалізується прямо в ньому.

Лістинг 17.2. Клас розширює Thread

```
class TwoThreads2 extends Thread{
private String msg;
TwoThreads2(String s, String name){
super(name); msg = s;
}
public void run(){
for(int i = 0; i < 20; i++){
```

```

try{
Thread.sleep(100);
}catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of " + getName());
}
public static void main(String[] args){
new TwoThreads2("HIP", "Thread 1").start();
new TwoThreads2("hop", "Thread 2").start();
System.out.println();
}
}

```

Третій варіант: клас TwoThreads3 реалізує інтерфейс Runnable. Цей варіант записаний в лістингі 17.3. Тут не можна використовувати методи класу Thread, але зате клас TwoThreads3 може бути розширенням іншого класу. Наприклад, можна зробити його аплетом, розширивши клас Applet або JApplet.

Лістинг 17.3. Реалізація інтерфейса Runnable

```

class TwoThreads3 implements Runnable{
private String msg;
TwoThreads3(String s){ msg = s; }
public void run(){
forint i = 0; i < 20; i++){
try{
Thread.sleep(100);
}catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of thread.");
}
public static void main (String[] args){
new Thread(new TwoThreads3("HIP"), "Thread 1").start ();
new Thread(new TwoThreads3("hop"), "Thread 2").start ();
System.out.println();
}
}

```

Частіше всього в новому підпроцесі задаються нескінчені дії, виконувані на фоні основних дій: програється музика, на екрані крутиться анімований логотип фірми,

біжить рекламний рядок. Для реалізації такого підпроцеса в методі run() задається нескінчений цикл, зупинюваний після того, як об'єкт-підпроцес отримає значення null. В лістингі 17.4 показано четвертий варіант тієї ж самої програми, в якій метод run() виконується до тих пір, доки поточний об'єкт-підпроцес th співпадає з об'єктом go, запустившим поточний підпроцес. Для переривання його виконання передбачений метод stop(), до якого звертається головний підпроцес. Ця стандартна конструкція, рекомендована документацією J2SDK. Головний підпроцес в даному прикладі тільки створює об'єкти-підпроцеси, чекає одну секунду і зупиняє їх.

Лістинг 17.4. Зупинка роботи підпроцесів

```
class TwoThreadsS implements Runnable{
private String msg;
private Thread go;
TwoThreadsS(String s){
msg = s;
go = new Thread(this);
go.start();
}
public void run(){
Thread th = Thread.currentThread();
while(go == th){
try{
Thread.sleep(100);
} catch(InterruptedException ie){}
System.out.print(msg + " ");
}
System.out.println("End of thread.");
}
public void stop(){ go = null; }
public static void main(String[] args){
TwoThreadsS th1 = new TwoThreadsS("HIP");
TwoThreadsS th2 = new TwoThreadsS("hop");
try{
Thread.sleep(1000);
} catch(InterruptedException ie){}
th1.stop(); th2.stop();
System.out.println();
}
}
```

Приведемо простий приклад. Метод run() в лістингі 17.5 виводить рядок "Hello, World!" із затримкою в 1 секунду між словами. Цей метод виконується двома підпроцесами, працюючими з одним об'єктом th. Програма виконується два рази. Перший раз метод run() не синхронізований, другий раз синхронізований, його заголовок показано в лістингі 17.4 як коментар.

Лістинг 17.5. Синхронізація методу

```
class TwoThreads4 implements Runnable{
public void run(){
// synchronized public void run(){
System.out.print("Hello, ");
try{
Thread.sleep(1000);
} catch(InterruptedException ie){}
System.out.println("World!");
}
public static void main(String[] args){
TwoThreads4 th = new TwoThreads4();
new Thread(th).start();
new Thread(th).start();
}
}
```

Звернемося знову до схеми "постачальник-споживач", уже використану в уроці 15. Один підпроцес, постачальник, робить обчислення, другий, споживач, очікує результати цих обчислень і використовує їх в міру поступання. Підпроцеси передають інформацію через спільний екземпляр st класу store. Робота цих підпроцесів повинна бути узгоджена. Споживач зобовязаний чекати, доки постачальник не занесе результат обчислення в об'єкт st, а постачальник повинен чекати, доки споживач не візьме цей результат. Для простоти постачальник просто заносить в спільний об'єкт класу store цілі числа, а споживач лише забирає їх. В лістингі 17.6 клас store не забезпечує узгодженості отримання і видачі інформацію. Результат роботи показаний на рис. 17.4.

Лістинг 17.6. Неузгоджені підпроцеси

```
class Store{
private int inform;
synchronized public int getInform(){ return inform; }
synchronized public void setInform(int n){ inform = n; }
}
```

```

class Producer implements Runnable{
private Store st;
private Thread go;
Producer(Store st){
this.st = st;
go = new Thread(this);
go.start();
}
public void run(){
int n = 0;
Thread th = Thread.currentThread();
while(go == th){
st.setInform(n);
System.out.print("Put: " + n + " ");
n++;
}
}
public void stop(){ go = null;
}
}
class Consumer implements Runnable{
private Store st;
private Thread go;
Consumer(Store st){
this.st = st;
go = new Thread(this);
go.start();
}
public void run(){
Thread th = Thread.currentThread();
while(go == th) System.out.println("Got: " + st.getInform());
}
public void stop(){ go = null; }
}
class ProdCons{
public static void main(String[] args){
Store st = new Store();
Producer p = new Producer(st);
Consumer c = new Consumer(st);
try{
Thread.sleep(30);
} catch(InterruptedException ie){}
}

```

```

p.stop(); c.stop();
}
}
}

```

В лістингі 17.7 в клас store внесено логічне поле ready, що відмічає процес отримання і видачі інформації. Коли нова порція інформації отримана від постачальника Producer, в полі ready заноситься значення true, отримувач consumer може забирати цю порцію інформації. Після видачі інформації змінна ready становиться рівною false. Але цього мало. Те, що отримувач може забрати продукт, не означає, що він дійсно забере його. Тому в кінці методу setinform() отримувач сповіщається про поступанні продукту методом notify(). Поки поле ready не прийме потрібне значення, підпроцес переводиться в "залу очікування" методом wait(). Результат роботи програми з обновленим класом store показаний на рис. 17.5.

Лістинг 17.7. Узгодження отримання і видачі інформації

```

class Store{
private int inform = -1;
private boolean ready;
synchronized public int getInform(){
try{
if (! ready) wait();
ready = false;
return inform;
} catch(InterruptedException ie){
} finally{
notify();
}
return -1;
}
synchronized public void setInform(int n){
if (ready)
try{
wait();
} catch(InterruptedException ie){}
inform = n;
ready = true;
notify();
}
}

```


Урок 18

Потоки введення/виведення

- Консольне введення/виведення
- Файлове введення/виведення
- Отримання властистей файла
- Буферизоване введення/виведення
- Потік простих типів Java
- Кодування UTF-8
- Прямий доступ до файлу
- Канали обміну інформацією
- Серіалізація обєктів
- Друк в Java 2
- Друк засобами Java 2D
- Друк файлу
- Друк сторінок з різними параметрами

18.1. Введення/Виведення в Java

Програми, написані нами в попередніх уроках, сприймали інформацію тільки із параметрів командного рядка і графічних компонентів, а результати виводили на консоль або в графічні компоненти. Однаке в багатьох випадках треба виводити результати на принтер, у файл, базу даних або передавати по мережі. Вихідні дані теж часто приходиться завантажувати із файла, бази даних або із мережі. Для того щоб абстрагуватись від особливостей конкретних пристроїв **введення/виведення**, в **Java** використовується поняття **потоку (stream)**. Вважається, що в програму іде **вхідний поток (input stream)** символів **Unicode** або просто байтів, що сприймається в програмі методами **read()**. Із програми методами **write()** або **print ()**, **println()** виводиться **вихідний потік (output stream)** символів або байтів. При цьому не має значення, куди направлений потік: на консоль, на принтер, у файл або в мережу, методи **write()** і **print()** нічого про це не знають.

Можна уявити собі потік як трубу, по якій в одному напряму послідовно "течуть" символи або байти, один за другим. Методи **read()** , **write()** , **print()**, **println()** взаємодіють з одним кінцем труби, другий кінець зеднується з джерелом або приймачем даних конструкторами класів, в яких реалізовані ці методи. Звичайно, повне ігнорування особливостей пристроїв **введення/виведення** сильно сповільнює передачу інформації. Тому в **Java** все-таки виділяється файлове **введення/виведення**, **виведення** на друк, мережевий потік.

Три потоки визначені в класі **System** статичними полями **in**, **out** і **err**. Їх можна використовувати без всяких додаткових визначень, що ми весь час і робили. Вони називаються відповідно **стандартним введенням (stdin)**, **стандартним виведенням (stdout)** і **стандартним виведенням повідомлень (stderr)**. Ці стандартні потоки можуть бути зєднані з різними конкретними присторями **введення/виведення**. Потоки **out** і **err** — це екземпляри класу **Printstream**, організуючого вихідний потік байтів. Ці екземпляри виводять інформацію на консоль методами **print()**, **println()** і **write()**, яких в класі **Printstream** мається близько двадцяти для різних типів аргументів.

Потік **err** призначений для **виведення** системних повідомлень програми: трасування, повідомлень про помилки або, просто, про виконання якихось етапів програми. Такі дані звичайно заносяться в спеціальні журнали, **log**-файли, а не виводяться на консоль. В **Java** є засоби перепризначення потоку, наприклад, з консолі у файл.

Поток **in** — це екземпляр класу **InputStream**. Він призначений на клавіатурне **введення** з консолі методами **read()**. Клас **InputStream** абстрактний, тому реально використовується якийсь із його підкласів.

Поняття потоку виявилося настільки зручним і полегшуючим програмування **введення/виведення**, що в **Java** передбачена можливість створення потоків, направляючих символи або байти не на зовнішній пристрій, а в масив або із масиву, обто звязуючих програму з областю оперативної пам'яті. Більше того, можна створити потік, звязаний з рядком типу **String**, що знаходиться, знову-таки, в оперативній пам'яті.

Крім того, можна створити *канал* ([pipe](#)) обміну інформацією між підпроцесами.

Ще один вид потоку — потік байтів, складаючих об'єкт [Java](#). Його можна направити у файл або передати по мережі, потім відновити в оперативній пам'яті. Ця операція називається *серіалізацією* ([serialization](#)) об'єктів.

Методи організації потоків зібрані в класи пакета [java.io](#). Крім класів, організуючих потік, в пакет [java.io](#) входять класи з методами перетворення потоку, наприклад, можна перетворити потік байтів, утворюючих цілі числа, в потік цих чисел. Ще одна можливість, представлена класами пакета [java.io](#), — злити декілька потоків в один потік.

Отже, в [Java](#) єсть цілих чотири ієрархії класів для створення, перетворення і злиття потоків. На чолі ієрархії чотири класи, безпосередньо розширюючих клас [Object](#):

- [Reader](#) — абстрактний клас, в якому зібрані самі загальні методи символьного введення;
- [Writer](#) — абстрактний клас, в якому зібрані самі загальні методи символьного виведення;
- [InputStream](#) — абстрактний клас з загальними методами байтового введення;
- [OutputStream](#) — абстрактний клас з загальними методами байтового виведення.

Класи вхідних потоків [Reader](#) і [InputStream](#) визначають по три методи введення:

- [read \(\)](#) — повертає один символ або байт, взятий із вхідного потоку, вигляді цілого значення типу [int](#); якщо потік уже закінчився, повертає [-1](#);
- [read \(char\[\] buf\)](#) — заповняє заздалегідь визначений масив [buf](#) символами із вхідного потоку; в класі [InputStream](#) масив типу [byte\[\]](#) і заповнюється він байтами; метод повертає фактичне число взятих із потоку елементів або [-1](#), якщо потік уже закінчився;
- [read \(char\[\] buf, int offset, int len\)](#) — заповнює частину символьного або байтового масиву [buf](#), починаючи з індекса [offset](#), число взятих із потоку елементів рівне [len](#); метод повертає фактичне число взятих із потока елементів або [-1](#).

Ці методи викидають [IOException](#), якщо відбулася помилка [введення/виведення](#). Четвертий метод [skip \(long n\)](#) "промотує" потік з поточної позиції на [n](#) символів або байтів вперед. Ці елементи потоку не вводяться методами [read\(\)](#). Метод повертає реальне число пропущених елементів, яке може відрізнятися від [n](#), наприклад потік може закінчиться. Поточний елемент потоку можна помітити методом [mark \(int n\)](#), а потім повернутися до поміченого елементу методом [reset\(\)](#), але не більше ніж через [n](#) елементів. Не всі підкласи реалізують ці методи, тому перед розстановкою поміток належить звернутися до логічного методу [markSupported\(\)](#), який повертає [true](#), якщо реалізовані методи розстановки і повернення до поміток.

Класи вихідних потоків [Writer](#) і [OutputStream](#) визначають по три майже однакові методи введення:

- [write \(char\[\] buf\)](#) — виводить масив у вихідний потік, в класі [OutputStream](#) масив має тип [byte\[\]](#);
- [write \(char\[\] buf, int offset, int len\)](#) — виводить [len](#) елементів масиву [buf](#), починаючи з злемента із індексом [offset](#);
- [write \(int elem\)](#) в класі [Writer](#) - виводить 16, а в класі [OutputStream](#) 8 молодших бітів аргумента [elem](#) у вихідний потік.

В класі [Writer](#) є ще два методи:

- [write \(String s\)](#) — виводить рядок [s](#) у вихідний потік;
- [write \(String s, int offset, int len\)](#) — виводить [len](#) символів рядка [s](#), починаючи із символа з номером [offset](#).

Багато підкласів класів [Writer](#) і [OutputStream](#) здійснюють буферизацію 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 із програмами і виведення потоку, але часто треба вивести інформацію в потік ще до заповнення буфера. Для цього передбачений метод [flush\(\)](#). Даний метод зразу ж виводить весь вміст буфера в потік. Нарешті, по закінченню роботи з потоком його необхідно закрити методом [closed](#). Класи, що входять в ієрархії потоків [введення/виведення](#),

показані на рис. 18.1 и 18.2.

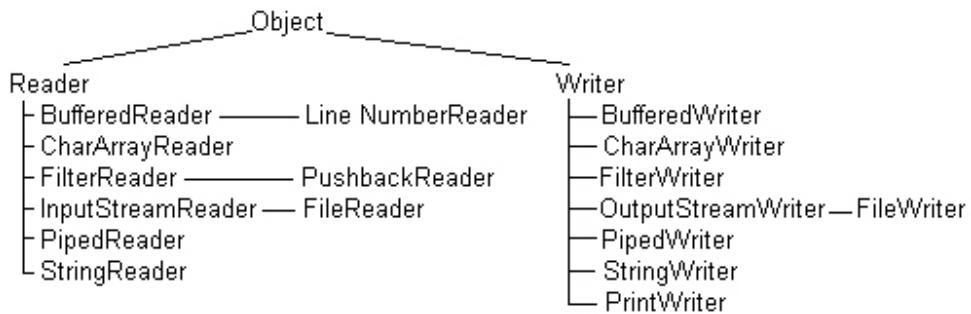


Рис. 18.1. Ієархія символьних потоків

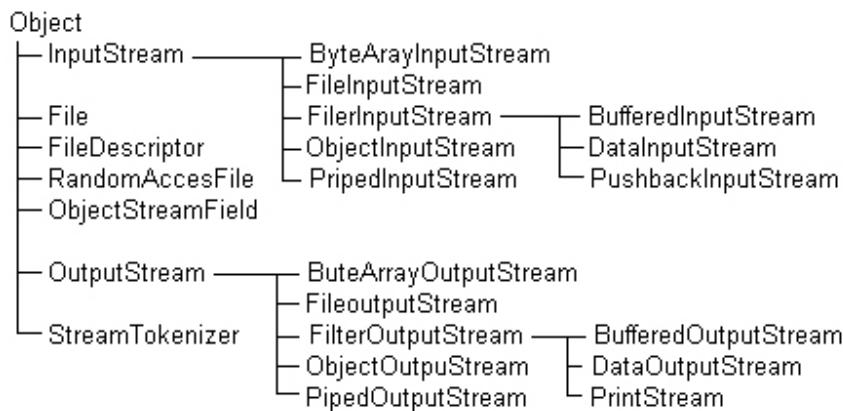


Рис. 18.2. Класи байтових потоків

Всі класи пакета `java.io` можна розділити на дві групи: класи, що створюють потік (**data sink**), і класи, що управлюють потоком (**data processing**). Класи, що створюють потоки, в свою чергу, можна розділити на п'ять груп:

- класи, що створюють потоки, звязані з файлами:
`FileReader`, `FileInputStream`, `FileWriter`, `OutputStream`, `RandomAccessFile`
- класи, що створюють потоки, звязані з масивами:
`CharArrayReader`, `ByteArrayInputStream`, `CharArrayWriter`, `ByteArrayOutputStream`
- клас, що створюють канали обміну інформацією між підпроцесами:
`PipedReader`, `PipedInputStream`, `PipedWriter`, `PipedOutputStream`
- класи, створюючі символьні потоки, звязані з рядком:
`StringReader`, `StringWriter`
- класи, створюючі байтові потоки із обєктів Java:
`ObjectInputStream`, `ObjectOutputStream`

Зліва перечислені класи символьних потоків, справа — класи байтових потоків. Класи, управлюючі потоком, отримують в своїх конструкторах уже наявний потік і створюють новий, перетворений потік. Можна уявляти їх собі як "перехідне кільце", після якого йде труба іншого діаметру. Чотири класи створені спеціально для перетворення потоків:

`FilterReader`, `FilterInputStream`, `FilterWriter`, `FilterOutputStream`

Самі по собі ці класи не мають ніякої користі — вони виконують totожне перетворення. Їх належить розширювати, перевизначаючи методи `введення/виведення`. Але для байтових фільтрів єсть корисні розширення, яким відповідають деякі символьні класи. Перечислимо їх.

- Чотири класи виконують буферизоване `введення/виведення`: `BufferedReader`, `BufferedInputStream`,

BufferedWriter, BufferedOutputStream

- Два класи перетворюють потік байтів, утворюючих вісім простих типів Java, в ці самі типи: [DataInputStream](#), [DataOutputStream](#)
- Два класи містять методи, дозволяючі повернути декілька символів або байтів у вхідний потік: [PushbackReader](#), [PushbackInputStream](#)
- Два класи звязані з виведенням на рядкові пристрої — екран дисплея, принтер: [PrintWriter](#), [PrintStream](#)
- Два класи звязують байтовий і символьний потоки:
 - ◆ [InputStreamReader](#) — перетворює вхідний байтовий потік у символьний потік;
 - ◆ [OutputStreamWriter](#) — перетворює вихідний символьний потік в байтовий потік.
- Клас [streamTokenizer](#) дозволяє розібрати вхідний символьний потік на окремі елементи (tokens) подібно до того, як клас [StringTokenizer](#), розглянутий нами в уроці 5, розбирає рядок.
- Із управлюючих класів виділяється клас [SequenceInputStream](#), зливаючий декілька потоків, заданих в конструкторі, в один потік, і клас [LineNumberReader](#), "уміючий" читати вихідний символьний потік порядково. Рядки в потоці розділяються символами '\n' і/або '\r'.

Цей огляд класів [введення/виведення](#) трохи проясняє ситуацію, але не пояснює, як їх використовувати. Перейдемо до розгляду реальних ситуацій.

18.2. Консольне введення/виведення

Для виведення на консоль ми завжди використовували метод [println\(\)](#) класу [PrintStream](#), ніколи не визначаючи екземпляри цього класу. Ми просто використовували статичне поле [out](#) класу [System](#), яке являється об'єктом класу [PrintStream](#). Виконуюча система Java звязує це поле з консольлю. Між іншим, якщо вам набридло писати [System.out.println\(\)](#), то ви можете визначити нове посилання на [System.out](#), наприклад:

```
PrintStream pr = System.out;
```

і писати просто [pr.println\(\)](#). Консоль являється байтовим пристроєм, і символи [Unicode](#) перед виведенням на консоль повинні бути перетворені в байти. Для символів [Latin 1](#) з кодами '\u0000' — '\u00FF' при цьому просто відкидається нульовий старший байт і виводяться байти '0x00' — '0xFF'. Для кодів кирилиці, які лежать в діапазоні '\u0400' — '\u04FF' кодування [Unicode](#), і інших національних алфавітів відбувається перетворення по кодовій таблиці, відповідній установлений на комп'ютері локалі. Ми обговорювали це в уроці 5.

Труднощі з відображенням кирилиці виникають, якщо виведення на консоль відбувається в кодуванні, відмінному від локалі. Саме так відбувається в русифікованих версіях [MS Windows NT/2000](#). Звичайно в них установлюється локаль з кодовою сторінкою [CP1251](#), а виведення на консоль відбувається в кодуванні [CP866](#). В цьому випадку треба замінити [PrintStream](#), який не може працювати з символьним потоком, на [PrintWriter](#) і "вставити перехідне кільце" між потоком символів [Unicode](#) і потоком байтів [System.out](#), що виводиться на консоль, у вигляді об'єкта класу [OutputStreamWriter](#). В конструкторі цього об'єкта належить указати потрібне кодування, в даному випадку, [CP866](#). Все це можна зробити одним оператором:

```
PrintWriter pw = new PrintWriter(new OutputStreamWriter(System.out, "Cp866"), true);
```

Клас [PrintStream](#) буферизує вихідний потік. Другий аргумент [true](#) його конструктора викликає примусове зкидання вмісту буфера у вихідний потік після кожного виконання методу [println\(\)](#). Але після [print\(\)](#) буфер не спорожнюється! Для зкидання буфера після кожного [print\(\)](#) треба писати [flush\(\)](#), як це зроблено в лістингі 18.2.

Зауваження

Методи класу [PrintWriter](#) по замовчуванню не очищають буфер, а метод [print\(\)](#) не очищає його в будь-якому випадку. Для очистки буфера використовуй метод [flush\(\)](#). Після цього можна виводити будь-який текст методами класа [PrintWriter](#), які просто дублюють методи класу [PrintStream](#), і писати, наприклад, [pw.println\("Це український текст"\)](#); як показано в лістингі 18.1 і на рис. 18.3. Слід відмітити, що конструктор класу [PrintWriter](#), в якому заданий байтовий потік, завжди неявно створює об'єкт класу [OutputStreamWriter](#)

з локальним кодуванням для перетворення байтового потоку в символний потік.

Введення з консолі відбувається методами `read()` класу `InputStream` за допомогою статичного поля `in` класу `System`. З консолі йде потік байтів, отриманих із `scan`-кодів клавіатури. Ці байти повинні бути перетворені в символи `Unicode` такими ж кодовими таблицями, як і при виведенні на консоль. Перетворення йде по тій же схемі — для правильного введення кирилиці зручніше всього визначити екземпляр класу `BufferedReader`, використовуючи в якості "перехідного кільця" об'єкт класу `InputStreamReader`:

```
BufferedReader br = new BufferedReader( new InputStreamReader(System.in, "Cp866"));
```

Клас `BufferedReader` перевизначає три методи `read()` свого суперкласу `Reader`. Крім того, він містить метод `readLine()`. Метод `readLine()` повертає рядок типу `String`, що містить символи вхідного потоку, починаючи з поточного, і закінчуєчи символом `\n` і/або `\r`. Ці символи-розділювачі не входять в повернений рядок. Якщо у вхідному потоці немає символів, то повертається `null`. В листині 18.1 приведена програма, ілюструюча перечислені методи консольного **введення/виведення**. На рис. 18.3 показано виведення цієї програми.

Лістинг 18.1. Консольне введення/виведення

```
import java.io.*;
class PrWr{
public static void main(String[] args){
try{
BufferedReader br = new BufferedReader(new InputStreamReader(System.in, "Cp866"));
PrintWriter pw = new PrintWriter(
new OutputStreamWriter(System.out, "Cp866"), true);
String s = "Це рядок з українським текстом";
System.out.println("System.out puts: " + s);
pw.println("PrintWriter puts: " + s);
int c = 0;
pw.println("Посимвольне введення:");
while((c = br.read()) != -1)
pw.println((char)c);
pw.println("Порядкове введення:");
do{
s = br.readLine();
pw.println(s);
}while(!s.equals("q"));
} catch(Exception e){
System.out.println(e);
}
}
}
```

Пояснимо рис. 18.3. Перший рядок виводиться потоком `System.out`. Як бачимо, кирилиця виводиться неправильно. Наступний рядок попередньо перетворений в потік байтів, записаних в кодуванні `CP866`. Потім, після тексту "Посимвольне введення:" з консолі вводяться символи "Україна" і натискується клавіша `<Enter>`. Кожний введений символ відображається на екрані — операційна система працює в режимі так званого "еха". Фактичне введення з консолі починається тільки після натискання клавіші `<Enter>`, тому що клавіатурне введення буферизується операційною системою. Символи зразу ж після введення відображаються по одному в рядку. Зверніть увагу на два порожніх рядки після літери `a`. Це виведені символи `\n` і `\r`, які попали у вхідний потік при натисканні клавіші `<Enter>`. У них немає ніякого графічного накреслення (`glyph`). Потім натиснута комбінація клавіш `<Ctrl>+<Z>`. Вона відображається на консоль як `"^Z"` і означає закінчення клавіатурного введення, завершуючи цикл введення символів. Коди цих клавіш уже не попадають у вхідний потік. Далі, після тексту "Порядкове введення:" з клавіатури набирається рядок "Це рядок" і, вслід за натисканням клавіші `<Enter>`, заноситься в рядок `s`. Потім рядок `s` виводиться знову на консоль. Для закінчення роботи набираємо `q` і натискуємо клавішу `<Enter>`.

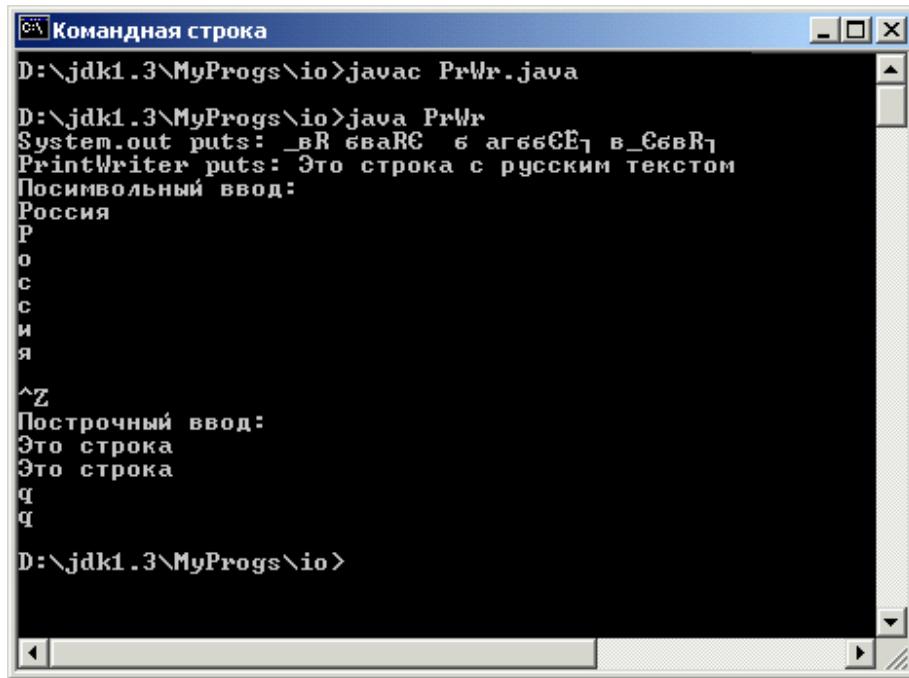


Рис. 18.3. Консольне введення/виведення

18.3. Файлове введення/виведення

Оскільки файли в більшості сучасних операційних систем розуміються як послідовність байтів, для файлового **введення/виведення** створюються байтові потоки за допомогою класів `FileInputStream` і `FileOutputStream`. Це особливо зручно для бінарних файлів, що зберігають байт-коди, архіви, зображення, звук. Але дуже багато файлів містять тексти, складені із символів. Незважаючи на те, що символи можуть зберігатися в кодуванні `Unicode`, ці тексти частіше всього записані байтових кодуваннях. Тому і для текстових файлів можна використовувати байтові потоки. В такому випадку з боку програми прийдеться організовувати перетворення байтів у символи і навпаки.

Щоб полегшити це перетворення, в пакет `java.io` введені класи `FileReader` в `FileWriter`. Вони організовують перетворення потоку: із сторони програми потоки символьні, із сторони файла — байтові. Це відбувається тому, що дані класи розширяють класи `InputStreamReader` і `OutputStreamWriter`, відповідно, значить, містять "перехідне кільце" всередині себе. Незважаючи на відмінність потоків, використання класів файлового **введення/виведення** дуже схоже. В конструкторах всіх чотирьох файлових потоків задається ім'я файла у вигляді рядка типу `string` або посилка на об'єкт класу `File`. Конструктори не тільки створюють об'єкт, але і відшукують файл і відкривають його. Наприклад:

```
FileInputStream fis = new FileInputStream("PrWr.java");
FileReader fr = new FileReader("D:\\jdk1.3\\src\\PrWr.java");
```

При невдачі викидається виключення класу `FileNotFoundException`, але конструктор класу `FileWriter` викидає більш загальне виключення `IOException`. Після відкриття вихідного потоку типу `FileWriter` або `FileOutputStream` вміст файла, якщо він не був порожнім, зтирається. Для того щоб можна було робити запис в кінець файла, і в тому і в іншому класі передбачений конструктор з двома аргументами. Якщо другий аргумент рівний `true`, то відбувається дозапис в кінець файла, якщо `false`, то файл заповнюється новою інформацією. Наприклад:

```
FileWriter fw = new FileWriter("ch18.txt", true);
FileOutputStream fos = new FileOutputStream("D:\\samples\\newfile.txt");
```

Увага

Вміст файлу, відкритого для запису конструктором з одним аргументом, стирається. Зразу після виконання конструктора можна читати файл:

`fis.read(); fr.read();` або записувати в нього: `fos.write((char)c); fw.write((char)c);`

По закінченню роботи з файлом потік належить закрити методом `close()`. Перетворення потоків у класах `FileReader` і `FileWriter` виконується по кодових таблицях установленої на комп'ютері локалі. Для правильного введення кирилиці треба застосувати `FileReader`, а не `FileInputStream`. Якщо файл містить текст в кодуванні, відмінний від локального кодування, то прийдеться вставляти "перехідне кільце" вручну, як це робилось для консолі, наприклад:

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

Байтовий потік `fis` визначений вище.

18.4. Отримання властивостей файла

В конструкторах класів файлового [введення/виведення](#), описаних в попередньому розділі, указувалось ім'я файла у вигляді рядка. При цьому залишалось невідомим, чи існує файл, чи дозволений до нього доступ, яка довжина файла. Отриати такі дані можна від попередньо створеного екземпляру класу `File`, що містить дані про файл. В конструкторі цього класу

`File(String filename)`

указується шлях до файла або каталогу, записаний по правилах операційної системи. В [UNIX](#) імена каталогів розділяються похилою рискою `/`, в [MS Windows](#) — зворотною похилою рискою `\`, в [Apple Macintosh](#) — двокрапкою `:`. Цей символ міститься в системній властивості `file.separator` (див. рис. 6.2). Шляху до файла передує префікс. В [UNIX](#) це похила риска, в [MS Windows](#) — літера розділу диска, двокрапка і зворотна похила риска. Якщо префікса немає, то шлях вважається відносним і до нього додається шлях до поточного каталогу, який зберігається в системній властивості `user.dir`. Конструктор не перевіряє, чи існує файл з таким іменем, тому після створення об'єкта слід це перевірити логічним методом `exists()`.

Класс `File` містить близько сорока методів, що дозволяють узнати різні властивості файла або каталога. Перш за все, логічними методами `isFile()`, `isDirectory()` можна вияснити, чи являється шлях, указаний в конструкторі, шляхом до файла або каталога. Для каталога можна отримати його зміст — список імен файлів і підкаталогів — методом `list()`, повертаючим масив рядків `String[]`. Можна отримати такий же список у вигляді масиву об'єктів класу `File[]` методом `listFiles()`. Можна вибрати із списку тільки деякі файли, реалізувавши інтерфейс `FileNameFilter` і звернувшись до методу

`list(FileNameFilter filter).`

Якщо каталог з указаним в конструкторі шляхом не існує, його можна створити логічним методом `mkdir()`. Цей метод повертає `true`, якщо каталог удалось створити. Логічний метод `mkdir()` створює ще і всі неіснуючі каталоги, указані в шляху. Порожній каталог видаляється методом `delete()`. Для файла 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()` показують права доступу до файла. Файл можна переіменувати логічним методом `renameTo(File newMame)` або видалити логічним методом `delete()`. Ці методи повертають `true`, якщо операція пройшла успішно. Якщо файл з указаним в конструкторі шляхом не існує, його можно створити логічним методом `createNewFile()`, що повертає `true`, якщо файл не існував, і його удалось створити, і `false`, якщо файл уже існував.

Статичними методами

`createTempFile(String prefix, String suffix, File tmpDir)`
`createTempFile(String prefix, String suffix)`

можна створити тимчасовий файл з іменем `prefix` і розширенням `suffix` в каталозі `tmpDir` або каталозі, указаному в системній властивості `java.io.tmpdir` (см. рис. 6.2). Імя `prefix` повинно містити не менше трьох символів. Якщо `suffix = null`, то файл одержить суфікс `.tmp`. Перечислені методи повертають посилку типу `File` на створений файл. Якщо звернутися до методу `deleteOnExit()`, то по завершенні роботи `JVM` тимчасовий файл буде знищений.

Декілька методів `getxxx` повертають імя файла, імя каталога і інші дані про шлях до файлу. Ці методи корисні в тих випадках, коли посилка на об'єкт класу `File` повертається іншими методами і потрібні дані про файл. Нарешті, метод `toURL()` повертає шлях до файлу у формі `URL`. В лістингі 18.2 показано приклад використання класу `File`, а на рис. 18.4 — початок виведення цієї програми.

Лістинг 18.2. Визначення властивостей файла і каталога

```
import java.io.*;
class FileTest{
public static void main(String[] args) throws IOException{
PrintWriter pw = new PrintWriter(
new OutputStreamWriter(System.out, "Cp866"), true);
File f = new File("FileTest.Java");
pw.println();
pw.println("Файл \"" + f.getName() + "\" " +
(f.exists()?"":"не ") + "существует");
pw.println("Вы " + (f.canRead()?"":"не ") + "можете читать файл");
pw.println("Вы " + (f.canWrite()?"":"не ") +
"можете записывать в файл");
pw.println("Длина файла " + f.length() + " б");
pw.println();
File d = new File("D:\\jdk1.3\\MyProgs");
pw.println("Содержимое каталога:");
if (d.exists() && d.isDirectory()) {
String[] s = d.list();
for (int i = 0; i < s.length; i++)
pw.println(s[i]);
}
}
}
```

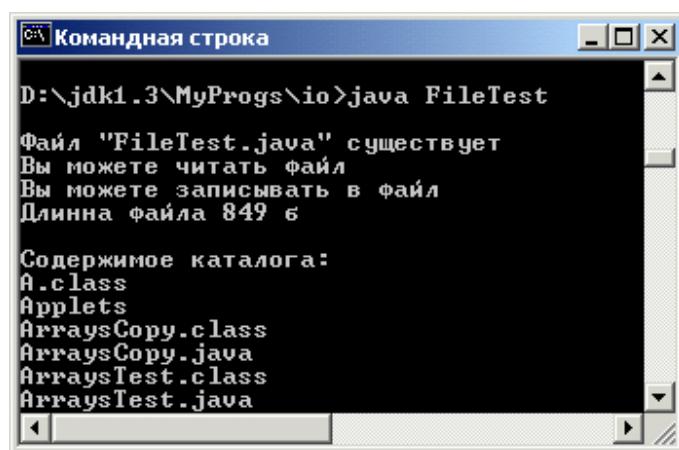


Рис. 18.4. Властивості файла і початок виведення каталога

18.5. Буферизоване введення/виведення

Операції `введення/виведення` в порівнянні з операціями в оперативній пам'яті виконуються дуже повільно. Для компенсації в оперативній пам'яті виділяється деяка проміжна область — буфер, в який поступово

накопичується інформація. Коли буфер заповнений, його вміст швидко переноситься процесором, буфер очищається і знову заповнюється інформацією. Життєвий приклад буфера — поштова скринька, в якій накопичуються листи. Ми кидаємо в нього листа і йдемо в своїх справах, не очікуючи приїзду поштової машини. Поштова машина періодично очищає поштову скриньку, переносячи зразу велику кількість листів. Уявіть собі місто, в якому немає поштових скриньок, і гатові людей з листами в руках очікує приїзд поштової машини.

Класи файлового [введення/виведення](#) не займаються буферизацією. Для цієї мети є чотири спеціальні класи [BufferedXxx](#), перечислені вище. Вони приєднуються до потоків [введення/виведення](#) як "перехідне кільце", наприклад:

```
BufferedReader br = new BufferedReader(isr);
BufferedWriter bw = new BufferedWriter(fw);
```

Потоки [isr](#) і [fw](#) визначені вище. Програма лістинга 18.3 читає текстовий файл, написаний в кодуванні [CP866](#), і записує його вміст у файл в кодуванні [KOI8_R](#). При читанні і записі застосовується буферизація. Ім'я вихідного файла задається в командному рядку параметром [args\[0\]](#), ім'я копії — параметром [args\[1\]](#).

Лістинг 18.3. Буферизоване файлове введення/виведення

```
import java.io.*;
class DOStoUNIX{
public static void main(String[] args) throws IOException{
if (args.length != 2){
System.err.println("Usage: DOStoUNIX Cp866file KOI8_Rfile");
System.exit(0);
}
BufferedReader br = new BufferedReader(
new InputStreamReader(
new FileInputStream(args[0]), "Cp866"));
BufferedWriter bw = new BufferedWriter(
new OutputStreamWriter(
new FileOutputStream(args[1]), "KOI8_R"));
int c = 0;
while ((c = br.read()) != -1)
bw.write((char)c);
br.close(); bw.close();
System.out.println("The job's finished.");
}
}
```

18.6. Потік простих типів Java

Клас [DataOutputStream](#) дозволяє записати дані простих типів [Java](#) у вихідний потік байтів методами [writeBoolean\(boolean b\)](#), [writeByte\(int b\)](#), [writeShort\(int h\)](#), [writeChar\(int c\)](#), [writeln\(int n\)](#), [writeLong\(long l\)](#), [writeFloat\(float f\)](#), [writeDouble\(double d\)](#). Крім того, метод [writeBytes\(String s\)](#) записує кожний символ рядка [s](#) в один байт, відкидаючи старший байт кодування кожного символу [Unicode](#), а метод [writeChar\(String s\)](#) записує кожний символ рядка [s](#) в два байти, перший байт — старший байт кодування [Unicode](#), так же, як це робить метод [writeChar\(\)](#).

Ще один метод [writeUTF\(String s\)](#) записує рядок [s](#) у вихідний потік в кодуванні [UTF-8](#). Треба пояснити це кодування.

18.7. Кодування UTF-8

Запис потоку в байтовому кодуванні викликає труднощі з використанням національних символів, запис потоку в [Unicode](#) збільшує довжину потоку в два рази. Кодування [UTF-8 \(Universal Transfer Format\)](#) являється компромісом. Символ в цьому кодуванні записується одним, двома або трьома байтами. Символи [Unicode](#) із діапазону '[\u0000](#)' — '[\u007F](#)', в якому лежить англійський алфавіт, записуються одним

байтом, старший байт просто відкидається. Символи [Unicode](#) із діапазону '\u0080' —'\u07FF', в якому лежать найбільш розповсюджені символи національних алфавітів, записуються двома байтами наступним чином: символ [Unicode](#) з кодуванням 0000xxxxxууууу записується як 110xxxxx10ууууу. Решта символів [Unicode](#) із діапазону '\u0800' —'\UFFFF' записуються трьома байтами по наступному правилу: символ [Unicode](#) з кодуванням xxxхууууууzzzzзззззз записується як 1110xxxхххх10ууууу10zzzzзззззз. Такий дивний спосіб розподілу бітів дозволяє по першим бітам коду узнати, скільки байтів складає код символа, і правильно відрахувати символи в потоці.

Так ось, метод `writeUTF(String s)` спочатку записує в потік в перші два байти потоку довжину рядка `s` в кодуванні [UTF-8](#), а потім символи рядка в цьому кодуванні. Читати цей запис потім треба парним методом `readUTF()` класу [DataInputStream](#). Клас [DataInputStream](#) перетворює вхідний потік байтів типу [InputStream](#), що містить дані простих типів [Java](#), в дані того ж типу. Такий потік, як правило, створюється методами класу [DataOutputStream](#). Дані із цього потоку можна прочитати методами `readBoolean()`, `readByte()`, `readShort()`, `readChar()`, `readInt()`, `readLong()`, `readFloat()`, `readDouble()`, повертаючими дані відповідного типу. Крім того, методи `readUnsignedByte()` і `readUnsignedShort()` повертають ціле типу `int`, в якому старші три або два байти нульові, а молодші один або два байти заповнені байтами із вхідного потоку.

Метод `readUTF()`, двоїстий методу `writeUTF()`, повертає рядок типу `String`, отриманий із потоку, записаного методом `writeUTF()`. Ще один, статичний, метод `readUTF(DataInputStream in)` робить те ж саме із вхідним потоком `in`, записаним в кодуванні [UTF-8](#). Цей метод можна застосовувати, не створюючи об'єкт класу [DataInputStream](#). Програма в лістингі 18.4 записує у файл `fib.txt` числа Фібоначчі, а потім читає цей файл і виводить його зміст на консоль. Для контролю записувані у файл числа теж виводяться на консоль. На рис. 18.5 показано виведення цієї програми.

Лістинг 18.4. Введення/виведення даних

```
import java.io.*;
class DataPrWr{
public static void main(String[] args) throws IOException{
DataOutputStream dos = new DataOutputStream (
new FileOutputStream("fib.txt"));
int a = 1, b = 1, c = 1;
for (int k = 0; k < 40; k++){
System.out.print(b + " ");
dos.writeInt(b);
a = b; b = c; c = a + b;
}
dos.close();
System.out.println("\n");
DataInputStream dis = new DataInputStream (
new FileInputStream("fib.txt"));
while(true)
try{
a = dis.readInt();
System.out.print(a + " > ");
} catch(IOException e){
dis.close();
System.out.println("End of file");
System.exit(0);
}
}
}
```

Зверніть увагу на те, що спроба читання за кінцем файла викидає виключення класу [IOException](#), його обробка заключається в закритті файла і закінченні програми.

18.8. Прямий доступ до файла

Якщо необхідно інтенсивно працювати з файлом, записуючи в нього дані різних типів [Java](#), змінюючи їх,

відшукючи і читаючи потрібні інформацію, то краще всього скористатися методами класу [RandomAccessFile](#). В конструкторах цього класу

[RandomAccessFile\(File file, String mode\)](#)
[RandomAccessFile\(String fileName, String mode\)](#)

другим аргументом mode задається режим відкриття файла. Це може бути рядок "r" — відкриття файла тільки для читання, або "rw" — відкриття файла для читання і запису. Цей клас зібрав всі корисні методи роботи з файлом. Він містить всі методи класів [DataInputStream](#) і [DataOutputStream](#), крім того, дозволяє прочитати зразу цілий рядок методом [readln\(\)](#) і відшукати потрібні дані у файлі. Байти файла нумеруються, починаючи з 0, подібно елементам масиву. Файл має неявний показчик (file pointer) поточної позиції. Читання і запис відбувається, починаючи з поточної позиції файла. При відкритті файла конструктором показчик стоять на початку файла, в позиції 0. Поточну позицію можна узнати методом [getFilePointer\(\)](#). Кожне читання або запис переміщає показчик на довжину прочитаного або записаного даного. Завжди можна перемістити показчик в нову позицію pos методом [seek \(long pos\)](#). Метод [seek\(0\)](#) переміщає показчик на початок файла. В класі немає методів перетворення символів в байти і назад по кодовим таблицям, тому він не пристосований для роботи з кирилицею.

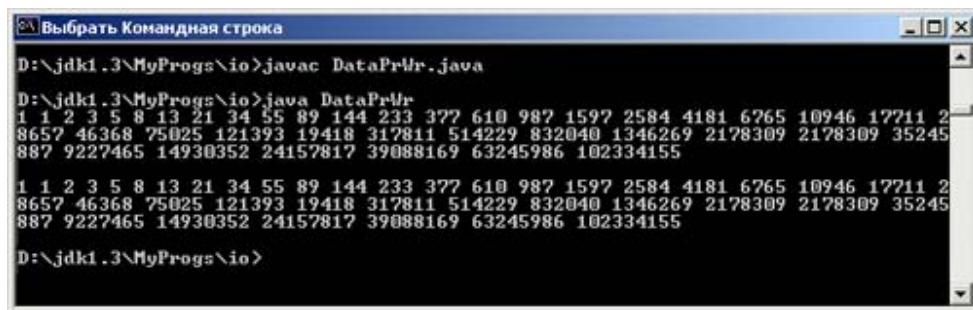


Рис. 18.5. Введення/виведення даних

18.9. Канали обміну інформацією

В попередньому уроці ми бачили, яких зусиль коштує організувати правильний обмін інформацією між підпроцесами. В пакеті [java.io](#) єсть чотири класи [pipedxxx](#), полегшує це завдання. В одному підпроцесі — джерелі інформації — створюється об'єкт класу [PipedWriter](#) або [PipedOutputStream](#), в який записується інформація методами [write\(\)](#) цих класів. В другому підпроцесі — приймачу інформації — формується об'єкт класу [PedReader](#) або [PipedInputStream](#). Він звязується з об'єктом-джерелом за допомогою конструктора або спеціальним методом [connect\(\)](#), і читає інформацію методами [read\(\)](#). Джерело і приймач можна створити і звязати в зворотному порядку.

Так створюється однонаправлений канал (pipe) інформації. На самім ділі це деяка область оперативної пам'яті, до якої організований сумісний доступ двох або більше підпроцесів. Доступ синхронізується, записуючі процеси не можуть завадити читанню. Якщо треба організувати двосторонній обмін інформацією, то створюються два канали. В лістингі 18.5 метод [run\(\)](#) класу [Source](#) генерує інформацію, для простоти просто цілі числа [k](#), і передає їх в канал методом [pw.write \(k\)](#). Метод [run\(\)](#) класу [Target](#) читає інформацію із канала методом [pr.read\(\)](#). Кінці каналу звязуються за допомогою конструктора класу [Target](#). На рис. 18.6 видно послідовність запису і читання інформації.

Лістинг 18.5. Канал обміну інформацією

```
import java.io.*;
class Target extends Thread{
private PedReader pr;
Target(PipedWriter pw){
try{
pr = new PedReader(pw);
}catch( IOException e){
}
```

```

System.err.println("From Target(): " + e);
}
}
PipedReader getStream(){ return pr;}
public void run(){
while(true)
try{
System.out.println("Reading: " + pr.read());
}catch(IOException e){
System.out.println("The job's finished.");
System.exit(0);
}
}
}
class Source extends Thread{
private PipedWriter pw;
Source (){
pw = new PipedWriter();
}
PipedWriter getStream(){ return pw;}
public void run(){
for (int k = 0; k < 10; k++)
try{
pw.write(k);
System.out.println("Writing: " + k);
}catch(Exception e){
System.err.println("From Source.run(): " + e);
}
}
}
class PipedPrWr{
public static void main(String[] args){
Source s = new Source();
Target t = new Target(s.getStream());
s.start();
t.start();
}
}

```

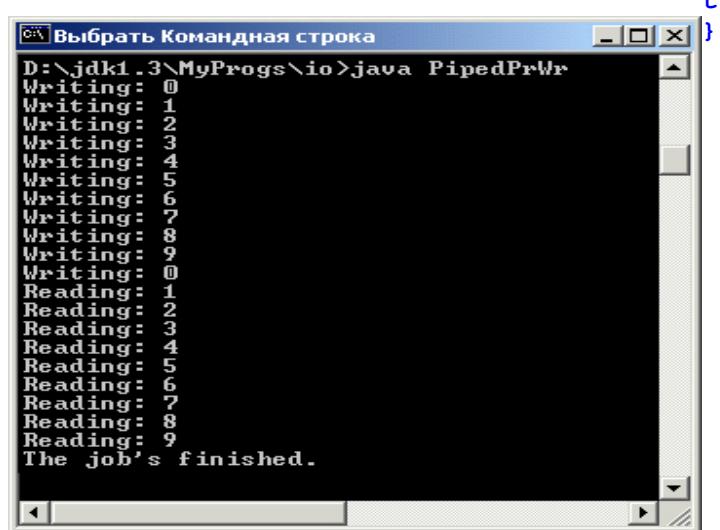


Рис. 18.6. Дані, передавані між підпроцесами

18.10. Серіалізація обєктів

Методи класів [ObjectInputStream](#) і [ObjectOutputStream](#) дозволяють прочитати із вхідного байтового потоку або записати у вихідний байтовий потік дані складних типів — обекти, масиви, рядки — подібно тому, як методи класів [DataInputStream](#) і [DataOutputStream](#) читають і записують дані простих типів. Схожість підсилюється тим, що класи [Objectxxx](#) містить методи як для читання, так і запису простих типів. Між іншим, ці методи призначені не для використання в програмах, а для запису/читання полів об'єктів і елементів масивів.

Процес запису об'єкта у вихідний потік отримав назву *серіалізації* (*serialization*), а читання об'єкта із вхідного потоку і відновлення його в оперативній пам'яті — *десеріалізації* (*deserialization*). Серіалізація об'єкта порушує його безлеку, оскільки зловредний процес може серіалізувати об'єкт в масив, переписати деякі елементи масиву, представляючи *private*- поля об'єкта, забезпечивши собі, наприклад, доступ до секретного файлу, а потім десеріалізувати об'єкт із зміненими полями і здійснювати з ним недопустимі дії. Тому серіалізувати можна не кожний об'єкт, а тільки той, котрий реалізує інтерфейс [Serializable](#). Цей інтерфейс не містить ні полів, ні методів. Реалізувати в ньому нема чого. По суті справи запис

```
class A implements Serializable{...}
```

це тільки помітка, дозволяюча серіалізацію класу [A](#). Як завжди в [Java](#), процес серіалізації максимально автоматизований. Досить створити об'єкт класу [ObjectOutputStream](#), звязавши його з вихідним потоком, і виводить в цей потік об'єкти методом [writeObject\(\)](#):

```
MyClass me = new MyClass("abc", -12, 5.67e-5);
int[] arr = {10, 20, 30};
ObjectOutputStream oos = new ObjectOutputStream(
new FileOutputStream("myobjects.ser"));
oos.writeObject(me);
oos.writeObject(arr);
oos.writeObject("Some string");
oos.writeObject (new Date());
oos.flush();
```

У вихідний потік виводяться всі нестатичні поля об'єкта, незалежно від прав доступу до них, а також дані про клас цього об'єкта, необхідні для його правильного відновлення при десеріалізації. Байт-коди методів класу не серіалізуються. Якщо в об'єкті присутні посилки на інші об'єкти, то вони теж серіалізуються, а в них можуть бути посилки на інші об'єкти, котрі знову-таки серіалізуються, и отримується ціла множина чудернацьки звязаних між собою серіалізованих об'єктів. Метод [writeObject\(\)](#) розпізнає дві посилки на один об'єкт і виводить його у вихідний потік тільки один раз. До того ж, він розпізнає посилки, замкнуті в кільце, і уникне зациклювання.

Всі класи об'єктів, що входять в таку серіалізовану множину, а також всі їх внутрішні класи, повинні реалізувати інтерфейс [Serializable](#), в протилежному випадку буде викинути виключення класу [NotSerializableException](#) і процес серіалізації перерветься. Багато класів [J2SDK](#) реалізують цей інтерфейс. Врахуйте також, що всі потомки таких класів наслідують реалізацію. Наприклад, клас [java.awt.Component](#) реалізує інтерфейс [Serializable](#), значить, всі графічні компоненти можна серіалізувати. Не реалізують цей інтерфейс звичайно класи, тісно звязані з виконанням програм, наприклад, [java.awt.Toolkit](#). Стан екземплярів таких класів немає рації зберігати або передавати по мережі. Не реалізують інтерфейс [Serializable](#) і класи, що містять внутрішні дані [Java](#) "для службового користування". Десеріалізація відбувається так же просто, як і серіалізація:

```
ObjectInputStream ois = new ObjectInputStream(
new FileInputStream("myobjects.ser"));
MyClass mcl = (MyClass)ois.readObject();
int[] a = (int[])ois.readObject();
String s = (String)ois.readObject();
Date d = (Date)ois.readObject();
```

Треба тільки додержуватися порядку читання елементів потоку. В лістингі 18.6 ми створюємо об'єкт класу [GregorianCalendar](#) з поточною датою і часом, серіалізуємо його у файл [date.ser](#), через три секунди

десеріалізуємо і зрівнюємо з поточним часом. Результат показано на рис. 18.7.

Лістинг 18.6. Серіалізація обєкта

```
import java.io.*;
import java.util.*;
class SerDatef
public static void main(String[] args) throws Exception{
GregorianCalendar d = new GregorianCalendar();
ObjectOutputStream oos = new ObjectOutputStream(
new FileOutputStream("date.ser"));
oos.writeObject(d);
oos.flush();
oos.close();
Thread.sleep(3000);
ObjectInputStream ois = new ObjectInputStream(
new FileInputStream("date.ser"));
GregorianCalendar oldDate = (GregorianCalendar)ois.readObject();
ois.close();
GregorianCalendar newDate = new GregorianCalendar();
System.out.println("Old time = " +
oldDate.get(Calendar.HOUR) + ":" +
oldDate.get(Calendar.MINUTE) + ":" +
oldDate.get(Calendar.SECOND) +"\nNew time = " +
newDate.get(Calendar.HOUR) + ":" +
newDate.get(Calendar.MINUTE) + ":" +
newDate.get(Calendar.SECOND));
}
}
```

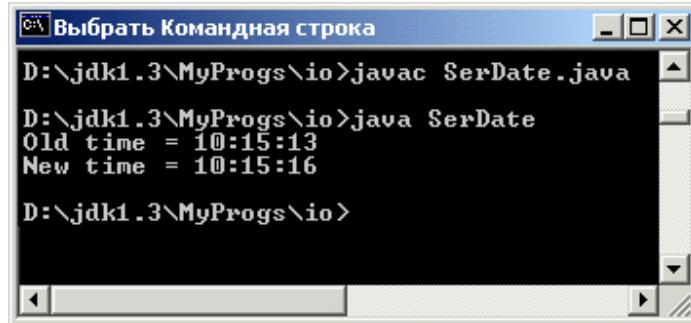


Рис.18.7. Серіалізація обєкта

Якщо не потрібно серіалізувати якесь поле, то достатньо помітити його службовим словом `transient`, наприклад:

```
transient MyClass me = new MyClass("abc", -12, 5.67e-5);
```

Метод `writeObject()` не записує у вихідний потік поля, помічені `static` і `transient`. Взагалі ж, цю ситуацію можна змінити, перевизначивши метод `writeObject()` або задавши список серіалізованих полів. Процес серіалізації навіть можна повністю налагодити під свої потреби, перевизначивши методи `введення/виведення` і скористувавшись допоміжними класами. Можна також взяти весь процес на себе, реалізувавши не інтерфейс `Serializable`, а інтерфейс `Externalizable`, але тоді прийдеться реалізувати методи `readExternal()` і `writeExternal()`, виконуючи введення/виведення.

Ми не будемо займатися тут цими питаннями. Якщо вам необхідно повністю освоїти процес серіалізації, то зверніться до специфікації `Java Object Serialization Specification`, розташований серед документації

J2SDK в каталозі `docs\guide\serialization\spec`. Там же єсть і приклади програм, реалізуючих цю специфікацію.

18.11. Друк в Java

Оскільки принтер — пристрій графічний, виведення на друк дуже схоже на виведення графічних об'єктів на екран. Тому в Java засоби друку входять в графічну бібліотеку AWT і в систему Java 2D. В графічному компоненті крім графічного контекста — об'єкта класу `Graphics`, створюється ще "друкарський контекст". Це теж об'єкт класу `Graphics`, але реалізуючий інтерфейс `printGraphics` і отриманий із іншого джерела — об'єкта класу `printjob`, що входить в пакет `java.awt`. Сам же цей об'єкт створюється за допомогою класу `Toolkit` пакета `java.awt`. На практиці це виглядає так:

```
PrintJob pj = getToolkit().getPrintJob(this, "Job Title", null);
Graphics pg = pj.getGraphics();
```

Метод `getPrintJob()` спочатку виводить на екран стандартне вікно `Print` операційної системи. Коли користувач вибере в цьому вікні параметри друку і почне друк кнопкою OK, створюється об'єкт `pj`. Якщо користувач відмовляється від друку кнопкою `Cancel`, то метод повертає `null`. В класі `Toolkit` два методи `getPrintJob()`:

- `getPrintJob(Frame frame, String jobTitle, JobAttributes jobAttr, PageAttributes pageAttr)`
- `getPrintJob(Frame frame, String jobTitle, Properties prop)`

Коротко охарактеризуємо аргументи методів.

- Аргумент `frame` указує на вікно верхнього рівня, управляюче друком. Цей аргумент не може бути `null`. Рядок `jobTitle` задає заголовок завдання, який не друкується, і може бути рівним `null`.
- Аргумент `prop` залежить від реалізації системи друку, часто це просто `null`, в даному випадку задаються стандартні параметри друку.
- Аргумент `jobAttr` задає параметри друку. Клас `JobAttributes`, екземпляром якого являється цей аргумент, має складну будоу. В ньому п'ять підкласів, які містять статичні константи — параметри друку, які використовуються в конструкторі класу. Єсть також конструктор по замовчуванню, задаючий стандартні параметри друку.
- Аргумент `pageAttr` задає параметри сторінки. Клас `PageProperties` теж містить п'ять підкласів із статичними константами, які і задають параметри сторінки і використовуються в конструкторі класу. Якщо для друку досить стандартних параметрів, то можна скористатися конструктором по замовчуванню.

Після того як "друкарський контекст" — об'єкт `pg` класу `Graphics` — визначений, можна викликати метод `print(pg)` або `printAll(pg)` класу `Component`. Цей метод установлює зв'язок з принтером по замовчуванню і викликає метод `paint(pg)`. На друк виводиться все те, що задано цим методом. Наприклад, щоб роздрукувати текстовий файл, треба в процесі введення розбити його текст на рядки і в методі `paint(pg)` вивести рядки методом `pg.drawString()` так же, як ми виводили їх на екран в уроці 9. При цьому слід врахувати, що в "друкарському контексті" немає шрифту по замовчуванню, завжди треба установлювати шрифт методом `pg.setFont()`.

Після виконання всіх методів `print()` застосовується метод `pg.dispose()`, викликаючий прогін сторінки, і метод `pj.end()`, закінчуючий друк.

В лістингі 18.7 приведено простий приклад друку тексту і кола, заданих в методі `paint()`. Цей метод працює два рази: перший раз креслячи текст і коло на екрані, другий раз, точно так же, на листі паперу, вставленому в принтер. Всі методи друку зібрані в один метод `simplePrint()`.

Лістинг 18.7. Друк засобами AWT

```
import java.awt.*;
import java.awt.event.*;
class PrintTest extends Frame{
PrintTest(String s){
```

```

super(s);
setSize(400, 400);
setVisible(true);
}
public void simplePrint(){
PrintJob pj =
getToolkit().getPrintJob(this, "JobTitle", null);
if (pj != null){
Graphics pg = pj.getGraphics();
if (pg != null){
print(pg);
pg.dispose();
} else System.out.println("Graphics's null");
pj.end();
} else System.out.println("Job's null");
}
public void paint(Graphics g){
g.setFont(new Font("Serif", Font.ITALIC, 30));
g.setColor(Color.black);
g.drawString("Страница 1", 100, 100);
}
public static void main(String[] args){
PrintTest pt = new PrintTest(" Простий приклад друку");
pt.simplePrint();
pt.addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent ev) {
System.exit(0);
}
});
}
}
}

```

18.12. Друк засобами Java 2D

Розширенна графічна система [Java 2D](#) пропонує нові інтерфейси і класи для друку, зібрани в пакет [java.awt.print](#). Ці класи повністю перекривають всі стандартні можливості друку бібліотеки [AWT](#). Більше того, вони зручніші в роботі і пропонують додаткові можливості. Якщо ви маєте цей пакет, то, безсумнівно, треба використовувати його, а не стандартні засоби друку [AWT](#). Як і стандартні засоби [AWT](#), методи класів [Java 2D](#) виводять на друк вміст графічного контексту, заповненого методами класу [Graphics](#) або класу [Graphics2D](#).

Всякий клас [Java 2D](#), що збирається друкувати хоча б одну сторінку тексту, графіки або зображення називається класом, *рисуючим сторінки* (*page painter*). Такий клас повинен реалізувати інтерфейс [Printable](#). В цьому інтерфейс описані дві константи і тільки один метод [print\(\)](#). Клас, рисуючий сторінки, повинен реалізовувати цей метод. Метод [print\(\)](#) повертає ціле типу [int](#) і має три аргументи:

```
print(Graphics g, PageFormat pf, int ind);
```

Перший аргумент [g](#) — це графічний контекст, що виводиться на лист паперу, другий аргумент [pf](#) — екземпляр класу [PageFormat](#), визначаючий розмір і орієнтацію сторінки, третій аргумент [ind](#) — порядковий номер сторінки, що відраховується з нуля. Метод [print\(\)](#) класу, рисуючого сторінки, заміняє собою метод [paint\(\)](#), що використовується стандартними засобами друку [AWT](#). Клас, рисуючий сторінки, не зобовязаний розширити клас [Frame](#) і перевизначити метод [paint\(\)](#). Все заповнення графічного контексту методами класу [Graphics](#) або [Graphics2D](#) тепер виконується в методі [print\(\)](#). Коли друк сторінки буде закінчено, метод [print\(\)](#) повинен повернути ціле значення, задане константою [PAGE_EXISTS](#). Буде зроблено повторне звернення до методу [print\(\)](#) для друку наступної сторінки. Аргумент [ind](#) при цьому зростає на 1. Коли [ind](#) перевищить кількість сторінок, метод [print\(\)](#) повинен повернути значення [NO_SUCH_PAGE](#), що служить сигналом закінчення друку.

Слід памятати, що система друку може декілька раз звернутися до методу `paint()` для друку однієї і тієї ж сторінки. При цьому аргумент `ind` не змінюється, а метод `print()` повинен створити той же графічний контекст.

Клас `PageFormat` визначає параметри сторінки. На сторінці вводиться система координат з одиницею довжини 1/72 дюйма, початок якої і напрям вісей визначається однією із трьох констант:

- `PORTRAIT` — початок координат розташовано в лівому верхньому куті сторінки, вісь `Ox` направлена вправо, вісь `Oy` — вниз;
- `LANDSCAPE` — початок координат в лівому нижньому куті, вісь `Ox` йде вверх, вісь `Oy` — вправо;
- `REVERSE_LANDSCAPE` — початок координат в правому верхньому куті, вісь `Ox` йде вниз, вісь `Oy` — вліво.

Більшість принтерів не може друкувати без полів, на всій сторінці, а здійснює виведення тільки в деяку область друку (`imageable area`), координати лівого верхнього кута якої повертаються методами `getImageableX()` і `getImageableY()`, а ширина і висота — методами `getImageableWidth()` і `getImageableHeight()`. Ці значення треба враховувати при розташуванні елементів в графічному контексті, наприклад, при розміщенні рядків тексту методом `drawstring()`, як це зроблено в листинзі 18.9. В класі тільки один конструктор по замовчуванню `PageFormat()`, задаючий стандартні параметри сторінки, визначені для принтера по замовчуванню обчислювальною системою.

Виникає питання: "Як же тоді задати параметри сторінки?" Відповідь проста: "За допомогою стандартного вікна операційної системи". Метод `pageDialog(PageDialog pd)` відкриває на екрані стандартне вікно Параметри сторінки (`Page Setup`) операційної системи, в якому уже задані параметри, визначені в обєкті `pd`. Якщо користувач вибрав у цьому вікні кнопку Відміна, то повертається посилання на об'єкт `pd`, якщо кнопку `OK`, то створюється і повертається посилка на новий об'єкт. Об'єкт `pd` в будь-якому випадку не змінюється. Він звичайно створюється конструктором. Можна задати параметри сторінки і із програми, але тоді слід спочатку визначити об'єкт класу `Paper` конструктором по замовчуванню:

```
Paper p = new Paper()
```

Потім методами

```
p.setSize(double width, double height)
p.setImageableArea(double x, double y, double width, double height)
```

задати розмір сторінки і області друку. Потім визначити об'єкт класу `pageFormat` з параметрами по замовчуванню:

```
PageFormat pf = new PageFormat()
```

і задати нові параметри методом

```
pf.setPaper(p)
```

Тепер викликати на екран вікно Параметри сторінки методом `pageDialog()` уже не обов'язково, і ми отримаємо мовчазний (`silent`) процес друку. Так робиться в тих випадках, коли друк виконується на фоні окремим підпроцесом. Отже, параметри сторінки визначені, метод `print()` — теж. Тепер треба дати завдання на друк (`print job`) — указати кількість сторінок, їх номери, порядок друку сторінок, кількість копій. Всі ці дані збираються в класі `PrinterJob`.

Система друку `Java 2D` розрізняє два види звадань. В більш простих зваданнях — `Printable Job` — єсть тільки один клас, рисуючий сторінки, тому у всіх сторінок одні і ті ж параметри, сторінки друкуються послідовно з першої по останню або з останньої сторінки по першу, це залежить від системи друку. Другий, більш складний вид завдань — `Pageable Job` — визначає для друку кожної сторінки свій клас, рисуючий сторінки, тому у кожної сторінки можуть бути власні параметри. Крім того, можна друкувати не все, а тільки вибрані сторінки, виводити їх в зворотному порядку, друкувати на обох сторонах листа. Для здійснення цих можливостей визначається екземпляр класу `Book` або створюється клас, реалізуючий

інтерфейс [Pageable](#). В класі [Book](#), знову-таки, один конструктор, створюючий порожній об'єкт:

```
Book b = new Book()
```

Після створення в даний об'єкт додаються класи, рисуючі сторінки. Для цього в класі [Book](#) є два методи:

- [append \(Printable p, PageFormat pf\)](#) — додає об'єкт [p](#) в кінець;
- [append\(Printable p, PageFormat pf, int numPages\)](#) — додає [numPages](#) екземплярів [p](#) в кінець; якщо число сторінок невідомо, то задається константа [UNKNOWN_NUMBER_OF_PAGES](#).

При складанні завдання на друк, тобто після створення екземпляра класу [PrinterJob](#), треба указати вид завдання одним і тільки одним із трьох методів цього класу [setPrintable\(Printable pr\)](#), [setPrintable\(Printable pr, PageFormat pf\)](#) або [setPageable \(Pageable pg\)](#). Заодно задаються один або декілька класів [pr](#), рисуючих сторінки в цьому завданні. Решта параметрів завдання можна задати в стандартному діалоговому вікні Print операційної системи, яке відкривається на екрані при виконанні логічного методу [printDialog\(\)](#). Указаний метод не має аргументів. Він поверне [true](#), коли користувач клікне по кнопці [OK](#), і [false](#) після натискання кнопки Відміна.

Залишається задати число копій, якщо воно більше 1, методом [setCopies\(int n\)](#) і завдання сформовано. Ще один корисний метод [defaultPage\(\)](#) класу [PrinterJob](#) повертає об'єкт класу [PageFormat](#) по замовчуванню. Цей метод можна використовувати замість конструктора класу [PageFormat](#).

Залишилось сказати, як створюється екземпляр класу [PrinterJob](#). Оскільки цей клас тісно звязаний з системою друку комп'ютера, його об'єкти створюються не конструктором, а статичним методом [getPrinterJob\(\)](#), що знаходиться в тому ж самому класі [Printer Job](#). Початок друку задається методом [print\(\)](#) класу [PrinterJob](#). Цей метод не має аргументів. Він послідовно викликає методи [print\(g, pf, ind\)](#) класів, рисуючих сторінки, для кожної сторінки.

Зберемо все це вмісті в лістинзі 18.8. В ньому засобами [Java 2D](#) друкується те ж, що і в лістинзі 18.7. Зверніть увагу на п. 6. Після закінчення друку програма не закінчується автоматично, для її завершеннями звертаємося до методу [System.exit \(0\)](#).

Лістинг 18.8. Простий друк методами Java 2D

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.print.*;
class Print2Test implements Printable{
public int print(Graphics g, PageFormat pf, int ind)
throws PrinterException{ // Друкуємо не більше 5 сторінок
if (ind > 4) return Printable.NO_SUCH_PAGE;
Graphics2D g2 = (Graphics2D)g;
g2.setFont(new Font("Serif", Font.ITALIC, 30));
g2.setColor (Color.black);
g2.drawString("Page " + (ind + 1), 100, 100);
g2.draw(new Ellipse2D.Double(100, 100, 200, 200));
return Printable.PAGE_EXISTS;
}
public static void main(String[] args){
// 1. Створюємо екземпляр завдання
PrinterJob pj = Printer Job.getPrinter Job();
// 2. Відкриваємо діалогове вікно Параметри сторінки
PageFormat pf = pj.pageDialog (pj.defaultPage() );
// 3. Задаємо вид завдання, об'єкт класу, рисуючого сторінку і вибрані параметри
// сторінки
pj.setPrintable(new Print2Test(), pf);
// 4. Якщо потрібно надрукувати декілька копій, то:
pj.setCopies(2); // По замовчуванню друкується одна копія
}
```

```

// 5. Відкриваємо діалогове вікно Друк (необовязково)
if (pj.printDialog())( // Якщо OK... try{
pj.print(); // Звертається до print(g, pf, ind)
}catch(Exception e){
System.out.println(e);
}
}
// 6. Завершуємо завдання
System.exit(0);
}
}

```

18.13. Друк файла

Друк текстового файла заключається в розміщенні його рядків в графічному контексті методом `drawString()`. При цьому необхідо прослідкувати за правильним розміщенням рядків в області друку і розбиттям файла на сторінці. В листинзі 18.9 приведений спрощений приклад друку текстового файла, імя якого задається в командному рядку. Із файла читаються готові рядки, програма не порівнює їх довжину із шириною області друку, не виділяє абзаци. Виведення відбувається в локальному кодуванні.

Листинг 18.9. Друк текстового файла

```

import java.awt.*;
import java.awt.print.*;
import java.io.*;
public class Print2File{
public static void main(String[] args){
if (args.length < 1){
System.out.println("Usage: Print2File path");
System.exit(0);
}
PrinterJob pj = PrinterJob.getPrinterJob();
PageFormat pf = pj.pageDialog(pj.defaultPage());
pj.setPrintable(new FilePagePainter(args[0]), pf);
if (pj.printDialog()){
try{
pj.print();
}catch(PrinterException e){}
}
System.exit(0);
}
}
class FilePagePainter implements Printable{
private BufferedReader br;
private String file;
private int page = -1;
private boolean eof;
private String[] line;
private int numLines;
public FilePagePainter(String file){
this.file = file;
try{
br = new BufferedReader(new FileReader(file));
}catch(IOException e){ eof = true; }
}
public int print(Graphics g, PageFormat pf, int ind)
throws PrinterException{
g.setColor(Color.black);
g.setFont(new Font("Serif", Font.PLAIN, 10));

```

```

int h = (int)pf.getImageableHeight();
int x = (int)pf.getImageableX() + 10;
int y = (int)pf.getImageableY() + 12;
try{
// Якщо система друку запросила цю сторінку перший раз
if (ind != page){
if (eof) return Printable.NO_SUCH_PAGE;
page = ind;
line = new String[h/12]; // Масив рядків на сторінці
numLines = 0; // Число рядків на сторінці
// Читаемо рядки із файла і формуємо масив рядків
while (y + 48 < pf.getImageableY() + h){
line[numLines] = br.readLine();
if (line[numLines] == null){
eof = true; break; }
numLines++;
y += 12;
}
}
// Розміщуємо колонтитул
y = (int)pf.getImageableY() + 12;
g.drawString("Файл: " + file + ", сторінка " +
(ind + 1), x, y);
// Залишаємо два порожні рядки
y += 36;
// Розміщуємо рядки тексту поточної сторінки
for (int i = 0; i < numLines; i++){
g.drawString(line[i], x, y) ;
y += 12;
}
return Printable.PAGE_EXISTS;
} catch(LOException e){
return Printable.NO_SUCH_PAGE;
}
}
}
}

```

18.14. Друк сторінок з різними параметрами

Друк виду **Printable Job** не зовсім зручний — у всіх сторінок повинні бути однакові параметри, неможна задати число сторінок і порядок їх друку, у вікні Параметри сторінки не видно число сторінок, що виводиться на друк. Всі ці можливості представляє друк виду **Pageable Job** за допомогою класу **Book**. Як уже говорилось вище, спочатку створюється порожній об'єкт класу **Book**, потім до нього додаються різні або однакові класи, рисуючі сторінки. При цьому визначаються об'єкти класу **pageFormat**, задаючі параметри цих сторінок, і число сторінок. Якщо число сторінок невідомо, то замість нього указується константа **UNKNOWN_NUMBER_OF_PAGES**. В такому випадку сторінки будуть друкуватися в порядку зростання їх номерів до тих пір, поки метод **print()** не поверне **NO_SUCH_PAGE**. Метод **setPage(int pageIndex, Printable p, PageFormat pf)** заміняє об'єкт в позиції **pageIndex** на новий об'єкт **p**.

В програмі лістинга 18.10 створюються два класи, рисуючі сторінки: **Cover** і **Content**. Ці класи дуже прості — в них тільки реалізований метод **print()**. Клас **Cover** рисує титульний лист крупним напівжирним шрифтом. Текст друкується знизу вверх вподовж довгої сторони аркуша на його правій половині. Клас **Content** виводить звичайний текст звичайним способом. Параметри титульного аркуша визначаються в класі **pf1**, параметри других сторінок задаються в діалоговому вікні Параметри сторінки і містяться в класі **pf2**. В об'єкт **bk** класу **Book** занесені три сторінки: перша сторінка — титульний аркуш, на двох інших друкується один і той же текст, записаний в методі **print()** класу **Content**.

Лістинг 18.10. Друк сторінок з різними параметрами

```

import j ava.awt.*;
import j ava.awt.print.*;
public class Print2Book{
public static void main(String[]
args){
PrinterJob pj = PrinterJob.getPrinterJob();
// Для титульного аркуша вибирається альбомна орієнтація
PageFormat pfl = pj.defaultPage();
pfl.setOrientation(PageFormat.LANDSCAPE);
// Параметри інших сторінок задаються в діалоговому вікні
PageFormat pf2 = pj.pageDialog (new PageFormat());
Book bk = new Book();
// Перша сторінка – титульний аркуш
bk.append(new Cover(), pfl);
// Дві інших сторінки
bk.append(new Content(), pf2, 2);
// Визначається вид друку – Pageable Job
pj.setPageable(bk);
if (pj.printDialog()){
try{
pj.print();
}catch (Exception e){}
}
System.exit(0);
}
}

class Cover implements Printable{
public int print(Graphics g, PageFormat pf, int ind)
throws PrinterException{
g.setFont (new Font ("Helvetica-Bold", Font.PLAIN, 40)) ;
g.setColor(Color.black) ;
int y = (int) (pf.getImageableY() + pf.getImageableHeight() /2);
g.drawString("Це заголовок.", 72, y);
g.drawString("Він друкується вподовж довгої", 72, y+60);
g.drawString("сторони аркуша паперу.", 72, y+120);
return Printable.PAGE_EXISTS;
}
}

class Content implements Printable{
public int print(Graphics g, PageFormat pf, int ind)
throws PrinterException{
Graphics2D g2 = (Graphics2D)g;
g2.setFont(new Font("Serif", Font.PLAIN, 12));
g2.setColor(Color.black);
int x = (int)pf.getImageableX() + 30;
int y = (int)pf.getImageableY();
g2.drawString("Це рядки звичайного тексту.", x, y += 16);
g2.drawString("Вони друкуються з параметрами,", x, y += 16);
g2.drawString("вибраними в діалоговому вікні.", x, y += 16);
return Printable.PAGE_EXISTS;
}
}

```

Урок 19

Мережеві засоби Java

- [Робота в WWW](#)
- [Робота по протоколу TCP](#)
- [Робота по протоколу UDP](#)

Коли число комп'ютерів в установі перевалює за десяток і співробітникам набридає бігати з дискетами один до одного для обміна файлами, тоді в комп'ютери вставляються мережеві карти, протягуються кабелі і комп'ютери об'єднуються в мережу. Спочатку всі комп'ютери в мережі рівноправні, вони роблять одне і те ж — це однорангова (*peer-to-peer*) мережа. Потім купують комп'ютер з величими і швидкими жорсткими дисками, і всі файли установи починають зберігатися на даних дисках — цей комп'ютер становиться файл-сервером, що надає послуги зберігання, пошуку, архівації файлів. Потім купують дорогий і швидкий принтер. Комп'ютер, звязаний з ним, становиться принт-сервером, надаючим послуги друку. Потім появляються графічний сервер, обчислювальний сервер, сервер бази даних. Решта комп'ютерів становляться клієнтами цих серверів. Така архітектура мережі називається архітектурою **клієнт-сервер** (*client-server*).

Сервер постійно знаходить в стані очікування, він *прослуховує* (*listen*) мережу, чекаючи запиту від клієнтів. Клієнт звязується з сервером і посилає йому *запит* (*request*) з описанням послуги, наприклад, ім'я потрібного файла. Сервер обробляє запит і відправляє відповідь (*response*), в нашому прикладі, файл, або повідомлення про неможливість надати послугу. Після цього звязок може бути розірваним або продовжитися, організуючи сеанс звязку, названий *сесією* (*session*).

Запити клієнта і відповіді сервера формуються по строгим правилам, сукупність яких утворює **протокол** (*protocol*) звязку. Всякий протокол повинен, перш за все, містити правила зєднання комп'ютерів. Клієнт перед посиланням запиту повинен впевнитися, що сервер в робочому стані, прослуховує мережу, і почув клієнта. Пославши запит, клієнт повинен бути упевненим, що запит дійшов до сервера, сервер зрозумів запит і готовий відповісти на нього. Сервер повинен упевнитися, що відповідь дійшла до клієнта. Закінчення сесії повинно бути чітко зафіксовано, щоб сервер міг звільнити ресурси, заняті обробкою запитів клієнта.

Всі правила, утворюючі протокол, повинні бути зрозумілими, однозначними і короткими, щоб не загромождати мережу. Тому повідомлення, що пересилаються по мережі, нагадують шифровки, в них має значення кожний біт. Отже, всі мережеві зєднання основані на трьох основних поняттях: клієнт, сервер і протокол. Клієнт і сервер — поняття відносні. В одній сесії комп'ютер може бути сервером, а в другій — клієнтом. Наприклад, файл-сервер може послати принт-серверу файл на друк, становлячись його клієнтом.

Для обслуговування протоколу: формування запитів і відповідей, перевірок їх відповідності протоколу, розшифровки повідомлень, звязку з мережевими пристроями створюється программа, що складається із двох частин. Одна частина програми працює на сервері, друга — на клієнті. Ці частини так і називаються серверною частиною програми і клієнтською частиною програми, або, коротше, сервером і клієнтом. Дуже часто клієнтська і серверна частини програми пишуться окремо, різними фірмами, оскільки від цих програм вимагається тільки, щоб вони дотримувалися протоколу. Більш того, по кожному протоколу працюють десятки клієнтів і серверів, що відрізняються різними зручностями.

Звичайно на одному комп'ютері-сервері працюють декілька програм-серверів. Одна програма займається електронною поштою, друга — пересилкою файлів, третя надає *Web*-сторінки. Для того щоб їх відрізняти, кожній програмі-серверу надається *номер порта* (*port*). Це просто ціле додатне число, яке указує клієнт, що звертається до певної програми-серверу. Число, взагалі кажучи, може бути будь-яким, але найбільш розповсюдженим протоколам даються стандартні номери, щоб клієнти були твердо упевнені, що звертаються до потрібного сервера. Так, стандартний номер порта електронної пошти 25, пересилки файлів — 21, *Web*-сервера — 80. Стандартні номери простягаються від 0 до 1023. Числа, починаючи з 1024 до 65 535, можна використовувати для своїх власних номерів портів.

Все це схоже на телевізійні канали. Клієнт-телевізор звертається за допомогою антени до сервера-телецентру і вибирає номер каналу. Він упевнений, що на першому каналі НТН, на руці — 1+1 і т. д.

Щоб рівномірно розподілити навантаження на сервер, часто декілька портів прослуховуються програмами-серверами одного типу. [Web-сервер](#), крім порту з номером 80, може прослуховувати порт 8080, 8001 і ще який-небудь.

В процесі передачі повідомлення використовується декілька протоколів. Навіть коли ми відправляємо листа, ми спочатку пишемо повідомлення, починаючи його: "Вельмишановний Іване Петрович!" і закінчуєчи: "Щиро Ваш". Це один протокол. Можна почати листа словами: "Вася, привіт!" і закінчити: "Ну, пока". Це інший протокол. Потім ми кладемо листа в конверт і пишемо на ньому адресу по протоколу, запропонованому Міністерством звязку. Потім лист попадає на пошту, упаковується в мішок, на якому пишеться адреса по протоколу поштового звязку. Мішок завантажується в літак чи потяг, який переміщається по своєму протоколу. Зверніть увагу, що кожний протокол тільки додає до повідомлення свою інформацію, не змінюючи його, нічого не знаяючи про те, що зроблено по попередньому протоколу і що буде зроблено по правилах наступного протокола. Це дуже зручно — можна програмувати один протокол, нічого не знаяючи про інші протоколи.

Перш ніж дійти до адресата, лист проходить зворотний шлях: виймається із літака чи потяга, потім із мішка, потім із конверта. Тому говорять про стек (stack) протоколів: "Першим прийшов, останнім вийшов". В сучасних глобальних мережах прийнятий стек із чотирьох протоколів, названий стеком протоколів [TCP/IP](#). Спочатку ми пишемо повідомлення, користуючись програмою, реалізуючу [прикладний \(application\)](#) протокол: [HTTP \(80\)](#), [SMTP \(25\)](#), [TELNET \(23\)](#), [FTP \(21\)](#), [POP3 \(100\)](#) або інший протокол. В дужках записано стандартний номер порта. Потім повідомлення обробляється по [транспортному \(transport\)](#) протоколу. До нього додають номера портів відправника і отримувача, контрольна сума і довжина повідомлення. Найбільш розповсюджені транспортні протоколи [TCP \(Transmission Control Protocol\)](#) і [UDP \(User Datagram Protocol\)](#). В результаті роботи протокола [TCP](#) отримується [TCP-пакет \(packet\)](#), а протокола [UDP](#) — [дейтаграма \(datagram\)](#).

Дейтаграма невелика — всього біля кілобайта, тому повідомлення ділиться на частини, із яких створюються окремі дейтаграми. Дейтаграми посилаються одна за другою. Вони можуть іти до отримувача різними маршрутами, прийти зовсім в іншому порядку, деякі дейтаграми можуть загубитися. Прикладна програма отримувача повинна сама потурбуватися про те, щоб зібрати із отриманих дейтаграм вихідне повідомлення. Для цього звичайно перед посилкою частини повідомлення нумеруються, як сторінки в книзі. Таким чином, протокол [UDP](#) працює як поштова служба. Посилаячи книгу, ми розрізаемо її на сторінки, кожну сторінку відправляємо в своєму конверті, і ніколи не впевнені, що всі листи дойдуть до адресата.

[TCP](#)-пакет тоже невеликий, і пересилання також іде орємими пакетами, але протокол [TCP](#) забезпечує надійний звязок. Спочатку установлюється зєднання з отримувачем. Тільки після цього посилаються пакеты. Отримання кожного пакета підтверджується отримувачем, при помилці посилка пакета повторяється. Повідомленні акуратно збирається отримуваче. Для відправника і отримувача створюється враження, що пересилаються не пакети, а суцільний потік байтів, тому передачу повідомлень по протоколу [TCP](#) часто називають передачею потоком. Звязок по протоколу [TCP](#) більше нагадує телефону розмову, ніж поштовий звязок.

Далі повідомленням займається протокол, реалізуючий мережевий ([network](#)) протокол. Частіше всього це протокол [IP \(Internet Protocol\)](#). Він додає до повідомлення адресу відправника і адресу отримувача, і другі дані. В результаті одержується [IP-пакет](#). Нарешті, [IP-пакет](#) поступає в програму, працюючій по [канальному \(link\)](#) протоколу [ENET](#), [SLIP](#), [PPP](#), і повідомлення приймає вид, придатний для передачі по мережі.

На стороні отримувача повідомлення проходить через ці чотири рівня протоколів у зворотному порядку, звільнюючись від службової інформації, і доходить до програми, реалізуючої [прикладний протокол](#). Яка ж адреса заноситься в [IP-пакет](#)? Кожний комп'ютер або інший пристрій, підключений до обєднання мереж [Internet](#), так званий [хост \(host\)](#), отримує унікальний номер — чотирибайтове ціле число, назване [IP-адресою \(IP-address\)](#). По традиції вміст кожного байта записується десятковим числом від 0 до 255, названим [октетом \(octet\)](#), і ці числа пишуться через точку: 138.2.45.12 або 17.056.215.38. [IP-адреса](#) зручна для машини, але незручна для людини. Уявіть собі рекламний заклик: "Заходьте на наш сайт 154.223.145.26!" Тому [IP-адреса](#) хоста дублюється [доменним іменем \(domain name\)](#). В доменному імені присутні коротке позначення країни: [ru](#) — Росія, [ua](#) — Україна, [de](#) — ФРН і т. д., або позначення типу установи: [com](#) — комерційна структура, [org](#) — суспільна організація, [edu](#) — освітня установа. Далі

указується регіон: `msc.ru` — Москва, `spb.ru` — Санкт-Петербург, `kcn.ru` — Казань, або установа: `bhv.ru` — "БХВ-Петербург", `ksu.ru` — Казанський держуніверситет, `sun.com` — SUN Microsystems. Потім підрозділ: `www.bhv.ru`, `java.sun.com`. Такий ланцюжок коротких позначень можна продовжувати і далі. В Java IP-адреси і доменне імя об'єднуються в один клас `inetAddress` пакета `java.net`. Екземпляр цього класу створюється статичним методом `getByName(string host)` даного ж класу, в якому аргумент `host` — це доменне імя або IP-адреса.

Робота у WWW

Серед програмного забезпечення Internet велике розповсюдження отримала інформаційна система **WWW** (World Wide Web), основана на прикладному протоколі **HTTP** (Hypertext Transfer Protocol). В ній використовується розширенна адресація, названа **URL** (Uniform Resource Locator). Ця адресація має такі схеми:

[protocol://authority@host:port/path/file#ref](#)
[protocol://authority@host:port/path/file/extra_path?info](#)

Тут необов'язкова частина `authority` — це пара імя:пароль для доступу до хосту, `host` — це IP-адреса або доменне імя хоста. Наприклад:

[http://www.bhv.ru/](#)
[http://132.192.5.10:8080/public/some.html](#)
[ftp://guest:password@lenta.ru/users/local/pub](#)
[file:///C:/text/html/index.htm](#)

Якщо якийсь елемент `URL` відсутній, то береться стандартне значення. Наприклад, в першому прикладі номер порта `port` рівний 80, а імя файла `path` — якийсь головний файл, визначений хостом, частіше всього це файл з іменем `index.html`. В третьому прикладі номер порта рівний 21. В останньому прикладі у формі `URL` просто записано імя файла `index.htm`, розташованого в папці `C:` жорсткого диска тієї ж самої машини.

В Java для роботи з `URL` єсть клас `URL` пакета `java.net`. Об'єкт цього класу створюється одним із шести конструкторів. В основному конструкторі `URL(String url)` задається розширенна адреса `url` у вигляді рядка. Крім методів доступу `getxxxo`, дозволяючих отримати елементи `URL`, в цьому класі єсть два цікавих методи:

- `openConnection()` — визначає зв'язок з `URL` і повертає об'єкт класу `URLConnection`;
- `openStream()` — установлює зв'язок з `URL` і відкриває вхідний потік у вигляді повернутого об'єкта класу `InputStream`.

Лістинг 19.1 показує, як легко можна отримати файл із Internet, користуючись методом `openStream()`.

Лістинг 19.1. Отримання Web-сторінки

```
import java.net.*;
import java.io.*;
class SimpleURL{
public static void main(String[] args){
try{
URL bhv = new URL("http://www.bhv.ru/");
BufferedReader br = new BufferedReader(
new InputStreamReader(bhv.openStream()));
String line;
while ((line = br.readLine()) != null)
System.out.println(line);
br.close();
} catch(MalformedURLException me){
System.out.println("Unknown host: " + me);
}
}
}
```

```

System.exit(0);
}catch( IOException ioe){
System.err.println("Input error: " + ioe);
}
}
}
}

```

Якщо вам треба не тільки отримати інформацію з хосту, але і узнати її тип: текст, гіпертекст, архівний файл, зображення, звук, або вияснити довжину файла, або передати інформацію на хост, то необхідно спочатку методом `openConnection()` створити об'єкт класу `URLConnection` або його підкласу `HttpURLConnection`. Після створення об'єкта з'єднання ще не установлено, і можна задати параметри звязку. Це робиться наступними методами:

- `setDoOutput (boolean out)` — якщо аргумент `out` рівний `true`, то передача піде від клієнта на хост; значення по замовчуванню `false`;
- `setDoInput (boolean in)` — якщо аргумент `in` рівний `true`, то передача піде з хоста до клієнта; значення по замовчуванню `true`, але якщо уже виконано `setDoOutput(true)`, то значення по по замовчуванню рівне `false`;
- `setUseCaches (boolean cache)` — якщо аргумент `cache` рівний `false`, то передача піде без кешування, якщо `true`, то приймається режим по замовчуванню;
- `setDefaultUseCaches(boolean default)` — якщо аргумент `default` рівний `true`, то приймається режим кешування, передбачений протоколом;
- `setRequestProperty(String name, String value)` — додає параметр `name` із значенням `value` до заголовку посланого повідомлення.

Після задання параметрів треба установити з'єднання методом `connect()`. Після з'єднання задання параметрів уже неможливо. Треба врахувати, що деякі методи доступу `getXXX()`, яким треба отримати свої значення з хосту, автоматично уstanавлюють з'єднання, і звертання до методу `connect()` становиться зливим. `Web`-сервер повертає інформацію, запрошену клієнтом, разом із заголовком, дані із якого можна отримати методами `getXXX()`, наприклад:

- `getContentType()` — повертає рядок типу `string`, що показує тип пересланої інформації, наприклад, `"text/html"`, або `null`, якщо сервер його не указав;
- `getContentLength()` — повертає довжину отриманої інформації в байтах або — 1, якщо сервер її не указав;
- `getContent ()` — повертає отриману інформацію ізгляді об'єкта типу `Object`;
- `getContentEncoding()` — повертає рядок типу `string` з кодуванням отриманої інформації, або `null`, якщо сервер її не указав.

Два методи повертають потоки введення/виведення, створені для даного з'єднання:

- `getInputStream()` — повертає вхідний потік типу `InputStream`;
- `getOutputStream()` — повертає вихідний потік типу `OutputStream`.

Інші методи, а їх близько двадцяти, повертають різні параметри з'єднання. Звернення до методу `bhv.Openstream()`, записане в лістингі 19.1, — це, на самім ділі, скорочення запису `bhv.openConnection().getInputStream()`. В лістингі 19.2 показано, як переслати рядок тексту по адресу `URL`. `Web`-сервер, який отримує цей рядок, не знає, що робити з отриманою інформацією. Занести її в файл? Але з яким іменем, і чи має він право створювати файли? Переслати на іншу машину? Але куди? Вихід було знайдено в системі `CGI (Common Gateway Interface)`, яка працює наступним чином. При посилці повідомлення ми указуємо `URL` виконуваного файла деякої програми, розміщеної на машині-сервері. Отримавши повідомлення, `Web`-сервер запускає цю програму і передає повідомлення на її стандартне введення. Саме програма і знає, що робити з отриманим повідомленням. Вона опрацьовує повідомлення і виводить результат на своє стандартне виведення. `Web`-сервер підключається до стандартного виведення, приймає результат і відправляє його назад клієнту. `CGI`-программу можна написати на будь-якій мові: `C`, `C++`, `Pascal`, `Perl`, `PHP`, лише би в ній було стандартне введення і стандартне виведення. Можна написати її і на `Java`, але в технології `Java` є ще більш досконале вирішення цієї задачі за допомогою сервлетів (`servlets`). `CGI`-програми звичайно лежать на сервері в каталозі `cgi-bin`.

Лістинг 19.2. Посилка рядка на адресу URL

```

import java.net.*;
import java.io.*;
class PostURL{
public static void main(String[] args){
String req = "This text is posting to URL";
try{
// Указуємо URL потрібної CGI-програми
URL url = new URL("http://www.bhv.ru/cgi-bin/some.pl");
// Створюємо обєкт uc
URLConnection uc = url.openConnection();
// Збираємось відправляти
uc.setDoOutput(true);
// і отримувати повідомлення
uc.setDoInput(true);
// без кешування
uc.setUseCaches(false);
// Задаємо тип
uc.setRequestProperty("content-type",
"application/octet-stream");
// і довжину повідомлення
uc.setRequestProperty("content-length", "" + req.length());
// Установлюємо зв'язок
uc.connect();
// Відкриваємо вихідний потік
DataOutputStream dos = new DataOutputStream( uc.getOutputStream());
// і виводим в нього повідомлення, посилаючи його на адресу
URL dos.writeBytes(req);
// Закриваємо вихідний потік
dos.close();
// Відкриваємо вхідний потік для відповіді сервера
BufferedReader br = new BufferedReader(new InputStreamReader(
uc.getInputStream() ) );
String res = null;
// Читаємо відповідь сервера і виводимо його на консоль
while ((res = br.readLine()) != null)
System.out.println(res);
br.close();
}catch(MalformedURLException me){
System.out.println(me);
}catch(UnknownHostException he){
System.out.println(he);
}catch(UnknownServiceException se){
System.out.println(se);
}catch(IOException ioe){
System.out.println(ioe);
}
}
}
}

```

Робота по протоколу TCP

Програми-сервери, прослуховуючі свої порти, працюють під управлінням операційної системи. У машин-серверів можуть бути самі різноманітні операційні системи, особливості яких передаються програмам-серверам. Щоб згладити відмінності в реалізаціях різних серверів, між сервером і портом введений проміжний програмний шар, названий **сокетом (socket)**. Англійське слово **socket** перекладається як електричний розім, розетка. Так само як до розетки за допомогою вилки можна підключити будь-який електричний пристрій, лише би він був розрахований на 220 В і 50 Гц, до сокета можна приєднати будь-

якого клієнта, лише би він працював по тому ж протоколу, що і сервер. Кожний сокет звязаний ([bind](#)) з одним портом, говорять, що сокет прослуховує ([listen](#)) порт. Зєднання за допомогою сокетів устанавлюється так.

1. Сервер створює сокет, прослуховуючий порт сервера.
2. Клієнт також створює сокет, через який звязується з сервером, сервер починає устанавлювати ([accept](#)) звязок з клієнтом.
3. Установлюючи звязок, сервер створює новий сокет, прослуховуючий порт з другим, новим номером, і повідоляє цей номер клієнту.
4. Клієнт посилає запит на сервер через порт з новим номером.

Після цього зєднання становиться симетричним — два сокети обмінюються інформацією, а сервер через старий сокет продовжує прослуховувати попередній порт, чекаючи наступного клієнта. В [Java](#) сокет — це обєкт класу [socket](#) із пакета [java.io](#). В класі шість конструкторів, в які різними способами заноситься адреса хоста і номер порта. Частіше всього застосовується конструктор

[Socket\(String host, int port\)](#)

Численні методи доступу устанавлюють і отримують параметри сокета. Ми не будемо заглиблюватися в їх вивчення. Нам знадобляться тільки методи, створюючі потоки введення/виведення:

- [getInputStream\(\)](#) — повертає вхідний потік типу [InputStream](#);
- [getOutputStream\(\)](#) — повертає вихідний потік типу [OutputStream](#).

Приведемо приклад отримання файла з серверу по максимально спрощеному протоколу HTTP.

1. Клієнт посилає серверу запит на отримання файла рядком ["POST filename HTTP/1.1\n\n"](#), де [filename](#) — рядок із шляхом до файла на сервері.
2. Сервер аналізує рядок, відшукує файл з іменем [filename](#) і повертає його клієнту. Якщо ім'я файла [filename](#) закінчується похилою рискою [/](#), то сервер розуміє його як ім'я каталога і повертає файл [index.html](#), що знаходитьться в цьому каталозі.
3. Перед вмістом файла сервер посилає рядок виду ["HTTP/1.1 code OK\n\n"](#), де [code](#) — це код відповіді, одно із чисел: 200 — запит задоволений, файл посилається; 400 — запит не зрозумілий; 404 — файл не знайдено.
4. Сервер закриває сокет і продовжує слухати порт, чекаючи наступного запиту.
5. Клієнт виводить вміст отриманого файла в стандартне виведення [System.out](#) або виводить код повідомлення сервера в стандартне виведення повідомлень [System.err](#).
6. Клієнт закриває сокет, завершуючи звязок.

Цей протокол реалізується в клієнтській програмі лістинга 19.3 і серверній програмі лістинга 19.4.

Лістинг 19.3. Спрощений HTTP-клієнт

```
import java.net.*;
import java.io.*;
import java.util.*;
class Client{
public static void main(String[] args){
if (args.length != 3){
System.err.println("Usage: Client host port file");
System.exit(0) ;
}
String host = args[0];
int port = Integer.parseInt(args[1]);
String file = args[2];
try{
Socket sock = new Socket(host, port);
PrintWriter pw = new PrintWriter(new OutputStreamWriter(
sock.getOutputStream()), true);
pw.println("POST " + file + " HTTP/1.1");
pw.println("Host: " + host);
pw.println("Content-Type: application/x-www-form-urlencoded");
pw.println("Content-Length: 0");
pw.println();
pw.flush();
String response = "";
String line;
while ((line = sock.getInputStream().readLine()) != null)
response += line;
System.out.println(response);
}
catch (IOException e){
System.out.println("Error: " + e);
}
}
}
```

```

pw.println("POST " + file + " HTTP/1.1\n");
BufferedReader br = new BufferedReader(new InputStreamReader(
    sock.getInputStream() ) ) ;
String line = null;
line = br.readLine();
StringTokenizer st = new StringTokenizer(line);
String code = null;
if ((st.countTokens() >= 2) && st.nextToken().equals("POST")){
if ((code = st.nextToken()) != "200") {
System.err.println("File not found, code = " + code);
System.exit (0);
}
}
while ((line = br.readLine()) != null)
System.out.println(line);
sock.close();
}catch(Exception e){
System.err.println(e);
}
}
}
}

```

Закриття потоків введення/виведення викликає закриття сокета. Навпаки, закриття сокета закриває і потоки. Для створення сервера в пакеті `java.net` єсть клас `ServerSocket`. В конструкторі цього класу указується номер порта

ServerSocket(int port)

Основний метод цього класу `accept()` очікує появу запиту. Коли запит є отриманий, метод установлює з'єднання з клієнтом і повертає об'єкт класу `Socket`, через який сервер буде обмінюватися інформацією з клієнтом.

Лістинг 19.4. Спрощений HTTP-сервер

```

import j ava.net.*;
import java.io.*;
import j ava.util.*;
class Server{
public static void main(String[] args){
try{
ServerSocket ss = new ServerSocket(Integer.parseInt(args[0]));
while (true)
new HttpConnect(ss.accept());
}catch(ArrayIndexOutOfBoundsException ae){
System.err.println("Usage: Server port");
System.exit(0);
}catch(IOException e){
System.out.println(e);
}
}
}
class HttpConnect extends Thread{
private Socket sock;
HttpConnect(Socket s) {
sock = s;
setPriority(NORM_PRIORITY - 1);
start();
}
public void run(){

```

```

try{
PrintWriter pw = new PrintWriter(new OutputStreamWriter(
sock.getOutputStream()), true);
BufferedReader br = new BufferedReader(new InputStreamReader(
sock.getInputStream()) );
String req = br.readLine();
System.out.println("Request: " + req);
StringTokenizer st = new StringTokenizer(req);
if ((st.countTokens() >= 2) && st.nextToken().equals("POST")){
if ((req = st.nextToken()).endsWith("/") || req.equals(""))
req += "index.html";
try{
File f = new File(req);
BufferedReader bfr = new BufferedReader(new FileReader(f));
char[] data = new char[(int)f.length()];
bfr.read(data);
pw.println("HTTP/1.1 200 OK\n");
pw.write(data);
pw.flush();
} catch(FileNotFoundException fe){
pw.println("HTTP/1.1 404 Not Found\n");
} catch(IOException ioe){
System.out.println(ioe);
}
} else pw.println("HTTP/1.1 400 Bad Request\n");
sock.close();
} catch(IOException e){
System.out.println(e);
}
}
}
}

```

Спочатку треба запустити сервер, указавши номер порта, наприклад:

Java Server 8080

Потім треба запустити клієнт, указавши IP-адрес або доменне ім'я хоста, номер порта і ім'я файла:

Java Client localhost 8080 Server.java

Сервер відшукує файл **Server.java** в своєму поточному каталозі і посилає його клієнту. Клієнт виводить вміст цього файлу в стандартне виведення і завершує роботу. Сервер продовжує працювати, очікуючи наступного запиту.

Зауваження по налаштуванню

Програми, реалізуючі стек протоколів **TCP/IP**, завжди створюють так звану "петлю" с адресою 127.0.0.1 і доменним ім'ям **localhost**. Це адреса самого комп'ютера. Вона використовується для налаштування додатків клієнт-сервер. Ви можете запускати клієнт і сервер на одній машині, користуючись цією адресою.

Робота по протоколу UDP

Для посилки дейтаграм відправник і отримувач створюють сокети дейта-грамного типу. В **Java** їх представляє клас **DatagramSocket**. В класі три конструктори:

- **DatagramSocket ()** — створюваний сокет приєднується до будь-якого вільного порту на локальній машині;
- **DatagramSocket (int port)** — створюваний сокет приєднується до порту **port** на локальній машині;
- **DatagramSocket(int port, InetAddress addr)** — створюваний сокет приєднується до порту **port**;

аргумент `addr` — одна із адрес локальної машини.

Клас містить масу методів доступу до параметрів сокета і, крім того, методи відправлення і прийому дейтаграм:

- `send(DatagramPacket pack)` — відправляє дейтаграму, упаковану в пакет `pack`;
- `receive (DatagramPacket pack)` — очікує отримання дейтаграми і заносить її в пакет `pack`.

При обміні дейтаграмами з'єднання звичайно не установлюється, дейтаграми посилаються наудачу, в розрахунку на те, що отримувач чекає їх. Але можна установити з'єднання методом

`connect(InetAddress addr, int port)`

При цьому установлюється тільки одностороннє з'єднання з хостом по адресі `addr` і номером порта `port` — або на відправку або на прийом дейтаграм. Потім з'єднання можна разірвати методом `disconnect()`. При посилці дейтаграми по протоколу `JDP` спочатку створюється повідомлення у вигляді масиву байтов, наприклад,

```
String mes = "This is the sending message.";
byte[] data = mes.getBytes();
```

Потім записується адреса — об'єкт класу `InetAddress`, наприклад:

```
InetAddress addr = InetAddress.getByName (host);
```

Потім повідомлення упаковується в пакет — об'єкт класу `DatagramPacket`. При цьому указується масив даних, його довжина, адреса і номер порта:

```
DatagramPacket pack = new DatagramPacket(data, data.length, addr, port)
```

Далі створюється дейтаграмний сокет

```
DatagramSocket ds = new DatagramSocket()
```

і дейтаграма відправляється

```
ds.send(pack)
```

Після посилки всіх дейтаграм сокет закривається, не чекаючи якої-небудь реакції з боку отримувача:

```
ds.close ()
```

Прийом і розпаковка дейтаграм відбувається в зворотному порядку, замість метода `send()` застосовується метод `receive (DatagramPacket pack)`.

В лістингі 19.5 показано приклад класу `Sender`, посилаючого повідомлення, набране в командному рядку, на `localhost`, порт номер 1050. Клас `Recipient`, описаний в лістингі 19.6, приймає це повідомлення і виводить їх в своє стандартне виведення.

Лістинг 19.5. Посилка дейтаграм по протоколу UDP

```
import java.net.*;
import java.io.*;
class Sender{
private String host;
private int port;
Sender(String host, int port){
this.host = host;
this.port = port;
```

```

}
private void sendMessage(String mes) {
try{
byte[] data = mes.getBytes();
InetAddress addr = InetAddress.getByName(host);
DatagramPacket pack =
new DatagramPacket(data, data.length, addr, port);
DatagramSocket ds = new DatagramSocket();
ds.send(pack);
ds.close();
}catch(IOException e){
System.err.println(e);
}
}
public static void main(String[] args){
Sender sndr = new Sender("localhost", 1050);
for (int k = 0; k < args.length; k++)
sndr.sendMessage(args[k]);
}
}

```

Лістинг 19.6. Прийом дейтаграм по протоколу UDP

```

import j ava.net.*;
import java.io.*;
class Recipient{
public static void main(String[] args) {
try{
DatagramSocket ds = new DatagramSocket(1050);
while (true){
DatagramPacket pack =
new DatagramPacket(new byte[1024], 1024);
ds.receive(pack);
System.out.println(new String(pack.getData()));
}
}catch(Exception e){
System.out.println(e);
}
}
}

```