О.М. Ткаченко, В.А. Каплун

Об'єктно-орієнтоване програмування мовою Java

Міністерство освіти і науки України
Вінницький національний технічний університет

О.М. Ткаченко, В.А. Каплун

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ JAVA

Затверджено Вченою радою Вінницького національного технічного університету як навчальний посібник для студентів напрямів підготовки 0804 – "Комп'ютерні науки", 0915 – "Комп'ютерні мережі", 1601 – "Інформаційна безпека". Протокол № 2 від 28 вересня 2006 року

Вінниця ВНТУ 2006

УДК 681.3.07
Д 81

Р е ц е н з е н т и :

А.М. Пєтух,	доктор технічних наук, професор
С.М. Москвіна,	кандидат технічних наук, доцент
С.М. Абрамчук,	кандидат фізико-математичних наук, професор
С.М. Москвіна,	кандидат технічних наук, доцент

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України

Ткаченко О.М., Каплун В.А.
Д 81 **Об'єктно-орієнтоване програмування мовою Java.** Навчальний посібник. – Вінниця: ВНТУ, 2006. – 101 с.

Навчальний посібник присвячений об'єктно-орієнтованій мові програмування Java 2. Послідовно викладаються практичні прийоми роботи з основними конструкціями мови, графічною бібліотекою класів, розширеною бібліотекою Java 2D. Наведені фрагменти програм ілюструють наведені прийоми програмування.

Навчальний посібник призначений для студентів напрямів підготовки 0804 – "Комп'ютерні науки", 0915 – "Комп'ютерні мережі", 1601 – "Інформаційна безпека" для вивчення дисципліни «Засоби програмування».

УДК 681.3.07

ЗМІСТ

ВСТУП

Сучасні критерії якості програм – це, перш за все, надійність, відповідність меті розробки, наявність адекватних реакцій на будь-які дії користувача. Для досягнення цих цілей програма повинна мати просту структуру, бути зручною для здійснення аналізу та модифікації. Цим вимогам відповідає структурне програмування – технологія створення програм, яка дозволяє шляхом дотримання певних правил скоротити термін розробки і істотно зменшити кількість помилок. Структурний підхід до програмування дозволив успішно створювати досить великі проекти, але складність програмного забезпечення продовжувала неупинно зростати. Ідеї структурного програмування отримали свій подальший розвиток в об'єктно-орієнтованому програмуванні – технології, що дозволяє досягнути простоти як структури, так і зручної можливості керування складними програмними системами.

Наразі немає особливої потреби переконувати когось в перевагах об'єктно-орієнтованого підходу до розробки програм. Не тільки створення великих проектів, а навіть розуміння того, як працюють сучасні операційні системи, неможливе без досконалого знання принципів об'єктно-орієнтованого програмування. Більше того, об'єктний стиль розробки програм став своєрідним правилом "хорошого тону" в світі програмістів. Між тим, так було далеко не завжди. До 80-х років авторів цього посібника переконували в тому, що в найближчий час професія програміста стане непотрібною, оскільки ось-ось з'являться технології автоматичного виробництва програм. На щастя для всіх тих, хто отримує насолоду від самого процесу програмування, ці прогнози не справдилися.

Чому ж багато народу з радістю витрачає службовий і вільний час на складання програм? Мабуть, одною з причин є швидкість, з якою в галузі програмування впроваджуються нові ідеї та технології. Завжди на горизонті є щось новеньке.

Мова Java, можливо, якнайкраще ілюструє попереднє твердження. Лише за кілька років Java пройшла шлях від концепції до однієї з найпопулярніших машинних мов. До того ж протягом нетривалого часу Java пройшла дві серйозні ревізії. Перша відбулася, коли на зміну версії 1.0 було випущено версію 1.1. Другу зміну пов'язано з появою Java 2, яка розглядається в цьому посібнику.

Перша робоча версія мови, що носила назву "Oak", з'явилася в компанії Sun Microsystems, Inc в 1991 році. Весною 1995 року, після внесення достатньо суттєвих змін, її було перейменовано в Java. Головним стимулом її створення була потреба в незалежній від платформи мові, яку можна було б використовувати для розробки програмного забезпечення

різноманітних електронних пристроїв: комп'ютерів, мобільних телефонів, мікрохвильових печей тощо. Бурхливий розвиток Інтернет став додатковим фактором, що сприяв популярності Java, оскільки також вимагав розроблення програм, що легко переносяться.

Створення мови Java – це один із найзначніших кроків вперед в області розробки середовищ програмування за останні 20 років. Мова розмітки гіпертексту HTML (Hypertext Markup Language) була необхідна для статичного розміщення сторінок у "Всесвітній павутині" WWW (World Wide Web). Мова Java потрібна була для якісного стрибка в створенні інтерактивних продуктів для Інтернет.

Швидке розповсюдження технології Java пов'язано з тим, що вона використовує нові, спеціально розроблені можливості програмування, хоча і створена на базі мов SmallTalk, Pascal, C++ та інших, увібравши в себе найкращі риси згаданих мов і відкинувши найгірші.

Багато властивостей Java отримала від C і C++. Проектувальники Java свідомо пішли на це, оскільки знали, що знайомий синтаксис робить мову привабливою для легіонів досвідчених програмістів C і C++. В той же час між Java і C++ існують суттєві практичні та філософські відмінності. Неправильно вважати Java і вдосконаленою версією мови C++, розробленою з метою її заміни. Це – різні мови, кожна з яких вирішує своє коло проблем. Принципи об'єктно-орієнтованого програмування, втілені в C++, були розширені та вдосконалені в Java.

Однією з переваг мови Java є можливість працювати не тільки зі статичними текстами і графікою, але й з різноманітними динамічними об'єктами. Крім того, важливою особливістю програмування мовою Java є незалежність розроблених програм від типу комп'ютера, на якому працює користувач. Для Java немає різниці, використовується Macintosh, Pentium, Silicon Graphics чи Sun Workstation – для Java-аплетів це не має ніякого значення. Вони зберігаються на головному вузлі таким чином, що можуть бути запущені для виконання на будь-якому комп'ютері, де встановлено браузер, який підтримуватиме Java.

Ще один важливий аргумент на користь Java – відсутність потреби в наявності ліцензії на її використання. Хоча сторонні виробники пропонують платні інструментальні середовища для розробки та відлагодження Java-програм, на Webсервері фірми Sun *http://java.sun.com* завжди можна знайти та завантажити "рідний" безкоштовний варіант компілятора Java (Java Development Kit) разом з усім необхідним для створення програм.

1 ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ JAVA

1.1 Загальні принципи об'єктно-орієнтованого програмування

Перш ніж розпочати вивчення мови Java, необхідно розглянути, що таке об'єктно-орієнтоване програмування. Як ми знаємо, всі комп'ютерні програми складаються з двох елементів: коду та даних. Відповідно будь-яка програма може бути концептуально організована або навколо її коду, або навколо її даних. Іншими словами, деякі програми концентрують свою увагу на тому, "що робиться з даними", а інші – на тому, "на що цей процес впливає".

Існує декілька парадигм (ключових підходів), які управляють конструюванням програм. Перший підхід вважає програму **моделлю, орієнтованою на процес** (process-oriented model). При цьому програму визначають послідовності операторів її коду.

Перші компілятори (наприклад, FORTRAN) підтримували *процедурну модель програмування,* в основі якої лежить використання функцій. Наступний етап розвитку пов'язаний з переходом до *стуктурної моделі програмування* (до неї відносяться компілятори ALGOL, Pascal, C), в основі якого лежать такі положення: програми подаються у вигляді сукупності взаємопов'язаних процедур і даних, якими ці процедури (або блоки) оперують. Тут широко використовуються процедурні блоки і мінімальне використання GOTO. Ці програми є більш прості. Такі мови програмування, як Pascal, C успішно використовують цю модель. Але тут часто виникають проблеми, коли зростає розмір і складність програм.

Інший підхід, названий **об'єктно-орієнтованим програмуванням** (ООП), було створено для управління зростаючою складністю програм. ООП організовує програму навколо її даних (тобто навколо об'єктів) і набору чітко визначених інтерфейсів з цими даними. Об'єктно-орієнтовану програму можна характеризувати як *доступ до коду, що управляється даними* (data controlling access to data). Як ми побачимо далі, такий підхід має деякі організаційні переваги.

1. Можна повторно використовувати код програми і таким чином економити час на розробку.
2. Програми з використанням ООП добре структуровані, що дозволяє добре розуміти, які функції виконують окремі підпрограми.
3. Програми з використанням ООП легко тестувати і модифікувати. Можна розбити програму на компоненти і тестувати роботу кожної з них.

Всі мови ООП забезпечують механізми, які допомагають реалізувати об'єктно-орієнтовану модель. До них відносять абстракцію, інкапсуляцію, успадкування і поліморфізм. Також їх часто називають основними принципами ООП.

Абстракція та інкапсуляція

Абстракція даних – введення типів даних, визначених користувачем і відмінних від базових. Ця концепція полягає у можливості визначати нові типи даних, з якими можна працювати так само, як і з основними типами даних. Крім того, абстракція має місце і при застосуванні шаблонів, тобто введенні абстрактних типів даних, які в залежності від умов їх застосування приймають той або інший тип.

Інкапсуляція – це механізм, який пов'язує код з даними, що ним обробляються, та зберігає їх як від зовнішнього впливу, так і від помилкового використання. Інкапсуляцію можна уявити як захисну оболонку, яка запобігає доступу до коду та даних з іншого коду, що знаходиться зовні цієї оболонки. Доступ до коду та даних всередині оболонки відбувається через чітко визначений інтерфейс. Потужність такого підходу полягає у тому, що кожен знає, як отримати доступ до інкапсульованого коду, і може користуватися ним незалежно від деталей його реалізації та без побоювання несподіваних наслідків.

Основою абстракції та інкапсуляції в Java є **клас**. Клас визначає структуру та поведінку (дані і код) деякого набору об'єктів. Кожен **об'єкт** заданого класу містить як структуру (дані), так і поведінку, що визначається класом (так, як би ці об'єкти було проштамповано шаблоном у формі класу). Тому об'єкт іноді ще називають **екземпляром класу**. Таким чином, клас – це логічна конструкція, а об'єкт – це фізична реальність.

При створенні класу необхідно специфікувати код і дані, що складають цей клас. Разом всі ці елементи називають *членами* (members) класу. Дані, що визначаються в класі, називають *змінними-членами* (member variables) або *полями* класу. Код, який оперує цими даними (тобто функції, що знаходяться всередині класу), називають *методами-членами* (member methods), або просто *методами*. В правильно записаних Java-програмах методи визначають, як можна використовувати змінні-члени. Отже, поведінку та інтерфейс класу визначають методи, що оперують даними його екземплярів.

Оскільки мета створення класу – це інкапсуляція складності, існують механізми приховування її реалізації всередині класу.

Методи і поля класу можуть бути помічені як *private* (приватний, локальний) або *public* (загальний). Модифікатор *public* вказує на все, що потрібно знати зовнішнім користувачам класу. Методи та поля з модифікатором *private* є доступними лише для методів даного класу. Будь-який код, що не є членом даного класу, не має доступу до *private*-методів і полів. Оскільки *private*-члени класу є доступними для інших частин програми тільки через *public*-методи класу, можна бути впевненим, що ніякі непотрібні дії не виконуються.

Успадкування

Успадкування (наслідування) – це процес, за допомогою якого один об'єкт отримує властивості іншого об'єкта. Воно важливе, тому що підтримує концепцію ієрархічної класифікації.

Переважною частиною знань можна управляти лише за допомогою ієрархічних (тобто організованих "згори донизу") класифікацій. Без застосування класифікацій кожен об'єкт потребував би явного визначення всіх своїх характеристик. Завдяки використанню успадкування об'єкт потребує визначення лише тих якостей, які роблять його унікальним у власному класі. Тому саме механізм успадкування дає можливість одному об'єкту бути специфічним екземпляром загального випадку.

У навколишньому світі ми бачимо об'єкти, пов'язані між собою ієрархічно, наприклад, тварини, ссавці, собаки. Якщо ми хочемо описати тварини абстрактним чином, нам необхідно визначити деякі їх **атрибути,** наприклад, розмір, вага, стать, вік тощо. Тваринам також притаманна певна **поведінка** – вони їдять, дихають, сплять. Такий опис атрибутів і поведінки і визначає клас тварин.

Якщо ми захочемо описати більш специфічний клас тварин, такий як ссавці, вони повинні були б мати більш специфічні атрибути, такі як тип зубів і молочні залози. Такий клас відомий як **підклас** (успадкований клас, похідний клас, дочірній клас) тварин, тоді як сам по собі клас тварин називають **суперкласом** (батьківським класом, класом-пращуром, базовим класом).

Оскільки ссавці – це більш точно специфіковані тварини, то говорять, що вони успадковують всі атрибути тварин. Підклас, що знаходиться на більш глибокому рівні ієрархії, успадковує всі атрибути кожного свого батьківського класу в ієрархії класів.

Успадкування щільно пов'язане з інкапсуляцією. Якщо даний клас інкапсулює деякі атрибути, то будь-який підклас буде мати ті ж самі атрибути плюс атрибути, які він додає як частину своєї спеціалізації. Це ключова концепція, яка дозволяє об'єктно-орієнтованим програмам зростати за складністю в арифметичній, а не геометричній прогресії. Новий підклас успадковує всі атрибути всіх своїх батьків. Він не має непередбачуваних зв'язків з рештою коду в програмі.

Крім того, у похідному класі успадковані функції можуть бути перевизначені. Таким чином будують ієрархії класів, пов'язаних між собою. Якщо об'єкт наслідує свої атрибути від одного базового класу, це буде *просте успадкування*. Якщо об'єкт успадковує атрибути від кількох батьків, це буде *множинне успадкування*. Але у мові Java множинне успадкування у безпосередньому вигляді не існує, його механізм є дещо специфічним.

Поліморфізм

Поліморфізм (грецькою ”polymorphos” – множинність форм) – властивість, яка дозволяє використовувати один інтерфейс для спільного класу дій. Специфічна дія точно визначається в залежності від конкретної ситуації. Наприклад, розглянемо стек (список типу LIFO – Last-In, First-Out; останнім увійшов, першим вийшов).

Програма може потребувати три типи стеків. Один стек використовується для цілих чисел, другий – для чисел з плаваючою точкою, третій – для символів. Алгоритм, який реалізує кожен стек, один і той самий, хоча дані, що зберігаються, різні. Не об’єктно-орієнтована мова вимагала б створення трьох різних стекових підпрограм, кожна з яких мала б своє власне ім’я. Завдяки поліморфізму в мові Java можна визначити спільний для всіх типів даних набір стекових підпрограм, що використовують одне і те саме ім’я.

Взагалі суть концепції поліморфізму можна виразити фразою “один інтерфейс, багато методів”. Це означає, що можна спроектувати спільний інтерфейс для групи пов’язаних родинними зв’язками об’єктів. Це дозволяє зменшити складність, припускаючи використання одного й того самого інтерфейса для загального класу дій. Задача компілятора – вибрати специфічну дію (тобто метод) для його використання в кожній конкретній ситуації. Програміст не повинен робити це “вручну”. Йому необхідно тільки пам’ятати та використовувати спільний інтерфейс.

Отже, поліморфізм – це властивість програмного коду поводитись по-різному в залежності від ситуації, що виникає в момент виконання.

Якщо провести аналогію з собакою, можна сказати, що нюх у собаки поліморфний. Якщо він чує кицьку, то гавкає та біжить за нею. Якщо чує їжу, виділяє слину та біжить до миски. В обох ситуаціях працює одне й те саме чуття – нюх. Різниця полягає в тому, що саме він нюхає, тобто в типі даних, з якими оперує ніс собаки. Ту ж спільну концепцію реалізовано в мові Java відносно методів у Java-програмах.

При правильному застосуванні наведені принципи ООП – абстракція, інкапсуляція, поліморфізм та успадкування – взаємодіють таким чином, щоб створити деяке середовище програмування, яке має забезпечити більш стійкі та масштабовані програми порівняно з моделлю, орієнтованою на процеси. Вдало спроектована ієрархія класів є базисом повторного використання коду, для створення та тестування якого було витрачено чимало часу та зусиль. Інкапсуляція дозволяє реалізаціям подорожувати в часі без руйнування коду, доступ до якого здійснюється з допомогою *public*-інтерфейса класів. Поліморфізм дозволяє створювати ясний та читабельний код.

Контрольні питання

1. *Назвіть відомі моделі програмування і охарактеризуйте їх.*
2. *Які переваги має об'єктно-орієнтована модель програмування?*
3. *Що спільного та відмінного в класах і об'єктах? Наведіть приклади класів і об'єктів.*
4. *Назвіть елементи, що входять до складу класів та поясніть різницю між ними.*
5. *Назвіть основні принципи об'єктно-орієнтованого програмування і охарактеризуйте їх.*
6. *У чому полягають взаємодії між основними принципами ООП?*

1.2 Основні поняття мови JAVA. Перша програма

Мова Java – це об'єктно-орієнтована мова програмування, що бере свою історію від відомої мови C++. Але, на відміну від останньої, Java є мовою, що інтерпретується. Програми, написані на ній, здатні працювати в різних місцях мережі і не залежать від платформи, на якій виконуються написані нею додатки. Java свідомо уникає арифметики з покажчиками й іншими ненадійними елементами, якими вирує C++, тому, розробляючи на ній додатки, ми позбавляємося багатьох проблем, звичайних при створенні програмного забезпечення.

Для відлагодження програм мовою Java підійде будь-який з пакетів: Microsoft Visual J++, Symantec Cafe, Java Add-On зі складу Borland C++ 5.0 чи Sun Java WorkShop. Якщо є бажання користуватися командними файлами, то можна завантажити з Web-сервера *http://java.sun.com* "рідний" варіант компілятора Java компанії Sun – Java Development Kit (JDK).

У термінах мови Java маленький додаток, що вбудовується в сторінку Web, називається **аплетом**. Власне кажучи, створення аплетів − основне застосування для Java. Аплети набули собі звання справжніх прикрас для Web. Аплет може бути і вікном анімації, і електронною таблицею, і усім, що тільки можна собі уявити. Але це не означає, що мовою Java не можна писати нормальні додатки з вікнами. Ця мова програмування спочатку була створена для звичайних додатків, що виконуються в Інтернет і в інтрамережах, і вже потім стала використовуватися для розробки аплетів.

Елементарні будівельні блоки в Java називаються **класами** (як і в C++). Клас складається з даних і коду для роботи з ними. У засобах для розробки мовою Java усі стандартні класи, доступні програмісту, об'єднані для зручності в **пакети** − ще одні елементарні блоки Java-програм.

От найпростіша програма, що наводиться в багатьох підручниках є Java:

```
class Hello
```

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

На рисунку 1 схематично подані основні етапи створення додатка за допомогою стандартних засобів JDK. Запустимо компілятор Java з назвою *javac* і отримаємо готовий клас Java – Hello.class:

javac Hello.java

Компілятор javac генерує окремий файл для кожного класу, визначеного у файлі вихідного тексту, незалежно від кількості файлів вихідного тексту.

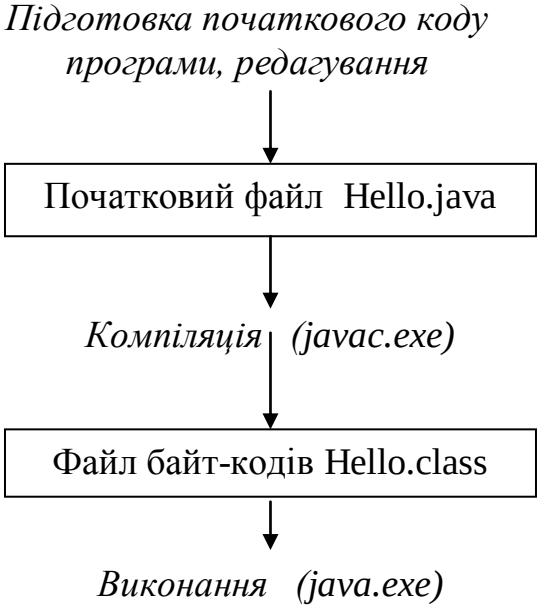

Рисунок 1 – Редагування, компіляція та запуск на виконання Java-програми

Примітка 1. Java не є чистим інтерпретатором, як, наприклад, Basic. В результаті компіляції вихідного тексту програми створюється проміжний файл з розширенням .class, що містить так званий байт-код. Таким чином досягається компроміс між ефективністю виконання Java-програм та їх незалежністю від платформи.

Якщо хочемо подивитися, як цей додаток працює, виконаємо його за допомогою команди

java Hello.

При цьому необхідно набрати ім'я класу, що запускається, точно так, як воно написано у вихідному тексті програми, тобто з дотриманням регістра, інакше ви одержите повідомлення про помилку.

Примітка 2. При компіляції Java-програми ім’я файла вказується з розширенням, при запуску на виконання – без розширення.

Розглянемо поелементно вихідний текст нашого прикладу. Вся програма складається з одного класу з ім'ям Hello. У цьому класі є єдиний метод *main(),* аналогічний функції *main()* у мовах програмування С і С++, який і визначає місце, з якого програма починає виконуватися (так звана точка входу). Модифікатор доступу *public* перед ім'ям методу *main()* вказує на те, що цей метод доступний усім класам, що бажають його викликати, незалежно від прав доступу і від місця їхнього розташування. Модифікатор *static* говорить про те, що для всіх екземплярів класу Hello і в наслідуваних від нього класах існує лише один метод *main()*, розділений між усіма класами, що, можливо, будуть успадковані від Hello. Це допомагає уникнути появи безлічі точок входу в програму, що викликає помилку.

Через змінну-масив *args* типу *String* (рядок) передаються параметри командного рядка класу. У Java перший елемент списку параметрів відповідає першому параметру, а не імені програми, що запускається, як це прийнято в мовах С і С++. Доступ до нього можна здійснити через вираз *args[0]*. Рядок *System.out.println ("Hello, World!")* посилає рядок тексту в стандартний потік виведення, тобто на екран. Ми відправляємо повідомлення стандартному класу *System*, що відповідає за основні системно-незалежні операції, наприклад, виведення тексту на консоль. А вже з цього класу ми викликаємо клас стандартного потоку виведення. Слідом йде виклик методу *println()*, що, власне, і відображає рядок тексту на екрані монітора, після завершення чого переводить курсор на наступний рядок.

У Java **всі методи класу описуються тільки всередині цього класу**. Таким чином, відпадає необхідність у пересуванні по тексту в пошуку методів класів.

Примітка 3. Оброблені класи компілятор записує в окремі файли. Так, якщо ми опишемо в одному вихідному файлі відразу кілька класів, то в результаті компіляції одержимо декілька файлів з розширенням .class, по одному для кожного класу, і кожний з них буде мати те ж ім'я, що і відповідний клас.

Якщо засвоїти вміст пакета Java з ім'ям *java.awt*, що розшифровується як Abstract Windowing Toolkit (набір абстрактної роботи з віконною системою), то відкриються незлічимі можливості щодо створення інтерфейсів і віконної графіки. Цей пакет забезпечує машинно-незалежний інтерфейс керування віконною системою будь-якої віконної операційної системи. До складу *java.awt* входять більше 40 класів, що відповідають за елементи графічного середовища користувача (GUI). В основному awt застосовується при написанні аплетів для сторінок Web. При перегляді сторінки на Web-сервері аплет передається на машину користувача, де і запускається на виконання.

Тепер ми можемо розглянути аплет, який робить те саме, що і вже розглянутий раніше приклад, тобто виводить рядок на екран:

```
import java.awt.*;
public class Hello extends java.applet.Applet
{
        public void init() {}
        public void paint(Graphics g)
        {
                g.drawString("Hello, Java!",20,30);
        }
}
```

Першим рядком в аплет включаються всі необхідні класи з пакета *java.awt*, про який ми тільки що говорили. Ключове слово *import* має приблизно те ж значення, що й оператор *#include* мов C і C++. Далі слідує опис класу нашого аплета, якому передує модифікатор доступу *public*. Його задача − дати можливість використовувати наш клас ззовні, тобто запускати його із зовнішніх програм. Якщо цього слова не буде, компілятор видасть повідомлення про помилку, указавши, що аплету потрібно опис інтерфейсу доступу. Далі йде ключове слово *extends* і назва класу. Так у Java позначається процес успадкування. Цим словом ми вказуємо компілятору успадкувати (розширити) стандартний клас *java.applet.Applet*, відповідальний за створення і роботу аплета. Метод *init()* викликається в процесі ініціалізації аплета. Зараз цей метод порожній, але, згодом, можливо ми скористаємося ним для своїх цілей. За відображення рядка відповідає інший метод – *paint()*. Він викликається в той момент, коли потрібно перерисувати дані на екрані. Тут за допомогою методу *drawString()* стандартного класу *Graphics* рисується рядок "Hello, Java!" з екранними координатами (20, 30).

Контрольні питання

1. *Назвіть основні етапи створення додатків мовою Java.*
2. *В чому полягає відмінність аплетів від додатків?*
3. *Як можна передати список параметрів з операційного середовища в Java-програму?*

1.3 Типи даних

Можна було б очікувати, що в об’єктному світі Java всі типи даних належать деякому класу. Але розробники Java дещо відійшли від такого ортодоксального підходу і залишили майже незмінними стандартні типи даних мови C++, назвавши їх базовими. Інша категорія – об’єктні типи даних, до яких належать класи, масиви й інтерфейси. Звичайно, основну

увагу ми будемо приділяти саме об’єктним типам, але перед усім коротко опишемо базові типи даних.

Базові типи даних

Ідентифікатори мови Java повинні починатися з букви будь-якого регістра або символів "_" і "$". Далі можуть йти і цифри. Наприклад, _Java - правильний ідентифікатор, а 1_$ - ні. Ще одне обмеження Java виникає з його властивості використовувати для збереження символів кодування Unicode, тобто можна застосовувати тільки символи, що мають порядковий номер більший 0x0 у розкладці символів Unicode.

Коментарі. У стандарті мови Java існує три типи коментарів:

```
/*Comment*/
// Comment
/** Comment*/
```

Перші два являють собою звичайні коментарі, застосовувані як у Java, так і в C++. Останній − особливість Java, введена в цю мову для автоматичного документування. Після написання вихідного тексту утиліта автоматичної генерації документації збирає тексти таких коментарів в один файл.

Цифрові літерали схожі з аналогічними в мові C++. Правила для цілих чисел прості:

- якщо в цифри немає суфікса і префікса, то це *десяткове число*;
- у *вісімкових числах* перед цифрою стоїть нуль;
- для *шістнадцяткових чисел* префікс складається з нуля і букви X (0x або 0X).
- при доданні до цифри букви L числу присвоюється тип *long (довге ціле)***.**

Приклади: 23 (десяткове),
0675 (вісімкове),
0x9FA (шістнадцяткове),
456L (довге ціле).

Числа із плаваючою точкою. Для них передбачено два види описів:

- звичайний і
- експонентний.

При звичайному описі числа з плаваючою точкою записуються в такій же формі, як і ті числа, що ми пишемо на папері від руки: 3.14, 2.73 і т.д. Це ж стосується і експонентного формату: 2.67E4, 5.0E-10. При доданні суфіксів D і F виходять числа типів *double* і *float*. Наприклад, 2.71D і 0.981F.

Цілочислові типи. У мові Java з'явився новий 8-бітовий тип *byte*. Тип *int*, на відміну від аналогічного в C++, має довжину 32 біти. А для 16-бітових чисел передбачений тип *short*. У відповідності з усіма цими змінами тип *long* збільшився, ставши 64-бітовим.

Числові типи даних наведено в таблиці 1:

Таблиця 1 – Числові типи даних

Тип	Форма подання	Значення за замовчу-ванням	Довжина (в бітах)	Максимальне значення
byte	Ціле число зі знаком	0	8	127
short	Ціле число зі знаком	0	16	32767
int	Ціле число зі знаком	0	32	2147483647
long	Ціле число зі знаком	0	64	порядку 10^{18}
float	Число з плаваючою точкою	0	32	порядку 10^{38}
double	Число з плаваючою точкою	0	64	порядку 10^{308}

Примітка. Всі числові типи в Java є знаковими.

У стандарт Java був введений тип *boolean*, якого так довго чекали програмісти, що використовують C++. Він може приймати лише два значення: *true і false.*

У порівнянні з C++ **масиви** Java перетерпіли значні зміни. По-перше, змінилися правила їхнього опису. Масив тепер може бути описаний двома такими способами:

type name[];

type[] name;

При цьому масив не створюється, а лише описується. Отже, для резервування місця під його елементи треба скористатися динамічним виділенням за допомогою ключового слова *new*, наприклад:

char[] arrayName;

arrayName[] = new char[100];

або сполучити опис масиву з виділенням під нього пам'яті:

char array[] = new char[100];

Багатовимірних масивів у Java немає, тому доводиться вдаватися до хитрощів. Наприклад, створити багатовимірний масив можна як масив масивів:

float matrix[][] = new float[5][5];

Класи

Говорячи про класи, необхідно ще раз пригадати один з трьох основних принципів ООП – успадкування. Використовуючи його, можна створити головний клас, який визначає властивості, спільні для набору елементів. Надалі цей клас може бути успадкований іншими, більш

специфічними класами. Кожен з них додає ті властивості, які є унікальними для нього. В термінології Java клас, який успадковується, називається **суперкласом** (superclass). Клас, який виконує успадкування, називається **підкласом** (subclass). Тому підклас – це спеціалізована версія суперкласу. Він успадковує всі поля та методи суперкласу, та додає до них свої власні унікальні елементи. Щоб успадкувати клас, необхідно просто ввести означення одного класу в інше, використовуючи ключове слово *extends*.

Розглянемо тепер, як описуються основні базові будівельні блоки мови Java − класи. Схема синтаксису опису класу така:

```
[Модифікатори] class Ім'яКласу [extends Ім'яСуперкласу]
                                [implements ІменаІнтерфейсів]
{
  Дані класу;
  Методи класу;
}
```

де *Модифікатори* − ключові слова типу *public, private* і т.д., що модифікують поведінку класу за замовчуванням;

Ім'яКласу − ім'я, що ви привласнюєте класу;

Ім'яСуперкласу − ім'я класу, від якого успадковується ваш клас;

ІменаІнтерфейсів − імена інтерфейсів, що реалізуються даним класом (про це в наступному розділі).

Типовий приклад класу ми вже наводили раніше. Це клас аплета, що виводить рядок на екран.

Модифікатори доступу

Модифікатори визначають способи подальшого використання класу. При розробці самого класу вони не мають особливого значення, але вони дуже важливі при створенні інших класів або інтерфейсів на базі даного класу. Модифікаторів доступу є три плюс ще один за замовчуванням.

public – класи *public* доступні для всіх об'єктів незалежно від пакета, тобто повністю не захищені. *public*-класи мають знаходитися в файлах з іменами, що збігаються з іменами класів.

friendly – значення за замовчуванням (тобто слово *friendly* в описі класу ніколи не пишеться!). *friendly*-класи доступні лише для об'єктів, що знаходяться в тому ж самому пакеті, що і даний клас, незалежно від того, чи є вони нащадками даного класу.

final-класи не можуть мати підкласів-нащадків. Тим самим ми втрачаємо одну з головних переваг ООП. Але іноді треба заборонити іншим класам змінювати поведінку розроблюваного класу (наприклад, якщо даний клас буде використаний як стандарт для обслуговування мережних комунікацій).

abstract **–** це клас, в якому є хоча б один абстрактний метод, тобто метод, для якого описаний лише заголовок (прототип функції), а саме тіло методу відсутнє. Зрозуміло, що реалізацію цього відсутнього методу покладено на класи-нащадки. Наприклад, створюється деякий клас для перевірки орфографії. Замість того, щоб закладати в нього перевірку української, російської, англійської орфографії, створюючи окремі методи *ukraineCheck(), russianCheck()* і т. д. можна просто створити абстрактний метод *grammarCheck()*, перекладаючи роботу щодо перевірки конкретної граматики на класи-нащадки (які, можливо, будуть створені іншими фахівцями).

Суперкласи (батьківські класи)

Можливість створення класів-нащадків – одна з головних переваг ООП. Для того, щоб використати вже існуючий клас, слід указати в об'яві класу слово *extends* (розширює). Наприклад,

public class MyClass extends Frame.

В основі ієрархії в java знаходиться клас *Object*. Тому якщо навіть не використовується слово *extends* в описі класу, то створюється нащадок класу *Object*.

Нагадаємо, що клас-нащадок успадковує всі дані та методи суперкласу.

В мові Java відсутнє множинне успадкування (але є інтерфейси, які з успіхом замінюють його).

Чому в Java немає множинного успадкування? Це не випадково, тут спостерігається певна система. В мові Java розробники вирішили позбавитись від всього, що могло викликати проблеми в C++, наприклад, покажчики, множинна спадковість. Про проблеми, що пов'язані з множинною спадковістю, можна прочитати в [1].

Конструктори. Створення екземплярів класу

Конструктор – це метод класу, що має особливе призначення. Зокрема він використовується для установлення деяких параметрів (наприклад, ініціалізації змінних) та виконання певних функцій (наприклад, виділення пам'яті).

Конструктор має те ж саме ім'я, що і клас. Наприклад,

MyClass(String name) {myName=name;}

Як відрізнити – це клас чи конструктор?

MyClass *MyClass()*

Так само, як в C++, один клас може мати декілька конструкторів. У цьому випадку вони мають відрізнятися за своїми параметрами. Також в описі конструктора можна використовувати модифікатори доступу, але не всі. Найчастіше конструктори роблять *public*.

Щоб використати розроблений клас, треба створити екземпляр класу, тобто об'єкт, що має тип даного класу. Створимо екземпляр класу *MyClass*:

MyClass myClass1= new MyClass();

Екземпляр класу може бути полем іншого класу. Так утворюється ієрархія за складом (спадковість реалізує ієрархію за номенклатурою).

```
public class Checker
{
        MyClass myClass1= new MyClass();
        int a=5;
        . . .
}
```

Різниця між об'явою базового типу (наприклад, *int*) та створенням об'єкта – використання ключового слова *new*. При цьому відбувається:

1) виділення пам'яті під об'єкт;
2) виклик конструктора;
3) повернення посилання на об'єкт і присвоюється змінній *myClass1*.

Тому не зовсім коректно говорити, що конструктор нічого не повертає – він повертає посилання на об'єкт. Правильно – в описі конструктора відсутній тип даних, що повертається.

У чому полягає відмінність об'єктів від змінних базових типів? До базових типів завжди звертаються за значенням. До об'єктних типів (до яких відносяться масиви, класи, інтерфейси) – завжди за посиланням. Проілюструємо цю різницю на прикладі.

```
int  x=5;
int  y=x;
y++;
```

Чому дорівнює *x*? Звичайно 5, оскільки зміна значення *y* ніяк не впливає на значення *x*.

```
MyClass w = new MyClass();
MyClass z = w;
w.f = 5;
z.f = 6;
```

Чому дорівнює *w.f* ? Відповідь 6, оскільки і *z*, і *w* посилаються на одну і ту саму ділянку пам'яті. Це ілюструється на рисунку 2.

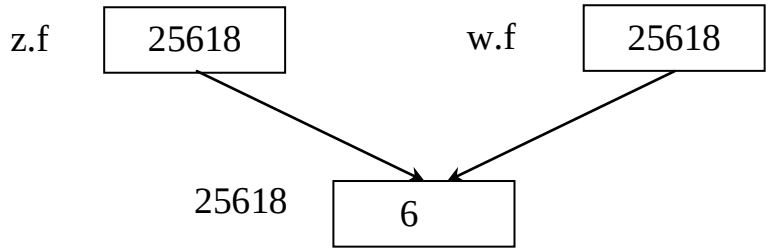

Рисунок 2 – Звертання до об’єктних типів

Контрольні питання

1. *Наведіть порівняльну характеристику стандартних типів даних мови С і базових типів даних мови Java.*
2. *Чи можуть модифікатори доступу з’єднуватися між собою та які саме?*
3. *Назвіть елементи, що належать до об’єктних типів даних в мові Java.*
4. *Чим відрізняється звертання до об’єктних типів від звертання до базових типів?*

1.4 Поля і методи класів

В класах існують два типи змінних: одні належать самому класу, інші ж – різним методам класу.

Ті змінні, що описані не в методах, але всередині даного класу, називають **полями** класу. Вони доступні для всіх методів даного класу (це як мінімум).

Крім того, можна описувати змінні всередині методу класу. Такі змінні є локальними для методу і доступні тільки для цього методу.

Загальне правило: кожна змінна доступна лише всередині того блока (обмеженого фігурними дужками), в якому її описано.

І останнє. Як поля, так і локальні змінні, можуть бути як базового типу, так і екземплярами класу.

Модифікатори доступу до членів класу

В таблиці 2 наведено правила доступу до полів і методів, описаних за допомогою різних модифікаторів. Елемент, описаний *public*, доступний з будь-якого місця. Все, що описано *private*, доступно лише всередині класу. Якщо у елемента взагалі не указаний модифікатор рівня доступу, то такий елемент буде видно з підкласів-нащадків і класів того ж пакета. Саме цей рівень доступу використовується в Java за замовчуванням. Якщо треба зробити, щоб елемент крім того був доступний ззовні пакета, але тільки підкласам того ж класу, якому він належить, його слід описати як *protected*. І, насамкінець, якщо треба щоб елемент був доступний тільки

підкласам, незалежно від того, чи знаходяться вони в даному пакеті, слід використовувати комбінацію *private protected*.

Таблиця 2 – Доступ до полів і методів для різних модифікаторів

	private	friendly (модифікатор відсутній)	private protected	protected	public
один і той же клас	+	+	+	+	+
підклас в тому ж пакеті	-	+	+	+	+
незалежний клас в тому ж пакеті	-	+	-	+	+
підклас в іншому пакеті	-	-	+	+	+
незалежний клас в іншому пакеті	-	-	-	-	+

З полями та методами можуть використовуватись всі специфікатори доступу, з класами – лише *public* і значення за замовчуванням.

Інші модифікатори доступу

Модифікатор ***static.*** Цей опис може використовуватись як з полями, так і з методами. Для поля класу опис *static* означає, що таке поле створюється в єдиному екземплярі незалежно від кількості об'єктів даного класу (звичайні поля – для кожного екземпляру класу). Статичне поле існує навіть тоді, коли немає жодного екземпляра класу. Статичні поля розташовуються Java-машиною окремо від об'єктів класу в момент першого звертання до цього класу. Розглянемо приклад:

```
public class Exam
{
        int a = 10;             // звичайне поле
        static int cnt = 0;  // статичне поле

        public void print()
        {
                System.out.println("cnt=" + cnt);
                System.out.println("a=" + a);
         }

        public static void main(String args[])
        {       Exam obj1 = new Exam();
                cnt++;        // збільшуємо cnt на 1
```

```
            obj1.print();
            Exam obj2 = new Exam();
            cnt++;          // збільшуємо cnt на 1
            obj2.a = 0;
            obj1.print();
            obj2.print();
        }
}
```

У цьому прикладі поле *cnt* є статичним. На екрані для обох об'єктів буде виведено одне й те саме значення поля *cnt*. Це пояснюється тим, що воно існує в одному екземплярі.

За аналогією зі статичними полями статичні методи не прив'язані до конкретного об'єкта класу. Коли викликається статичний метод, перед ним можна вказувати не посилання, а ім'я класу. Наприклад:

```
class SomeClass
{       static int t = 0;   // статичне поле
        . . .
        public static void f()
        {       // статичний метод
                . . .
        }
        public void g()
        {       // звичайний метод
                . . .
        }
}
   . . .
   SomeClass MyClass1 = new SomeClass();
   MyClass1.g();
   SomeClass.f();
. . .
```

І, насамкінець, оскільки статичний метод не пов'язаний з конкретним об'єктом, всередині такого методу не можна звертатися до нестатичних полів класу без посилання на об'єкт перед ім'ям поля (практично це означає, що такому методу треба передавати як параметр посилання на об'єкт). До статичних полів класу такий метод може звертатися вільно.

Модифікатор ***final.*** Цей опис може використовуватись як з полями, так і з методами. Для полів – це простий засіб створення констант (в Java відсутня директива *#define*). Наприклад,

```
final int SIZE = 5;
```

За замовчуванням константи записуються великими літерами. Крім того, для економного використання пам'яті константи звичайно роблять статичними.

Що стосується *final*-методів, то це такі методи, які не можуть бути перевизначеними в класах-нащадках.

Передача параметрів в Java

Як відомо, існують два способи передачі параметрів:

- за значенням;
- за посиланням.

Правило таке: якщо передається базовий тип (*int, char*), то результат передається за значенням. Якщо передається об'єкт (наприклад, клас), то він передається за посиланням.

Контрольні питання

1. *Як трактується в Java змінна, у якої не вказано модифікатор доступу?*
2. *В якому випадку перед ім'ям методу вказують не посилання на об'єкт, а ім'я класу?*
3. *Як в Java утворюються константи?*

1.5 Пакети та інтерфейси

Уявіть ситуацію, що ви спеціалізуєтеся на анімації Web-сторінок. Ви створили власний клас *Animator* і надалі вставляєте цей клас в усі свої програми. Щоб кожен раз не копіювати текст з одного файла в інший ви можете винести цей клас в окремий файл (за бажанням можна зробити його *public*). Якщо ви хочете тепер скористатися цим класом в іншій програмі, ви маєте імпортувати його, так саме, як і стандартні класи:

```
import  Animator;
```

Тепер уявіть ситуацію, що ви пишете велику програму, яка складається з багатьох класів. Оскільки кожен *public*-клас має знаходитися в окремому файлі, виникає необхідність якимось чином упорядкувати розміщення цих файлів. І такий спосіб існує.

Для упорядкування файлів класів в Java використовуються **пакети.** Кожен пакет можна розглядати як підкаталог на диску. Пакети – це набори класів. Вони нагадують бібліотеки, які існують на багатьох мовах. Звичайно пакети містять класи, логічно пов'язані між собою.

Щоб внести клас в пакет, необхідно скористатися оператором *package*, наприклад:

package game;

При цьому необхідно витримати дві умови:

1. Вихідний текст класу має знаходитись в тому ж каталозі, що і інші файли пакета (тобто цей каталог треба попередньо створити).
2. Оператор *package* має бути першим оператором в файлі.

Якщо файл є частиною пакета, справжнє ім'я класу складається з імені пакета, точки та імені класу. Тому, якщо ми внесли клас *Animator* в пакет *game*, щоб імпортувати його слід вказати повне ім'я класу:

import game.Animator;

Якщо ж вам необхідно використати в своїй програмі декілька класів з пакета *game* і ми не бажаємо імпортувати кожен клас окремо, виписуючи всі їх імена, ми можете імпортувати весь пакет, отримавши доступ одразу до всіх класів даного пакета:

import game.;*

І насамкінець. Навіть нічого не імпортуючи, ми можемо отримати доступ до деякого класу з пакета, вказуючи його повне ім'я при оголошенні екземпляра класу:

game.Animator a;

Загальні відомості про інтерфейси

Інтерфейси – це варіант множинного успадкування, яке є в C++, але відсутнє в Java. Іншими словами, клас в Java не може успадковувати поведінку одразу кількох класів, але може реалізовувати одразу декілька інтерфейсів. Також клас може бути одночасно і нащадком одного класу, і реалізовувати один або кілька інтерфейсів.

В чому відмінність інтерфейсів від класів? Класи описують об'єкт, а інтерфейси визначають набір методів і констант, які реалізуються іншим об'єктом.

Інтерфейси мають одне головне обмеження: вони можуть описувати абстрактні методи та поля *final*, але не можуть мати жодної реалізації цих методів. В прикладному відношенні інтерфейси дозволяють програмісту визначити деякі функціональні характеристики, не турбуючись про те, як потім ці характеристики будуть описані.

Наприклад, якщо деякий клас реалізує інтерфейс *java.lang.Runnable* він має містити метод *run()*. Тому java-машина може «всліпу» викликати метод *run()*, для будь якого *Runnable*-класу. Неважливо, які дії він при цьому виконує, важливо, що він є.

Створення інтерфейсів

З точки зору синтаксису інтерфейси дуже схожі на класи. Головна відмінність полягає в тому, що жоден метод в інтерфейсі не має тіла та в ньому не можна об'являти змінні, а тільки константи.

В загальному випадку об'ява інтерфейсу має вигляд:

[public] interface ИмяІнтерфейсу [extends СписокІнтерфейсів]

Приклад інтерфейсу:

```
public interface Product
{
    static final String MAKER = "Cisco Corp.";
    static final String Phone = "555-123-4567";
    public int getPrice(int id );
 }
```

Ще один реальний приклад інтерфейсу з пакета java.applet:

```
public interface AudioClip
{        /**   Starts playing the clip.
                 Each time this method is called,
                 the clip is restarted from the beginning.       */
   void play();
       /**    Starts playing the clip in a loop.
        */
   void loop();
       /**    Stops playing the clip.
        */
   void stop();
}
```

Як бачимо, основна задача інтерфейсу – оголосити абстрактні методи, які будуть реалізовані в інших класах.

Зазначимо, що інтерфейси також можна розширювати, створюючи нащадків від вже існуючих інтерфейсів, наприклад:

```
interface Monitored extends java.lang.Runnable, java.lang.Clonable
{
        boolean IsRunning();
}
```

Таким чином, інтерфейси – це новий елемент у програмуванні, якого немає на інших мовах. Він дозволяє описувати у себе всередині методи і змінні. Не вказуючи при цьому конкретну реалізацію цих методів.

Реалізація інтерфейсів

В класі, що реалізує інтерфейс, мають бути перевизначені всі методи, які були оголошені в інтерфейсі, інакше клас буде абстрактним. Звичайно, в класі можуть бути присутні інші, власні методи, не описані в інтерфейсах. Приклад класу, що реалізує інтерфейс:

```
class Shoe implements Product
{       public int getPrice(int id)
        {
                if (id==1)    return 5;
                else          return 10;
         }
         public String getMaker()
         {
                return MAKER;
         }
}
```

Зверніть увагу на ключове слово *implements* (реалізація), яке з'явилось в заголовку класу на відміну від звичного *extends*.

Контрольні питання

1. *Що являє собою пакет з точки зору операційної системи?*
2. *Які елементи можуть входити до складу інтерфейсів?*
3. *В чому відмінність інтерфейсів від класів?*
4. *Чому розробники Java відмовилися від множинної спадковості на користь інтерфейсів?*

2 ПАКЕТ JAVA.AWT: ІНТЕРФЕЙС КОРИСТУВАЧА ТА ОБРОБКА ПОВІДОМЛЕНЬ

2.1 Реалізація інтерфейсу користувача

Розглянемо найбільший і, напевно, найкорисніший розділ мови Java, пов'язаний з реалізацією інтерфейсу користувача. Для цього необхідно вивчити базові класи пакета ***java.awt*** (Abstract Window Toolkit), показані на рисунку 3.

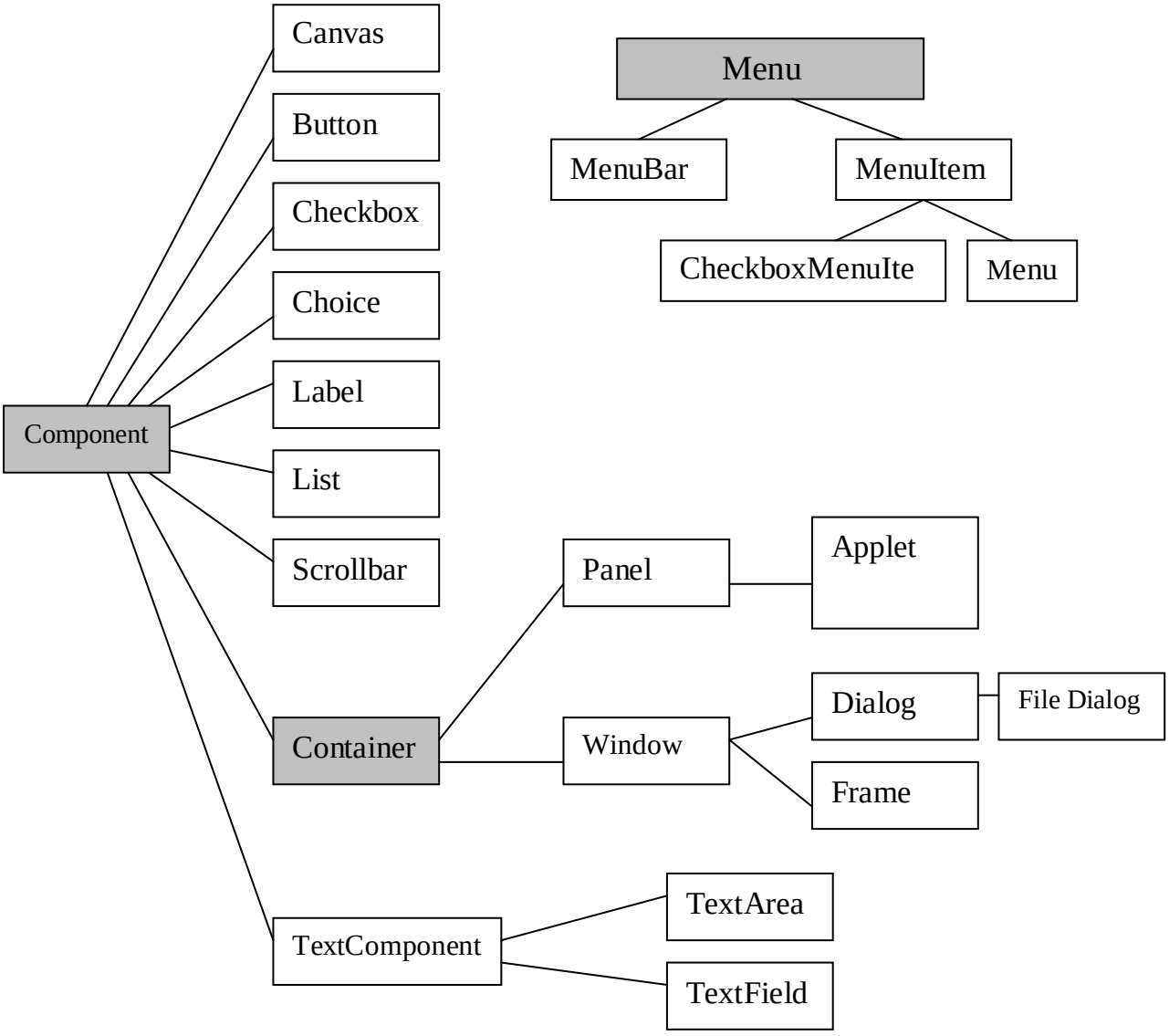

Рисунок 3 – Ієрархія класів пакета *java.awt*

Отже, що ж таке *awt*? Це набір класів Java, кожний з яких відповідає за реалізацію функцій і відображення того чи іншого елемента графічного інтерфейсу користувача (GUI). Практично всі класи візуальних компонентів є нащадками абстрактного класу *Component*. Лише візуальні елементи

меню успадковуються від іншого класу – *MenuComponent*. Керуючі елементи подані такими класами: *Button* (кнопка), *Checkbox* (кнопка з незалежною фіксацією), *Choice* (список Windows), *Label* (рядок), *List* (список вибору Windows) і *Scrollbar* (смуга прокручування). Це досить прості класи, успадковані від абстрактного класу *Component* безпосередньо.

Однак у складі *java.awt* є класи інтерфейсних елементів, що мають проміжного пращура. Гарним прикладом тому є клас *Panel* для створення різних панелей. У нього є проміжний абстрактний клас-пращур *Container*, що слугує родоначальником багатьох класів-контейнерів, здатних містити в собі інші елементи інтерфейсу. Від цього ж класу успадковується клас вікна *Window*, що зображає на екрані найпростіше вікно без меню і рамки. У цього класу є два часто використовуваних нащадки: *Dialog*, назва якого говорить сама за себе, і *Frame* – стандартне вікно Windows. Ще один проміжний клас *TextComponent* породжує два корисних класи – *TextField* (аналог рядка введення Windows) і багаторядкове вікно текстового введення *TextArea*. Відокремлено від інших елементів стоїть клас *Canvas*. Його візуальне зображення – порожній квадрат, на якому можна виконувати рисування і який може обробляти події натиснення кнопок миші.

Від свого батьківського класу *Component* всі візуальні елементи переймають загальну для них усіх поведінку, пов'язану з їх візуальною та функціональною сторонами. Наведемо список основних функцій, що їх виконують всі компоненти, та методів для їх реалізації (таблиця 3):

Таблиця 3 – Основні методи класу *Component*

Назва методу	Функціональне призначення
1	2
getFont() setFont() getFontMetrics()	визначає або встановлює шрифт компонента
setForeground() getForeground()	установлення і зчитування кольору компонента
setBackground() getBackground()	установлення і зчитування кольору тіла компонента
preferredSize() minimumSize()	повертають менеджеру розкладок інформацію про кращий і мінімальний розміри компонента, відповідно
resize() size()	встановлює і визначає розміри компонента
show() hide()	показує та приховує компонент
isVisible() isShowing()	повертає true, якщо компонент відображений, і значення false, якщо компонент прихований
disable() enable()	забороняє або дозволяє компонент

Продовження таблиці 3

1	2
isEnable()	повертає true, якщо компонент дозволений
paint() update() repaint()	відображення компонента
handleEvent() action()	обробка повідомлень
keyDown() keyUp()	обробка повідомлень клавіатури
mouseDown() mouseUp() mouseDrag() mouseMove() mouseEnter() mouseExit()	обробка повідомлень миші

Використання компонентів інтерфейсу користувача

Label (Текст)

За допомогою класу *Label* можна створювати текстові рядки у вікні Java-програм і аплетів. Для створення об'єкта цього типу існують три види конструкторів:

```
Label();                    // створюється пустий рядок
Label(String str);          // надпис, вирівняний вліво
Label(String str, int align);   // надпис із заданим вирівнюванням
```

де змінна *align* може приймати три значення: Label.LEFT, Label.CENTER, Label.RIGHT.

Найбільш корисні методи класу *Label* наведено в таблиці 4.

Таблиця 4 – Основні методи класу *Label*

SetText(String str)	змінює текст в рядку
setAlignment(int align)	змінює вирівнювання
String getText()	повертає текст рядка
int getAlignment()	повертає значення вирівнювання

Приклад.

```
Label MyLabel1 = new Label("Надпис по центру", Label.CENTER);
Label MyLabel2 = new Label("Вирівняний вліво");
```

Button (Кнопка)

Клас *Button* зображає на екрані кнопку. У цього класу є два типи конструктора. Перший з них створює кнопку без надпису, другий – з надписом:

```
Button();              // кнопка без тексту на ній
Button(String str); // кнопка  із  текстом на ній
```

Корисні методи:

setLabel() – створити або замінити надпис на кнопці;
getLabel() – дізнатись про надпис на кнопці.

Обробка кнопок. При натисненні на кнопку викликається метод *action()*:

```
public boolean action(Event evt, Object wA);
```

де *evt* – подія, яка відбулася (*evt.target* – до якого об'єкта відноситься подія, *evt.when* – час виникнення події, ...);
wA – текст надпису на кнопці.

Приклад.

```
...
public class Example1 extends Applet
{
  public void init()
  {
        add(new Button("Red");
        add(new Button("Green");
  }

  public Boolean action(Event e, Object wA)
  {
    if (!(e.target instanceof Button))
            // перевіряємо, чи натиснуто якусь кнопку
            return false;
        if ((String) wA = "Red")  // яку саме кнопку?
            setBackground(Color.red);
        else
            setBackground(Color.green);
        return true;
    }
}
```

Checkbox (Прапорець та Перемикач)

Клас *Checkbox* відповідає за створення і відображення кнопок з незалежною фіксацією. Це кнопки, що мають два стани: "увімкнена" і "вимкнена". Клацання на такій кнопці приводить до того, що її стан змінюється на протилежний. В початковий момент прапорець скинутий (тобто має стан *false*).

Якщо розмістити кілька кнопок з незалежною фіксацією всередині елемента класу *CheckboxGroup*, то замість них ми одержимо кнопки із залежною фіксацією – перемикачі (radio button*)*, тобто групу кнопок, серед яких у один і той самий момент може бути включена тільки одна. Якщо натиснути будь-яку кнопку з групи, то раніше натиснута кнопка буде відпущена (це нагадує радіоприймач, в якому одночасно може працювати тільки одна програма).

Конструктори:

```
Checkbox();                 // прапорець без тексту
Checkbox(String str);       // прапорець з текстом
Checkbox(String str, CheckboxGroup group, boolean initState);
    /* прапорець з текстом,  що належить групі (якщо
    створюється саме прапорець, а не перемикач, то цей параметр
    повинен мати значення null)  */
```

Корисні методи цього класу наведені в таблиці 5.

Таблиця 5 – Основні методи класу *Checkbox*

boolean getState()	метод класу Checkbox, що повертає статус прапорця
getCurrent()	метод класу CheckboxGroup, повертає перемикач, який в даний момента включений
SetCurrent()	встановлює активну кнопку перемикача (для класу CheckboxGroup)
setLabel() getLabel()	встановлює або повертає текст при прапорці або перемикачі

Обробка перемикачів та прапорців. При натисканні мишею на прапорці або перемикачі викликається метод *action()*, другий параметр якого *wA* є об’єктом класу *boolean*, який має значення *true*, якщо прапорець встановлено, і *false* – в протилежному випадку.

Приклад.

```
import java.awt.*;
import java.applet.*;
public class ExCheckbox extends Applet
{   Checkbox rbBlue, rbWhite;
    CheckboxGroup gr;
```

```
public void init()
    {       add (new Checkbox("Red"));
            add (new Checkbox("Green"));
            gr = new CheckboxGroup();
            Checkbox rbBlue = new Checkbox("Blue",gr,true);
            Checkbox rbWhite = new Checkbox("White",gr,false);
            add(rbBlue);
            add(rbWhite);
    }
      public boolean action(Event e, Object wA) {
            if (e.target instanceof Checkbox)
            {       Checkbox cur = (Checkbox) e.target;
                    if (cur.getLabel() == "Red" && cur.getState())
                            setBackground(Color.red);
                    if (cur.getLabel() == "Green" && cur.getState())
                            setBackground(Color.green);
                     if (cur.getLabel() == "Blue" && cur.getState())
                            setBackground(Color.blue);
                    if (cur.getLabel() == "White" && cur.getState())
                            setBackground(Color.white);
                    return true;
            }
      return false;
  } }
```

Choice (Список, що розкривається)

Коли потрібно створити список, що розкривається, можна вдатися за допомогою до класу *Choice*. Створити його можна за допомогою конструктора

Choice();

Корисні методи класу *Choice* наведено в таблиці 6:

Таблиця 6 – Основні методи класу *Choice*

addItem(String str)	додати елемент в список
select(int n)	вибрати рядок з визначеним номером у списку
select(String str)	вибрати визначений рядок тексту зі списку
int countItems()	повернути кількість пунктів у списку
int getSelectIndex()	повернути номер вибраного рядка (нумерація починається з 0)
String getItem(int n)	повернути рядок з визначеним номером у списку
String getItem()	повернути вибраний рядок

Обробка списків, що розкриваються. При виборі елемента списку викликається метод *action()*, другий параметр *wA* якого містить в собі назву вибраного елемента.

Приклад.

```
import java.awt.*;
import java.applet.*;
public class ExChoice extends Applet
{
        Choice ch;
        public void init()
        {       ch=new Choice();
                ch.addItem("Red");
                ch.addItem("Green");
                ch.addItem("Blue");
                ch.addItem("White");
                add(ch);
        }
        public boolean action(Event e, Object wA)
        {
                if (e.target instanceof Choice)
                {
                        Choice cur = (Choice) e.target;
                        switch (cur.getSelectedIndex())
                        {       case 0: setBackground(Color.red); break;
                                case 1: setBackground(Color.green); break;
                                case 2: setBackground(Color.blue); break;
                                case 3: setBackground(Color.white); break;
                        }
                        return true;
                }
                return false;
        }
}
```

List (Список)

Клас *List* (список) за призначенням дуже схожий на клас *Choice*, але надає користувачу не такий список, що розкривається, а вікно зі смугами прокручування, всередині якого знаходяться пункти вибору. Будь-який з цих пунктів можна вибрати подвійним клацанням миші або, вказавши на нього мишею, натиснути клавішу <Enter>. Причому можна зробити так, що буде можливим вибір декількох пунктів одночасно.

Корисні методи класу *List* наведено в таблиці 7.

Таблиця 7 – Основні методи класу *List*

addItem(String str)	додати рядок до списку;
addItem(String str, int index)	вставити рядок в список в позицію index (якщо index = -1, то додається в кінець списку);
replaceItem(String str, int index)	замінити елемент вибору в зазначеній позиції
delItem(int index)	видалити зі списку вказаний пункт
delItems(int start, int end)	видалити елементи вибору з номерами, що входять в інтервал від номера start до номера end;
getItem(int n)	текст пункту вибору
clear()	очистити список (знищити всі елементи списку відразу)
select(int n)	виділити пункт із вказаним номером
deselect(int n)	зняти виділення з вказаного пункту
isSelected(int n)	повернути значення true, якщо пункт із зазначеним номером виділений, інакше повернути false
countItems()	порахувати кількість пунктів вибору в списку
getRows()	повернути кількість видимих у списку рядків вибору
getSelectedIndex()	довідатися порядковий номер виділеного пункту; якщо повертається -1, то вибрано декілька пунктів (метод для списків I типу)
getSelectedItem()	прочитати текст виділеного пункту вибору (для списків I типу);
int[] getSelectedIndexes()	повернути масив індексів виділених пунктів (для списків II типу);
String[] getSelectedItems()	повернути масив рядків тексту виділених пунктів (для списків II типу)
allowsMultipleSelections()	повернути true, якщо список дозволяє множинний вибір
setMultipleSelections()	включити або виключити режим дозволу множинного вибору
makeVisible(int n)	зробити елемент із вказаним номером видимим у вікні списку
getVisibleIndex()	повернути індекс елемента вибору, що останнім після виклику методу makeVisible() став видимим у вікні списку

Створення об'єкта класу *List* може відбуватися двома способами. Ви можете створити порожній список і додавати в нього пункти, викликаючи метод *addItem()*. При цьому розмір списку буде рости при доданні пунктів.

Інший спосіб дозволяє відразу обмежити кількість видимих у вікні списку пунктів. Інші пункти вибору можна побачити, прокрутивши список.

Конструктори:

```
List ();          // список для вибору тільки одного параметра
List (int row, boolean mult);       // створення списку,  в якому
                  // row – кількість елементів, які можна одночасно
                  // бачити, mult – дозвіл вибору більше одного ел-та
```

Обробка списків. Клас *List* не використовує метод *action()*. Замість нього використовується метод

handleEvent(Event evt);

де *evt.id* – статична константа, що приймає одне з двох значень:
LIST_SELECT – елемент вибраний;
LIST_DESELECT – не вибраний;
evt.arg – змінна, яка містить індекс вибраного елемента.

Приклад.

```
import java.awt.*;
import java.applet.*;

public class ExList extends Applet
{       List ch;
        public void init()
        {       ch=new List();
                ch.addItem("Red");        ch.addItem("Green");
                ch.addItem("Blue");       ch.addItem("White");
                ch.addItem("Cyan");       add(ch);
        }
        public boolean handleEvent(Event e)
        {       if (e.target == ch)
                {   if (e.id==Event.LIST_SELECT)
                    {      Integer sel=(Integer) e.arg;
                           switch (sel.intValue())
                           {       case 0: setBackground(Color.red); break;
                                   case 1: setBackground(Color.green); break;
                                   case 2: setBackground(Color.blue); break;
                                   case 3: setBackground(Color.white); break;
                                   case 4: setBackground(Color.cyan); break;
                           }
                     }   return true;
              } return false;
      }
}
```

Scrollbar (Смуга прокручування)

Клас *Scrollbar* зображає на екрані знайому усім смугу прокручування. За допомогою цього елемента можна прокручувати зображення і текст у вікні або встановлювати деякі значення. Щоб створити смугу прокручування, необхідно викликати конструктор об'єкта класу *Scrollbar*. Це можна зробити трьома способами :

```
Scrollbar()   // смуга прокручування з параметрами за замовчуванням
Scrollbar(Scrollbar.VERTICAL);  //смуга прокручування
                                // з вертикальною орієнтацією
Scrollbar(<орієнт.>, <поч. зн.>, <видно>, <мін. зн.>, <макс. зн.>);
```

де <орієнт.> − орієнтація смуги, що задається константами:
Scrollbar.HORIZONTAL і
Scrollbar.VERTICAL;

<поч.зн.> − початкове значення, в яке ставиться движок смуги прокручування;

<видно> − скільки пікселів прокручуваної області видно, і наскільки цю область буде прокручено при натисканні мишею на смузі прокручування;

<мін. зн.> − мінімальна координата смуги прокручування;

<макс. зн.> − максимальна координата смуги прокручування.

Зазвичай як прокручувана область виступає об'єкт класу *Canvas* чи породжений від нього об'єкт. При створенні такого класу його конструктору необхідно передати посилання на смуги прокручування.

TextField і TextArea

Два родинних класи, *TextField* і *TextArea,* які успадковують властивості класу *TextComponent*, дозволяють відображати текст із можливістю його виділення і редагування. За своєю суттю це маленькі редактори: однорядковий (*TextField*) і багаторядковий (*TextArea*). Створити об'єкти цих класів дуже просто: потрібно лише передати розмір у символах для класу *TextField* і розмір у кількості рядків і символів для класу *TextArea*:

Конструктори:

```
TextField();          // пустий рядок невизначеної довжини
TextField(int);       // пустий рядок вказаної довжини
TextField(String); // рядок з попередньо визначеним текстом
TextArea();           // пусте поле введення невизначених розмірів
TextArea(int, int); // пусте поле визначених розмірів
TextArea(String);  // поле з попередньо визначеним текстом
TextArea(String, int, int); // поле з текстом заданих/ розмірів
```

Корисні методи класів *TextField* і *TextArea* наведено в таблиці 8.

Таблиця 8 – Основні методи класів *TextField* і *TextArea*

Спільні методи для обох класів	
getText()	прочитати текст
setText(String)	відобразити текст;
Select(int, int)	виділити текст між початковою і кінцевою позиціями;
selectAll()	виділити весь текст
GetSelectedText()	прочитати виділений текст;
SetEditable(Boolean)	заборонити редагування тексту
isEditable()	перевірити, чи дозволене редагування тексту
getSelectionStart()	повернути позицію початку виділення
GetSelectionEnd()	повернути позицію закінчення виділення
getColumns()	повернути кількість видимих символів у рядку редагування (не довжина рядка!)
Додаткові методи класу *TextField*	
setEchoChar(char)	встановити символ маски; застосовується при введенні паролів
getEchoChar()	дізнатися про символ маски
echoCharIsSet()	дізнатитися, чи встановлений символ маски
Додаткові методи класу *TextArea*	
int getRows()	дізнатися про кількість рядків у вікні
insertText(String, int)	вставити текст у вказаній позиції
appendText(String)	додати текст в кінці
replaceText(String, int, int)	замінити текст між заданими початковою і кінцевою позиціями

Обробка повідомлень текстового рядка та текстового поля. Як і клас *List*, клас *TextArea* не використовує метод *action()*. Оскільки події цього класу – це події клавіатури і миші, тому краще створити додаткову кнопку, яку користувач міг би натиснути, вказуючи, що введення завершено. Після цього можна застосувати метод *getText()* для отримання результату введення і редагування.

Для класу *TextField* також краще застосувати додаткову кнопку. Хоча може бути використаний і метод *action()*, але тільки тоді, коли користувач натискає клавішу <Enter>.

Розкладки

Для того щоб керувати розташуванням елементів всередині віконконтейнерів, у Java існує менеджер розкладок (layout manager). Від нього успадковуються п'ять класів, що визначають той чи інший тип розташування компонентів інтерфейсу користувача у вікні. Коли вам потрібно

змінити тип розташування, ви створюєте той чи інший клас розкладки, що відповідає вашим вимогам, і передаєте його у викликуваний метод *setLayout()*, що змінює поточну розкладку:

```
setLayout(new BorderLayout());
```

FlowLayout (послідовне розташування)

Це найпростіший спосіб розташування елементів один за одним, застосовуваний за замовчуванням. Коли в одному рядку вже не вміщуються нові елементи, заповнення продовжується з нового рядка.

Конструктори:

```
FlowLayout();              // розкладка з вирівнюванням рядків по центру
FlowLayout(int align);  // розкладка із заданим вирівнюванням
FlowLayout(int align, int horp, int verp);
                           // розкладка з вирівнюванням і заданням проміжків
                           // між елементами по горизонталі і вертикалі
```

Параметр *align* може приймати одне з значень:
FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.CENTER.

GridLayout (табличне розташування)

GridLayout розташовує елементи послідовно один за одним усередині деякої умовної таблиці. Всі елементи будуть однакового розміру. Розмір комірок можна програмно змінювати.

Конструктори:

```
GridLayout(int nRows, int nCols);
        // задає розкладку з вказаною  кількістю рядків і стовпців
GridLayout(int nRows, int nCols, int horp, int verp);
      // задає розкладку з вказаною кількістю рядків і стовпців і
      // величиною проміжків між елементами (в пікселах)
```

Якщо задано кількість рядків, то кількість стовпчиків буде розраховано, і навпаки. Якщо треба створити розкладку з заданим числом рядків, то кількість стовпців треба вказати 0. Якщо ж треба задати кількість стовпців, то замість кількості рядків слід задати 0. Таким чином, виклик *GridLayout(3, 4)* еквівалентний виклику *GridLayout(3, 0).*

BorderLayout (полярне розташування)

Дана розкладка розділяє контейнер на 5 областей і розміщає елементи або поруч з вибраним краєм вікна, або в центрі. Для цього після установлення *BorderLayout* додання елементів у вікно-контейнер виконується методом *add()* з додатковим параметром, що задається рядками *“North”, “South”, “East”, “West” і “Center”*. Таким чином, дана розкладка розділяє контейнер на п’ять областей.

Конструктори:

BorderLayout(); // розкладка без проміжків між елементами
BorderLayout(int horp,verp); // розкладка з проміжками між
// елементами

Дана розкладка не дозволяє додавати в одну область більше одного компонента. Якщо додано більше одного компонента, то буде видно лише останній.

Метод *add()* для даної розкладки матиме такий вигляд:

add(int poz, Component comp);

де *poz* позначає той край вікна, до якого необхідно притиснути елемент, що вставляється *("North", "South", "East", "West" і "Center").*

CardLayout (блокнотне розташування)

При розкладці цього типу елементи розміщаються послідовно один за одним, як карти в колоді (в кожний момент часу видно тільки один елемент). Звичайно така розкладка зручна, якщо нам необхідно динамічно змінювати інтерфейс вікна. Крім того, ми можемо створювати елементи, що будуть знаходись один над одним, по черзі. Так створюються вкладки, вміст яких відображається при натисканні кнопки миші на заголовку.

GridBagLayout (коміркове розташування)

Це найскладніша, але в той же час і найпотужніша розкладка. Вона розташовує елементи в умовній таблиці, як це робиться у випадку з *GridLayout*. Але на відміну від останньої, можна варіювати розмір кожного елемента окремо. Правда, прийдеться набрати додатково не один рядок вихідного тексту. Для кожного елемента задають власні "побажання". Ці побажання розташовують в полях об'єкта *GridBagConstraints*, який містить такі змінні:

gridx, gridy – координати комірки, куди буде розміщений наступний компонент. За замовчуванням

gridx=gridy= GridBagConstraint.RELATIVE,

тобто для *gridx* це означає комірку праворуч від останнього доданого елемента, для *gridy* – комірка знизу;

gridwidth, gridheight – кількість комірок, яку займає компонент по горизонталі і вертикалі. За замовчуванням – 1. Якщо

gridwidth = GridBagConstraint.REMAINDER або

gridheight = GridBagConstraint.EMAINDER,

то компонент буде передостаннім в рядку (у стовпці). Якщо компонент повинен бути розташований у рядку або у стовпці, слід задати *GridBagConstraint.RELATIVE*;

fill – повідомляє, що робити, якщо компонент менший, ніж виділена комірка. Він може приймати значення:

GridBagConstraint.NONE (за замовчуванням) – лишає розмір без змін;
GridBagConstraint.HORIZONTAL – розтягує компонент по горизонталі,
GridBagConstraint.VERTICAL – розтягує компонент по вертикалі,
GridBagConstraint.BOTH – розтягує по горизонталі і по вертикалі;

ipadx, ipady – вказують, скільки пікселів додати до розмірів компонент по горизонталі та вертикалі з кожної сторони (за замовчуванням 0);

insets – екземпляр класу *Insets* – вказує, скільки місця лишити між границями компонента і краями комірки (тобто "демаркаційна лінія" навколо компонента), містить окремі значення для верхнього, нижнього, лівого і правого проміжків;

anchor – використовується, коли компонент менший за розміри комірки. Може приймати значення:
GridBagConstraint.CENTER (за замовчуванням),
GridBagConstraint.NORTH, GridBagConstraint.NORTHEAST,
GridBagConstraint.EAST, GridBagConstraint.SOUTHEAST,
GridBagConstraint.SOUTH, GridBagConstraint.SOUTHWEST,
GridBagConstraint.WEST, GridBagConstraint.NORTHWEST;

weidhtx, weighty – задають відносні розміри компонентів.

Контейнери

Будь-який з компонентів, який необхідно показати на екрані, повинен бути доданий у клас-контейнер. Контейнери служать сховищем для візуальних компонентів інтерфейсу й інших контейнерів. У пакеті *awt* визначено такі контейнери:

- вікно (*Window*);
- панель (*Panel*);
- фрейм (*Frame*);
- діалогове вікно (*Dialog*).

Навіть якщо в аплеті явно не створюється контейнер, він все одно буде використовуватися, оскільки клас *Applet* є похідним від класу *Panel*.

Найпростіший приклад контейнера − клас *Frame*, об'єкти якого відображаються на екрані як стандартні вікна з рамкою.

Щоб показати компонент користувацького інтерфейсу у вікні, потрібно створити об'єкт-контейнер, наприклад, вікно класу *Frame*, створити необхідний компонент і додати його в контейнер, а вже потім відобразити його на екрані. Незважаючи на настільки довгий список дій, у вихідному тексті цей процес займає усього кілька рядків. Наприклад,

```
Label text = new Label("Рядок"); // створюється текстовий об'єкт
                                 // з надписом "Рядок"
SomeContainer.add (text);   // Об'єкт додається в деякий контейнер
SomeContainer.Show();       // Відображається контейнер
...
```

Методи контейнерів наведено в таблиці 9.

Таблиця 9 – Основні методи класу *Container*

add()	додати елемент інтерфейсу у вікно контейнера
add(Component, int)	передати порядковий номер, куди буде вставлено елемент, і посилання на об'єкт
add(String,Component)	передати посилання на об'єкт, що вставляється. Рядків, припустимих як перший аргумент, всього п'ять: North, South, East, West і Center
getComponent(int)	повернути посилання на компонент (повертає тип Component) за заданим індексом
getComponents()	повернути масив Component[] всіх елементів даного контейнера
countComponent()	повернути кількість компонентів у контейнері
remove()	видалити конкретний елемент
removeAll().	видалити усі візуальні компоненти

Приклад.

```
add(someControl);        // вставити елемент у вікно контейнера
add(-1, someControl);    // вставити елемент після інших
                         // елементів у  контейнері
add("North", someControl);     // вставити елемент у вікно
            // контейнера, притиснувши його до верхньої границі
```

Панель

Клас *Panel* (панель) – це простий контейнер, у який можуть бути додані інші контейнери чи елементи інтерфейсу. Звичайно він використовується в тих випадках, коли необхідно виконати складне розміщення елементів у вікні Java-програми й аплета. При цьому панель може бути включена в склад інших контейнерів.

Конструктор:

Panel();

Панель може містити в собі декілька інших панелей, тобто їх можна вкладати одна в одну.

Приклад.

```
Panel mainPanel, suPanel1, subPanel2;
sainPanel = new Panel();
subPanel1  = new Panel();
subPanel2  = new Panel();
mainPanel.add(subPanel1);
mainPanel.add(subPanel2);
add(mainPanel);
```

Frame (Фрейми)

Одним з найважливіших класів інтерфейсу користувача можна вважати клас *Frame*. За його допомогою реалізуються вікна для Java-програм і аплетів. На відміну від інших класів, самі по собі екземпляри класу *Frame* створюються рідко. Їх використовують для розробки окремих додатків. Звичайно від них успадковується новий клас, а вже потім створюється екземпляр нового класу:

```
public class NewWindow extends Frame
{
        TextArea output;
        public NewWindow (String title)  {    super(title); }
        ...
        public static void main (String args[])
        {               // Створення екземпляра нового класу
                NewWindow win = new NewWindow("New Window Class");
                        // Показати його на екрані
                win.show();
        }
}
```

Корисні методи класу *Frame* наведено в таблиці 10.

Таблиця 10 – Основні методи класу *Frame*

pack()	змінити розмір компонентів у вікні так, щоб їхній розмір був максимально наближений до бажаного
getTitle()	повернути заголовок вікна
setTitle(String)	встановити заголовок вікна
getIconImage()	повернути піктограму вікна
setIconImage(Image)	встановити піктограму вікна
getMenuBar()	повернути об'єкт меню вікна
setMenuBar(MenuBar)	встановити меню вікна
remove(MenuComponent)	забрати вказаний компонент із меню вікна
isResizable()	повернути true, якщо розмір вікна можна змінювати, інакше – false
setResizable(boolean)	дозволити зміну розмірів вікна
getCursorType()	повернути поточний тип курсора миші для вікна
setCursor(int)	встановити тип курсора миші для вікна: Frame.DEFAULT_CURSOR, Frame.CROSSHAIR_CURSOR, Frame.TEXT_CURSOR, Frame.WAIT_CURSOR, Frame.HAND_CURSOR, ...

Dialog

Для підтримки зв'язку з користувачем застосовується клас *Dialog*, на основі якого можна створювати діалогові панелі. На відміну від простих вікон діалогові панелі залежать від того чи іншого вікна, і тому в їхніх конструкторах присутній параметр-посилання на вікно класу *Frame*, в якому розташована ця діалогова панель, тобто для створення діалогового вікна необхідно мати фрейм. Як і у випадку з класом *Frame*, клас *Dialog* сам по собі практично не застосовується. Звичайно від нього успадковується новий клас, екземпляр якого і створюється:

```
class NewDialog extends Dialog
{       ...
        NewDialog(Frame frame, String title)
        { super(dw, title, false); }
        ...
}
```

Конструктори цього класу:

```
Dialog(Frame fr, boolean isModal);
Dialog(Frame fr, String title, boolean isModal);
```

Оскільки діалогові панелі можуть бути модальними (такими, що блокують роботу з іншими вікнами) і немодальними, у конструкторах класу *Dialog* останній параметр визначає модальність. Якщо він дорівнює *true*, то діалогове вікно створюється модальним, у протилежному випадку воно дозволяє переключитися на інше вікно додатка.

Крім загальних для усіх вікон методів *getTitle(), setTitle(), isResizable()* і *setResizable()* у класі *Dialog* є метод *isModal()*, що повертає *true*, якщо діалогова панель модальна.

Класи елементів меню

Навряд чи якийсь сучасний додаток зможе обійтися без смуги меню у вікні. Тому в мові Java є відразу кілька класів для створення меню, успадкованих від класу *MenuComponent*. Перший з них, *MenuBar* – це основний клас усієї системи меню. Він слугує контейнером для інших класів. Коли ви створюєте вікно, то як посилання на меню потрібно передати посилання на клас *MenuBar*. Таким чином, для створення меню використовується конструктор:

```
MenuBar myMenuBar = MenuBar();
```

Додати меню у фрейм можна так:

```
myFrame.setMenuBar(myMenuBar);
```

Наступний клас *Menu* на смузі меню відображується як пункт вибору, що, якщо по ньому клацнути, розкривається у вигляді сторінки з пунктами вибору (pop-up menu). Самі ж елементи вибору меню звичайно реалізуються як екземпляри класів *MenuItem* (простий елемент вибору) і *CheckboxMenuItem* (відмічуваний елемент вибору).

Приклад створення повнофункціональної смуги меню:

```
public class NewWindow extends Frame  {
   public NewWindow()
   {  MenuBar menuBar=new MenuBar(); // створюємо смугу меню
      Menu menu1=new Menu("Menu 1"); // створюємо перше меню
      menuBar.add(menu1);    // створюємо і  додаємо 1-й пункт
      MenuItem item1_1 = new MenuItem("Item #1");
      menu1.add(item1_1);
      CheckboxMenuItem item1_2 = CheckboxMenuItem("Item #2");
      menu1.add(item1_2);   // відмічаємо пункт меню
     Menu menu2 = new Menu("Menu 2");      // створюємо
     menuBar.add(menu2);                          //  і додаємо  друге меню
                        // створюємо і додаємо меню наступного рівня
    Menu nextLevel = New Menu("NextLevelMenu");
    menu2.add(nextLevel);
   }   ...
}
```

Як бачимо, створення меню хоча і тривалий, але зовсім не складний процес. В друге меню додається не пункт вибору класу *MenuItem*, а меню класу *Menu*. Це приводить до того, що при натисканні на пункт 2 смуги меню поруч з'являється наступне меню, вибравши з якого *NextLevelMenu*, можна одержати чергове меню. У такий спосіб у мові Java реалізовано каскадне меню.

Обробка пунктів меню виконується за допомогою методу *action()*, де перший параметр – об’єкт класу *Event,* при цьому треба робити перевірку на належність до меню, а саме:

```
if (e.tsrget instanceof MenuItem) { ... };
```

а другий параметр – назва вибраного пункту меню.

Контрольні питання

1. *Екземпляри яких класів пакета java.awt неможливо утворити?*
2. *Який клас в пакеті java.awt є безпосереднім або непрямим батьком для більшості візуальних компонентів?*
3. *Як створити кнопки з незалежною та залежною фіксацією?*
4. *Як в Java-програмах організовано управління розташуванням елементів інтерфейсу у вікні?*
5. *Що таке контейнер? Дати характеристику класів-контейнерів.*

2.2 Обробка повідомлень

Починаючи з версії Java 1.1 суттєвих змін зазнав спосіб обробки повідомлень (порівнюючи з версією Java 1.0). Хоча старий метод обробки повідомлень ще підтримується, він не рекомендується для використання в нових програмах, тому основну увагу ми будемо приділяти новому методу.

Модель делегування подій

Сучасний підхід до обробки подій оснований на моделі делегування подій (delegation event model), яка визначає стандартні механізми для генерації та обробки подій. Ця концепція доволі проста: джерело генерує подію та посилає її одному або кільком блокам прослуховування (listeners) подій. За цією схемою блок прослуховування просто очікує на подію. Отримавши подію, блок прослуховування обробляє її та повертає управління. Перевага такого способу полягає в тому, що логіку компонента, який обробляє подію, чітко відокремлено від логіки інтерфейсу користувача, що генерує ці події. Елемент інтерфейсу користувача здатний "делегувати" обробку події окремій частині коду.

В моделі делегування подій блоки прослуховування мають зареєструватися у джерелі, щоб приймати повідомлення про події. Це забезпечує важливу перевагу: повідомлення відсилаються тільки тим блокам прослуховування, які хочуть його прийняти. Це більш ефективний спосіб обробки подій, ніж старий метод, що використовується в Java 1.0. Раніше подія розповсюджувалася по обмеженій ієрархії компонентів, поки один з них не обробляв цю подію. Даний метод вимагає від компонентів прийняття подій, які вони не обробляють, на що витрачається певний час. Модель делегування подій усуває такі накладні витрати.

Події

В моделі делегування подія – це об'єкт, який описує зміни стану джерела. Він може бути згенерований як послідовність взаємодій оператора з елементами графічного інтерфейса користувача. Генерацію подій можуть викликати такі дії оператора, як натиснення кнопки, введення символу з клавіатури, вибір елемента в списку, клацання мишею та інші дії. Крім того, програміст має змогу сам визначати події, які буде знаходити та обробляти його додаток.

Джерела повідомлень

Джерело – це об'єкт, який генерує подію. Генерація події відбувається тоді, коли змінюється внутрішній стан цього об'єкта. Джерела можуть генерувати події кількох типів.

Щоб блоки прослуховування мали змогу приймати повідомлення про певний тип подій, джерело має реєструвати ці блоки. Кожен тип подій має власний метод реєстрації. Загальна форма таких методів:

public void addTypeListener(TypeListener el),

де *Type* – це ім'я події, *el* – посилання на блок прослуховування події. Наприклад, метод, який реєструє блок прослуховування події клавіатури, називається *addKeyListener()*. Метод, що реєструє блок прослуховування руху миші, називається *addMouseMotionListener()*. Коли подія відбувається, всі зареєстровані блоки прослуховування повідомляються про це та приймають копію об'єкта події. За будь-яких умов повідомлення відсилаються тільки тим блокам прослуховування, які зареєструвалися для їх приймання.

Деякі джерела можуть дозволяти реєструватися лише одному блоку прослуховування. Загальна форма такого методу:

public void addTypeListener(TypeListener el) throws
java.util.TooManyListenerException

Коли подія відбувається, про це повідомляється тільки цей зареєстрований блок прослуховування.

Джерело також має забезпечити метод, який дозволить блоку прослуховування не реєструвати зацікавленість у певному типі повідомлень. Загальна форма такого методу:

public void removeTypeListener(TypeListener el)

Наприклад, щоб видалити блок прослуховування клавіатури, слід викликати метод *removeKeyListener()*.

Методи, які додають або вилучають блоки прослуховування, забезпечуються джерелом, що генерує подію. Наприклад, клас *Component* забезпечує методи для додання та вилучення блоків прослуховування подій клавіатури та миші.

Блок прослуховування подій

Блок прослуховування – це об'єкт, який отримує повідомлення, коли відбувається подія. До нього висувається дві головних вимоги. По-перше, щоб приймати повідомлення відносно певних типів подій, він має бути зареєстрованим одним або кількома джерелами цих подій. По-друге, він має реалізувати методи для приймання та обробки цих повідомлень.

Методи, що приймають і обробляють події, визначені в наборі інтерфейсів, які знаходяться в пакеті *java.awt.event*. Наприклад, інтерфейс *MouseMotionListener* визначає два методи для приймання повідомлень про події перетягування та пересування миші. Будь-який об'єкт може приймати та обробляти одну або обидві ці події, якщо він забезпечує реалізацію цього інтерфейсу.

Класи подій

В основі механізму обробки подій знаходяться класи подій, які забезпечують зручні для використання засоби інкапсуляції подій. В корені ієрархії класів подій Java знаходиться клас *EventObject*, який розташовано в пакеті *java.util*. Це – суперклас для всіх подій. Один з його конструкторів:

EventObject(Object src),

де *src* – об'єкт, який генерує цю подію.

Клас *EventObject* містить два методи: *getSource(),* який повертає джерело події, і *toString()*, який повертає рядок – еквівалент події.

Клас *AWTEvent,* визначений в пакеті *java.awt,* є підкласом класу *EventObject*. Це суперклас (прямо або опосередковано) всіх AWT-подій, що використовуються моделлю делегування подій. Для визначення типу події можна використовувати його метод *int getID().*

Пакет *java.awt.event* визначає декілька типів подій, які генеруються різноманітними елементами інтерфейсу користувача. В таблиці 11 наведено найважливіші з цих класів подій та описується, коли вони генеруються. Відзначимо, що всі перераховані класи є нащадками класу *AWTEvent.*

Таблиця 11 – Основні класи подій *java.awt.event*

ActionEvent	генерується, коли натиснуто кнопку, зроблено подвійне клацання на елементі списку або вибрано пункт меню
AdjustmentEvent	генерується при маніпуляціях із смугою прокручування
ComponentEvent	генерується, коли компонент сховано, пересунуто, змінено в розмірі або зроблено видимим
ContainerEvent	генерується, коли компонент додається або вилучається з контейнера
FocusEvent	генерується, коли компонент отримує або втрачає фокус
InputEvent	абстрактний суперклас для всіх класів подій введення компонентів
ItemEvent	генерується, коли помічено прапорець або елемент списку, зроблено вибір елемента в списку вибору, вибрано (відмінено) елемент меню з міткою
KeyEvent	генерується, коли отримано введення з клавіатури
MouseEvent	генерується, коли об'єкт перетягується (dragged) або пересувається (moved), зроблено клацання (clicked), натиснуто (pressed) чи відпущено (released) кнопку; генерується, коли покажчик входить або виходить в (поза) межі компонента
TextEvent	генерується, коли змінено значення текстової області або текстового поля
WindowEvent	генерується, коли вікно активізовано/дизактивовано, розгорнуто/згорнуто, відкрито чи відбувся вихід з нього

Інтерфейси прослуховування подій

Модель делегування подій містить дві частини: джерела подій та блоки прослуховування подій. Блоки прослуховування подій створюються шляхом реалізації одного або кількох інтерфейсів прослуховування подій, визначених в пакеті *java.awt.event*. Коли подія відбувається, джерело події викликає відповідний метод, визначений блоком прослуховування, та передає йому об'єкт події як параметр.

В таблиці 12 перераховано найчастіше використовувані інтерфейси прослуховування та наведено короткий опис методів, визначених в цих блоках прослуховування.

Таблиця 12 – Інтерфейси прослуховування подій

ActionListener	визначає один метод для приймання action-події
AdjustmentListener	визначає один метод для приймання adjustment-події
ComponentListener	визначає чотири методи, що обробляють події, пов'язані з приховуванням, пересуванням, зміною та показом компонента
ContainerListener	визначає два методи, що обробляють події додання або вилучення елемента з контейнера
FocusListener	визначає два методи, що обробляють події, пов'язані з отриманням або втратою компонентом фокуса клавіатури
ItemListener	визначає один метод, що обробляє подію зміни стану елемента
KeyListener	визначає три методи, що обробляють події натиснення, відпускання клавіші та введення символу
MouseListener	визначає п'ять методів для обробки подій входу в межі компонента, виходу за межі компонента, клацання, натиснення та відпускання кнопки миші
MouseMotionListener	визначає два методи, що обробляють події перетягування або пересування миші
TextListener	визначає один метод для обробки події зміни текстового значення
WindowListener	визначає сім методів, що обробляють події, пов'язані з активізацією, деактивацією, закриттям, відкриттям, згортанням, розгортанням і виходом з вікна

Так, інтерфейс *ActionListener* визначає метод *actionPerformed()*, який викликається, коли відбувається *action*-подія. Його загальна форма:

void actionPerformed(ActionEvent evt).

Класи-адаптери

В Java існує спеціальний засіб, що носить назву класу адаптера (adapter class), який за певних умов може спростити створення оброблювачів повідомлень. Клас адаптера, або просто адаптер, забезпечує порожню реалізацію всіх методів в інтерфейсі прослуховування подій. Клас корисний, якщо ви хочете приймати та обробляти тільки частину подій, що надаються конкретним інтерфейсом прослуховування. Для цього необхідно визначити новий клас, діючий як блок прослуховування подій, розширюючи один з наявних в пакеті *java.awt.event* адаптерів і реалізуючи тільки ті події, які необхідно обробляти.

Іншими словами, якщо ми реалізуємо деякий інтерфейс, ми зобов'язані описати всі методи, визначені в цьому інтерфейсі. Якщо ми використовуємо замість інтерфейсу адаптер, нам достатньо реалізувати тільки ті методи, які є дійсно необхідними. Решту методів (у вигляді порожніх заготовок) вже містить сам адаптер.

В таблиці 13 подано класи-адаптери, що їх визначено в пакеті *java.awt.event* з відповідними інтерфейсами, які кожен з них реалізує.

Таблиця 13 – Інтерфейси прослуховування подій

Клас-адаптер	Інтерфейс прослуховування подій
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Контрольні питання

1. *Які переваги має модель делегування подій порівняно з традиційним способом обробки подій?*
2. *Як відбувається генерація подій?*
3. *За допомогою якого методу можна визначити джерело події? В якому класі визначено цей метод?*
4. *Перерахуйте основні типи подій. Як програма може дізнатися про тип події?*

2.3 Приклад програми з інтерфейсом користувача

Розглянемо приклад програми, яка дозволить проілюструвати можливості Java щодо створення інтерфейсу користувача та обробки подій.

В першому текстовому полі вводиться одне число, в другому – інше число. При натисненні на кнопку “Check” перевірити, чи не є одне з них квадратом іншого та вивести відповідь на екран.

Зовнішній вигляд вікна програми подано на рисунку 4.

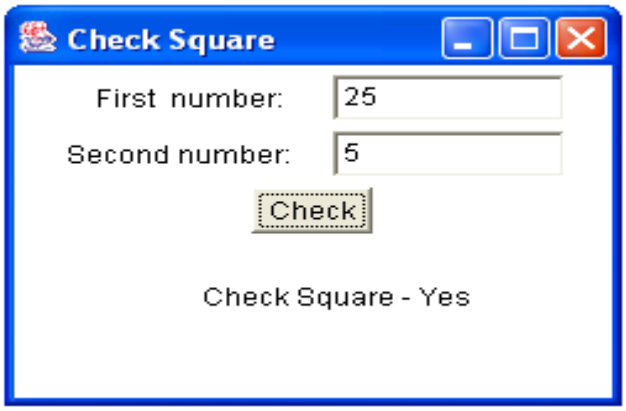

Рисунок 4 – Вигляд вікна програми

Наведемо текст програми для розв’язання даної задачі.

```
import java.awt.*;
import java.awt.event.*;

public class Square extends Frame implements ActionListener
{       TextField  textField1, textField2;
        Button button1;
        String displayStr;

        public Square()
        {       super("Check Square");
                setSize(250, 200);
                setLayout(new FlowLayout());
                Label label = new Label("First  number: "); add(label);
                textField1 = new TextField(10); add(textField1);
                label = new Label("Second number:"); add(label);
                textField2 = new TextField(10); add(textField2);
                button1 = new Button("Check");
                button1.addActionListener(this);
                add(button1);
                addWindowListener(new WindowAdapter()
                        { public void windowClosing(WindowEvent e)
                            {System.exit(0);}
```

```
                });
        displayStr = "";
        show();
    }

    public static void main(String[] args)
    {   new Square();   }

    public void actionPerformed(ActionEvent evt)
    {   String arg = evt.getActionCommand();
        if ("Check".equals(arg))
        {   String str1 = textField1.getText();
            String str2 = textField2.getText();
            int first = Integer.parseInt(str1);
            int second = Integer.parseInt(str2);
            String answer = "No";
            if (first*first==second || second*second==first)
                {answer = "Yes" ; }
            displayStr = "Check Square - " + answer;
            repaint();
        }
    }

    public void paint(Graphics g)
    {  g.drawString(displayStr, 80, 150);  }
}
```

Наведена програма потребує докладних коментарів. На самому початку необхідно імпортувати пакет *java.awt*, оскільки саме в ньому знаходяться класи, що визначають елементи інтерфейсу користувача, і пакет *java.awt. event,* в якому містяться класи обробки подій.

Головний клас *Square* (принагідно нагадаємо, що текст програми має знаходитися в файлі саме з цим ім'ям і розширенням java: Square.java) ми зробили нащадком класу *Frame.* Це дуже важливе рішення, оскільки воно значно мірою визначає всі подальші дії. Отже, які саме переваги дає такий вибір? По-перше, клас *Frame* має візуальний еквівалент у вигляді вікна, тобто вміє подати себе на екрані. По-друге, це вікно вміє реагувати на деякі події миші, змінюючи розміри та положення на екрані, а також згортаючись та розгортаючись. На жаль, воно не вміє закриватися, а це означає, що цьому нам його доведеться навчити. По-третє, клас *Frame*, як нащадок класу *Container*, дозволяє вставляти в себе елементи інтерфейсу користувача (кнопки, поля введення тощо), з якими буде безпосередньо працювати оператор. Наш клас *Square* успадковує всі ці візуальні та функціональні властивості від класу *Frame* як його безпосередній нащадок.

Окрім цього, клас *Square* реалізує інтерфейс *ActionListener*. Нагадаємо, що в ньому міститься метод *actionPerformed()*, на який передається

управління при натисканні кнопки. Отже, всі необхідні дії, пов'язані з перевіркою чисел, будуть знаходитися саме в цьому методі.

Опис класу починається з оголошення змінних. Ці змінні (поля класу) будуть доступними з будь-якого методу класу. Вони відповідають основним елементам інтерфейсу користувача: полям введення чисел і кнопці *Check*. Змінна *displayStr* призначена для виведення відповіді.

Може виникнути питання: чому серед цих елементів ми не бачимо надписів (*Label*). Дійсно, їх можна було б описати тут, зробивши глобальними, але, оскільки вони використовуються лише в одному місці, опишемо їх пізніше, в конструкторі класу.

Конструктор починається з виклику конструктора суперкласу. Для звертання до будь-якого батьківського класу в Java використовується ім'я *super*. Отже, вираз

```
super("Check Square");
```

означає звертання до конструктора класу *Frame*. Як параметр йому передається рядок – заголовок вікна.

Метод *setSize()* встановлює розміри вікна (в попередніх версіях Java для цього використовувався метод *resize()*). Наступний крок – вибір класу розкладки для управління розташуванням елементів інтерфейсу користувача. Ми вибираємо найпростіше послідовне розташування, це означає, що елементи будуть розташовані послідовно зліва направо та згори донизу. Після цього можна перейти безпосередньо до створення елементів і вставляння їх в контейнер за допомогою методу *add()*. При створенні кнопки *button1* зразу за допомогою методу *addActionListener()* реєструємо блок прослуховування події. Це необхідно для того, щоб відповідний метод отримав управління при натисненні кнопки. Інший підхід, який базується на використанні адаптера *WindowAdapter* ми застосуємо для обробки події закриття вікна для завершення роботи програми. Наприкінці конструктора за допомогою методу *show()* ми виводимо на екран наше вікно програми разом з усіма елементами інтерфейсу користувача.

Метод *main()* дуже простий – в ньому створюється клас програми за допомогою конструктора *Square()*. Якщо зараз завершити процес розробки програми, скомпілювати її та виконати, ми побачимо вікно, наведене на рисунку 4. Зрозуміло, що необхідна функціональність поки що відсутня, але зовнішній вигляд програми вже цілком відповідає умовам задачі.

Наступний метод *actionPerformed()* викликається при натисненні на кнопку. Як параметр йому передається об'єкт *evt* класу *ActionEvent*, який несе повну інформацію про подію, що відбулася. Спочатку за допомогою методу *getActionCommand()* ми отримуємо рядок, що містить інформацію про джерело події у вигляді рядка (тобто в цей момент *arg* дорівнює *"Check"*). Далі за допомогою методу *equals()* класу *String*, ми перевіряємо, чи дійсно натиснено саме кнопку *"Check"*, а не іншу. Далі ми отримуємо

інформацію з текстових полів – спочатку за допомогою методу *getText()* у вигляді рядка, а потім за допомогою статичного методу *parseInt()* класу *Integer* – у вигляді цілого числа. Далі формується відповідь – рядок displayStr. Для його виведення на екран слугує метод *paint()*. Відзначимо, що метод *paint()* напряму не викликається. Якщо необхідно обновити інформацію у вікні, програма викликає метод *repaint()*, а вже він передає управління методу *paint()*.

Такий спосіб виведення використовується винятково з метою ознайомлення з різними можливостями щодо виведення інформації. Простіше це було б зробити, виводячи відповідь в третє текстове поле.

Контрольні завдання

1. *Виведіть відповідь в текстове поле textField3. Чи необхідний при цьому виклик методу repaint()? (Підказка. Для виведення інформації в текстове поле скористайтеся методом setText(), що визначений в класі TextField.)*
2. *Перевірте, на які події вміє реагувати програма. Що відбувається при натисненні на клавішу Tab?*
3. *Додайте у програму реакцію на натиснення клавіші <Enter>. При цьому мають виконуватися такі саме дії, як і при натисненні на кнопку "Check".*
4. *Додайте в програму можливість виходу при натисненні клавіші <Esc>.*

3 ПОТОКИ. АПЛЕТИ. РАСТРОВІ ЗОБРАЖЕННЯ

3.1 Створення потоків і керування ними

Створюючи програми для Windows на C++ ви мали змогу розв'язувати такі задачі, як анімація або робота в мережі без застосування багатопоточності. Наприклад, для анімації можна було обробляти повідомлення WM_TIMER.

Якщо повернутися ще назад, то, при створенні програм під DOS, коли програма одночасно щось виводила на екран і аналізувала клавіші, ми використовували цикл *repeat ... until keypressed* або перехоплювали переривання таймера int8.

Звичайно, ані перший, ані другий варіанти в Java не проходять, оскільки тут не передбачено періодичного виклику будь-яких процедур. Тому для розв'язання багатьох задач нам не обминути багатопоточності. Кожен раз, коли нам треба буде робити щось паралельно з основною роботою, ми будемо запускати додатковий потік, що має працювати одночасно з головним потоком. І цей додатковий потік вже із заданим інтервалом часу буде, наприклад, перерисовувати зображення або зчитувати інформацію з буфера при обміні з іншими комп'ютерами.

Створення потоків

Для реалізації багатопоточності ми маємо скористатися класом *java.lang.Thread*. В ньому визначено всі методи для створення потоків, управління їх станом та методи синхронізації потоків.

Є дві можливості для того, щоб дати змогу нашим класам працювати в різних потоках.

По-перше, можна створити свій клас-нащадок від суперкласу *Thread*. При цьому ми отримуємо безпосередній доступ до всіх методів потоків:

public class MyClass extends Thread.

По-друге, наш клас може реалізувати інтерфейс *Runnable*. Це – кращий варіант, якщо ми бажаємо розширити властивості якогось іншого класу, наприклад, *Frame*, як ми це робили раніше, або *Applet*. Оскільки в мові Java немає множинної спадковості, реалізація інтерфейсу *Runnable* – єдина можливість вирішення цієї проблеми.

public class MyClass extends Frame implements Runnable.

І в тому, і в іншому випадку нам доведеться реалізовувати метод *run()*.

Є сім конструкторів в класі *Thread*. Найчастіше застосовується один з них, а саме – з одним параметром-посиланням на об'єкт, для якого буде викликатися метод *run()*. При використанні інтерфейсу *Runnable* метод *run()* визначено у головному класі додатка, тому як параметр конструктора передається значення посилання на цей клас (*this*).

Як бачите, ми знову згадали про метод *run()*. Мабуть, зараз дехто думає: саме середовище Windows періодично викликає метод *run()* – і помиляється. Насправді метод *run()* отримує управління при запуску потоку методом *start()* (безпосередньо не викликається!). Типова програма, що використовує метод *run()* для роботи з потоками, виглядає так:

```
public class MyClass extends Frame  implements Runnable
{
        private Thread myThread = null;   // об'ява потоку
        . . .
        public void start()
        {
                if (myThread == null)
                {
                        myThread = new Thread(this);
                        myThread.start();
                }
        }
        public void run()
        {
        . . .
        }
}
```

Що ж містить метод *run()*? Якщо потік використовується для виконання будь-якої періодичної роботи, цей метод містить цикл вигляду:

```
while (myThread != null).
```

При цьому можна вважати, що код додатка та код методу *run()* працюють одночасно як різні потоки. Коли цикл закінчується та метод *run()* повертає управління, потік завершує роботу нормальним чином.

А що знаходиться всередині циклу *while*? Як правило, він містить виклик методу *repaint()* для перерисовування, а також виклик методу *sleep()* класу *Thread*, який ми зараз розглянемо.

Керування потоками

Після того як потік створений, над ним можна виконувати різні керуючі операції: запуск, зупинку, тимчасову зупинку і т.д. Для цього необхідно використовувати методи, визначені в класі *Thread*.

Запуск потоку. Для запуску потоку на виконання ви повинні викликати метод *start()*:

public void start();

Як тільки додаток викликає цей метод для об'єкта класу *Thread* чи для об'єкта класу, що реалізує інтерфейс *Runnable*, керування отримує метод *run(),* визначений у відповідному класі.

Якщо метод *run()* повертає керування, запущений потік завершує свою роботу. Однак, звичайно метод *run()* запускає нескінченний цикл, тому потік не завершить своє виконання доти, поки він не буде зупинений (чи завершений) примусово. Щоб визначити, запущений даний потік чи ні, можна скористатися таким методом:

public final boolean isAlive();

Зупинка потоку. Якщо у додатку необхідно зупинити потік нормальним неаварійним способом, слід викликати для відповідного об'єкта метод *stop()*:

public final void stop();

Тимчасова зупинка і поновлення роботи потоку. За допомогою методу *sleep()* ви можете затримати виконання потоку на заданий час (в мілісекундах):

public static void sleep(long ms);

При цьому управління передається іншим потокам. Роботу ж нашого потоку буде поновлено через заданий інтервал часу. Метод *suspend()* тимчасово припиняє роботу потоку:

public final void suspend();

Для продовження роботи потоку необхідно викликати метод *resume()*:

public final void resume();

Пріоритети потоків. Клас *Thread* містить методи, які дозволяють управляти пріоритетами потоків. Є також засоби для синхронізації та блокування потоків.

Якщо необхідно запустити **один потік** для анімації, слід виконати такі дії:

1) в об’яві класу вказати, що він реалізує інтерфейс *Runnable*;
2) описати змінну класу *Thread* як поле класу;
3) описати метод *start()* всередині вашого класу. Ця вимога не є обов’язковою, але якщо ви потім захочете перетворити ваш додаток на аплет, ліпше зробити саме так. В методі *start()* створити об’єкт класу *Thread* (за допомогою *new*) і викликати метод *start()* для цього об’єкта:

myThread.start()

Сам метод *start()* (*MyClass.start()*) для додатка можна викликати з методу *main()*. Для аплета він викликається автоматично;

4) описати метод *run()* всередині вашого класу. Всередині методу *run()* створити "нескінченний" цикл. В циклі має знаходитися перерисовування зображення (метод *repaint()*) і призупинка роботи потоку на деякий час (метод *sleep()*), щоб інші потоки також могли виконати свої функції;
5) завершити роботу потоку можна за допомогою методу *stop()*.

Якщо наш додаток повинен запускати **декілька потоків**, варто скористатися іншою технікою. Вона полягає в тому, що ми створюємо один чи декілька класів на базі класу *Thread* або з використанням інтерфейсу *Runnable*. Кожен такий клас відповідає одному потоку і має свій власний метод *run()*. У класі аплета нам потрібно визначити необхідну кількість об'єктів класу, що реалізує потік, при цьому інтерфейс *Runnable* в самому аплеті реалізовувати не потрібно.

Контрольні питання

1. *Які програмні задачі розв'язуються за допомогою багатопоточності?*
2. *Що спільного та відмінного у використанні методів sleep() та suspend()?*
3. *Які дії слід виконати для створення одного додаткового потоку, наприклад, для анімації?*

3.2 Приклад програми для роботи з потоками

Наведемо приклад нескладної програми, що ілюструє основні принципи роботи з потоками. При запуску вона виводить на екран "танцюючі" букви, що утворюють фразу "Java is the best" (рисунок 5).

Рисунок 5 – Вікно програми, що ілюструє роботу з потоками

```
import java.awt.*;
import java.awt.event.*;

public class NervousText extends Frame implements Runnable
{       char separated[];
        String s = null;
```

```
    Thread killme = null;
    int x_coord = 0, y_coord = 0;

    public NervousText()
    {       super("Nervo");
            s = "Java is the best";
            separated =  new char [s.length()];
            s.getChars(0, s.length(), separated, 0);
            setSize(250, 100);
            setFont(new Font("TimesRoman", Font.BOLD, 36));
            addWindowListener(new WindowAdapter() {
                    public void windowClosing(WindowEvent e)
                                                {System.exit(0);}   });
            show();
    }

    public static void main(String[] args)
    {       NervousText nerv=new NervousText();
            nerv.start();
    }

    public void start()
    {       if (killme == null)
            {       killme = new Thread(this);
                    killme.start();
            }
    }

    public void stop()  {       killme = null;       }

    public void run()
    {       while (killme != null)
            {       try {Thread.sleep(100);}
                    catch (InterruptedException e){}
                    repaint();
            }
            killme = null;
    }

    public void paint(Graphics g)
    {       for (int i=0; i<s.length(); i++)
            {       x_coord = (int) (Math.random()*10 + 15*i);
                    y_coord = (int) (Math.random()*10 + 66);
                    g.drawChars(separated,  i, 1, x_coord, y_coord);
            }
    }
}
```

Короткий коментар до цієї програми. Оскільки кожною літерою необхідно при виведенні на екран управляти окремо, нам знадобиться масив символів *separated[]*. Цей масив заповнюється за допомогою методу *getChars()*, визначеному в класі *String*. Перші два параметри цього методу визначають стартову та кінцеву позицію в рядку, інформація з якого копіюється в масив, третій параметр – ім'я масиву, останній параметр – позиція, з якої починається заповнення масиву. Метод *setFont()* дозволяє перевизначити параметри шрифта при виведенні. Без нього текст у вікні виглядав би не так привабливо.

В методі *paint()* для кожної букви визначаються її координати *x_coord* та *y_coord* як сума детермінованої та випадкової складової. Випадкове число ми отримуємо за допомогою статичного методу *random()*, визначеного в класі *Math*. Він повертає випадкове число з діапазону (0, 1), яке потім масштабується до необхідної величини. Виведення символу на екран здійснюється в методі *drawChars()*. Решта методів точно відповідають схемі, наведеній у попередньому розділі і не потребують додаткових пояснень.

Контрольні завдання

1. *Як можна прискорити/уповільнити швидкість "танцю" букв?*
2. *Додайте в програму можливість призупинення виведення при натисненні на кнопку миші та поновлення виведення при повторному натисненні.*

3.3 Виняткові ситуації та їх обробка

Виняток або вилучення (exception) – це спеціальний тип помилки, який виникає у випадку неправильної роботи програми. Виняткова (виключна) ситуація може виникнути при роботі Java-програми в результаті, наприклад, ділення на нуль або може бути ініційована програмно всередині методу деякого класу. Прикладом такого винятку, що генерується програмно, може служити *FileNotFoundException*, який викидається (*throw*) методами класів введення-виведення при спробі відкрити неіснуючий файл. Замість терміну "викидається" часто вживають синоніми: збуджується, генерується, ініціюється.

Після того, як Java-машина створить об'єкт-виняток, цей об'єкт пересилається додатку. Об'єкт, що утворюється при збудженні винятку, несе інформацію про виняткову ситуацію (точка виникнення, опис тощо). Використовуючи методи цього об'єкта можна, наприклад, вивести на екран або в файл інформацію про цей виняток.

Додаток має перехопити виняток. Для перехоплення використовується так званий *try-catch* блок. Наприклад, при спробі читання даних з

потоку стандартного пристрою введення-виведення може виникнути виняток *IOException*:

```
try {
        System.in.read(buffer, 0, 255);
        . . .
}
catch (IOException e)
 {
        String err = e.toString();
        System.out.println(err);
}
```

Якщо читання не вдалося, Java ігнорує всі інші оператори в блоці *try* і переходить на блок *catch*, в якому програма обробляє виняток. Якщо все відбувається нормально, весь код всередині блока *try* виконується, а блок *catch* пропускається.

Зверніть увагу. Блок *catch* нагадує метод, адже йому передається як параметр об'єкт-виняток. Тип параметра – *IOException*, ім'я параметра – *e*. В об'єктному світі Java живуть майже самі об'єкти. І *e* – це також об'єкт класу *IOException*. Можна звернутися до методів цього об'єкта, наприклад, щоб отримати інформацію про виняток. Це і відбувається в нашому прикладі (метод *toString()*).

Що буде, якщо не перехопити виняток? В принципі, нічого особливого. Просто виконання даного потоку команд припиниться та буде виведено системне повідомлення на консоль. Роботу програми при цьому, можливо, буде завершено, а, можливо, і ні (якщо програма має декілька потоків виконання – наприклад, діалогові програми не завершуються, а лише видають повідомлення на консоль. Погано, що ці повідомлення можна навіть не побачити).

Отже, якщо в тому методі, де виникла виняткова ситуація, немає блока його перехоплення, то метод припиняє свою роботу. Якщо в методі, з якого викликано даний метод, також немає блока перехоплення, то і він припиняє свою роботу. І так доти, поки не буде знайдено блок перехоплення або не закінчиться ланцюжок викликаних методів.

Звідси **висновок:** обробляти виняток необов'язково в тому ж самому методі, в якому він генерується.

Розглянемо приклад того, як виникають виняткові ситуації та як їх можна проаналізувати та обробити. Пригадаємо нашу першу діалогову програму, точніше метод *actionPerformed()* з неї:

```
public void actionPerformed(ActionEvent evt)
{
        String arg = evt.getActionCommand();
        if ("Check".equals(arg))
```

```
{       String str1 = textField1.getText();
        String str2 = textField2.getText();
        int first = Integer.parseInt(str1);
        int second = Integer.parseInt(str2);
        String answer = "No";
        if (first*first==second || second*second==first)
                { answer = "Yes" ; }
        displayStr = "Check Square - " + answer;
        repaint();
    }
}
```

Навіть в такій простій програмі можуть виникнути винятки, пов'язані з введенням даних. Отже, користувач натискає кнопку *Check*, а дані не введено. Маємо виняткову ситуацію *NumberFormatException*, про яку сповіщає Java. Давайте змінимо програму, щоб вона перехоплювала виняток:

```
public void actionPerformed(ActionEvent evt)
{
    String arg = evt.getActionCommand();
    if ("Check".equals(arg))
    {
        String str1 = textField1.getText();
        String str2 = textField2.getText();
        try {
            int first = Integer.parseInt(str1);
            int second = Integer.parseInt(str2);
            String answer = "No";
            if (first*first==second || second*second==first)
                answer = "Yes" ;
            displayStr = "Check Square - " + answer;
        }
        catch (NumberFormatException e) {
            displayStr = ‘‘Input format error;
        }
        repaint();
    }
}
```

Тепер все гаразд, заодно ми запобігли і неправильному введенню (букви замість цифр).

Якщо додати ще один рядок

```
first=first/second;
```

(просто так, логіка програми цього не вимагає), ми отримаємо ще один виняток, на цей раз *ArithmeticException*. Його також можна обробити:

```
public void actionPerformed(ActionEvent evt)
{
        String arg = evt.getActionCommand();
        if ("Check".equals(arg))
        {
                String str1 = textField1.getText();
                String str2 = textField2.getText();
                try {
                        int first = Integer.parseInt(str1);
                        int second = Integer.parseInt(str2);
                        String answer = "No";
                        first = first/second;
                        if (first*first==second || second*second==first)
                                answer = "Yes" ;
                        displayStr = "Check Square - " + answer;
                }
                catch (NumberFormatException e) {
                        displayStr = “Input format error”;
                }
                catch (ArithmeticException e) {
                        displayStr = “Division by zero”
                }
                repaint();
        }
}
```

Два блоки *catch* ідуть один за одним, щоб обробити кожен виняток з блока *try.*

При написанні власних методів ви також маєте враховувати, що вони можуть збуджувати винятки. В цьому випадку метод має виглядати так:

```
public int fact(int num) throws IllegalArgumentException
{
    if ( num < 0 || num>10)
    {
        throw new IllegalArgumentException(“Number out of range”);
    }
        int res=1;
        for (int i =1; i<=num; i++)
                res*= i;
        return res;
}
```

Оскільки в методі *fact()* виняток не оброблявся, його має обробити метод, який викликав *fact():*

```
try {
        f = fact(x);
        . . .
}
catch (IllegalArgumentException e) {
        displayStr=e.toString();
}
```

Можна створювати власні класи винятків, але це інша тема.

Підводячи підсумки, можна сказати, що існує два варіанти генерації винятків: автоматична генерація (наприклад, *IOException*, *ArithmeticException*) та явна програмна генерація за допомогою оператора *throw:* наприклад, *throw new IllegalArgumentException.* В будь-якому випадку програма має перехопити виняток та обробити його в блоці *try-catch.*

Контрольні питання

1. *Де в програмі має бути оброблений виняток, якщо він виник?*
2. *Поясніть призначення оператора throw.*
3. *Яка конструкція в мовах програмування за своєю логікою (але не функціональним призначенням) схожа на блок try – catch?*

3.4 Аплети

Аплети – це невеличкі програми, що працюють всередині браузера.

Прикметник “невеличкі” відображує типову практику використання аплетів, а не формальні вимоги. Теоретично аплети можуть бути великими та складними. Але обсяг аплета впливає на час його запуску, оскільки код аплета звичайно передається по мережі Інтернет. Відповідно великий аплет потребує багато часу на завантаження.

Теоретично найбільш популярні браузери підтримують роботу аплетів. Але тут виникають певні проблеми. В середовищі Windows XP за допомогою Інтернет Explorer запустити аплет не вдається. Причина – відсутність віртуальної Java-машини, яка б дозволила виконати аплет. Шляхи подолання цієї проблеми – скачування та подальше використання віртуальної Java-машини – або використання іншого браузера, або використання програми appletviewer.exe, що входить до складу SDK, для відлагодження аплета з наступним запуском його під Windows 98/2000.

Проблема безпеки

Суть проблеми в тому, що аплет – це програма, яку користувач отримує із зовнішнього джерела. Відповідно вона є потенційно небезпечною. Тому аплети сильно обмежені в своїх правах. Наприклад, аплети не можуть читати локальні файли (тобто файли на клієнтській машині), а тим

більше в них писати. Є також обмеження на передачу даних через мережу: аплет може обмінюватись даними тільки з тим сервером Web, з якого його завантажено.

Створення аплетів

Для побудови аплета треба створити клас-нащадок класу *Applet*, який входить до складу пакета *java.applet*, та перевизначити в ньому низку методів класу *Applet*. З деякими з них ми вже познайомились при створенні програм-додатків. Справа в тому, що клас *Applet*, як і клас *Frame*, є непрямим нащадком класу *Component*, відповідно вони мають багато спільних методів. Зокрема, як і при створенні додатків, часто необхідно перекривати метод *paint().*

Але в класі *Applet* є свої специфічні методи, яких не було у класі *Frame* та якими ми не користувались. В класі *Applet* вони визначені як порожні заглушки; ми можемо їх спокійно перевизначати, нічого при цьому не втрачаючи. Тож розглянемо їх.

public void init()

Цей метод викликається браузером лише **один раз** одразу після завантаження аплета перед першим викликом методу *start()*. Цей метод треба перевизначати практично завжди, якщо в аплеті потрібна будь-яка ініціалізація. Мабуть ми не помилимося, якщо весь код, який знаходився у програмі-додатку, в конструкторі перенесемо в метод *init()* аплета.

Як і конструктор, *init()* має свою контрпару:

public void destroy()

Викликається браузером один раз перед вивантаженням даної сторінки. Якщо аплет використовував ресурси, які перед знищенням аплета треба звільнити, це необхідно зробити, перевизначивши цей метод.

public void start()

Викликається браузером при кожному "відвідуванні" даної сторінки. Тобто, можна завантажити дану сторінку, потім завантажити іншу, не закриваючи дану, а потім повернутися до даної. І кожен раз буде викликатися метод *start()* (на відміну від *init()*, який викликається лише один раз). Контрпара *start()* – метод

public void stop()

Викликається браузером при деактивації даної сторінки як у випадку завантаження нової сторінки без вивантаження даної, так і у випадку вивантаження даної. В останньому випадку *stop()* викликається перед *destroy().*

Методи *start()* і *stop()* використовуються в парі для заощадження ресурсів, наприклад, при створенні анімації. Тоді *stop()* може її зупинити, а *start()* запустити знову.

Зверніть увагу. Жоден з цих методів не викликається програмістом напряму, всі вони викликаються браузером.

Перетворення додатка на аплет (на прикладі розглянутої раніше програми NervousText.java) можна здійснити, виконавши такі дії:

- додати рядок

  import java.applet.Applet;

- в заголовку класу замінити *extends Frame* на *extends Applet*;
- конструктор *public NervousText()* замінити на *public void init();*
- знищити (або закоментувати) виклик конструктора суперкласу *super("Nervo");*
- знищити (закоментувати) метод *main()* повністю.
- якщо метод *stop()* не було визначено, то створити його, зупинивши в ньому роботу потоку kill: *kill.stop().*

Наприкінці наведемо приклад простого html-файла, який дозволить запустити на виконання створений аплет.

```
<title> Nervo </title>
<hr>
<applet
        code=NervoText.class
         width=250 height=100>
</applet>
<hr>
```

Даний текст, записаний у будь-якому текстовому редакторі, необхідно занести в файл з довільним ім'ям і розширенням .html, а потім запустити на виконання в якомусь браузері, наприклад, в середовищі Інтернет Explorer. Але попередньо слід створити клас аплета NervousText.class, як і завжди, за допомогою компілятора javac.exe.

Контрольні питання

1. *Чим відрізняється використання методів init() і start()?*
2. *За аналогією з перетворенням додатка на аплет наведіть дії, які необхідно виконати для перетворення аплета на додаток.*
3. *Який метод, обов'язковий для додатків, як правило, відсутній в аплетах?*
4. *Які методи є специфічними для аплетів?*
5. *Які зміни треба внести в заголовок головного класу для зміни додатка на аплет?*

3.5 Створення растрових зображень

При створенні додатка, наприклад, на C++ можна вибирати будь-який формат файлів зображення. При цьому, фактично, нам довелося б з нуля писати весь код завантаження файлів. Натомість Java має готові класи, що здатні завантажувати зображення. За ці зручності доводиться платити тим, що ми можемо завантажувати файли лише двох форматів GIF та JPEG.

Отже, Java підтримує лише два формати зображень: GIF та JPEG.

Завантаження растрового зображення

Виконується за допомогою методу *getImage()*. Існує декілька варіантів цього методу. Насамперед, варіант цього методу, про який, до речі, пишуть всі книжки, визначено в класі *Applet*:

public Image getImage(URL url, String name);

Клас *URL* надає URL (Uniform Resource Locator, уніфікований покажчик ресурсів), який є форматом адрес ресурсів в WWW. Другий параметр задає розташування файла зображення відносно адреси URL. Наприклад,

```
Image img;
img=getImage("http://www.glasnet.ru//~frolov//pic","cd.gif");
```

Якщо аплет бажає завантажити зображення, що розташоване в тому ж каталозі, де і він сам, це можна зробити так:

img = getImage(getCodeBase(), "pic.gif");

Метод *getCodeBase()*, який також належить класу *Applet*, повертає URL-адресу аплета. Замість нього можна використовувати метод *getDocumentBase()*, який повертає URL-адресу HTML-файла, що містить аплет.

img = getImage(getDocumentBase(), "pic.gif");

Якщо створювати не аплет, а додаток, краще використовувати інший варіант *getImage()*, який визначено в класі *Toolkit*

public abstract Image getImage(String filename).

Як звернутися до цього методу (зверніть увагу, що він має один параметр)? Наведемо приклад використання *getImage()* для завантаження файла *duke1.gif*, що знаходиться в підкаталозі *images* поточного каталога:

img = Toolkit.getDefaultToolkit().getImage("image//duke1.gif");

За будь-яких умов метод *getImage()* повертає об’єкт класу *Image*.

Виведення зображення

Зверніть увагу! Насправді метод *getImage()* **не завантажує** зображення через мережу, як це може здаватися. Він тільки створює об'єкт *Image*. Реальне завантаження файла растрового зображення буде виконуватися методом рисування *drawImage()*, який належить класу *Graphics*. Варіанти цього методу (не всі):

```
public abstract boolean drawImage(Image img, int x, int y,
                                  ImageObserver observer);
public abstract boolean drawImage(Image img, int x,int y,
                                  int width, int height, ImageObserver observer);
```

Перший параметр – посилання на об'єкт класу *Image*, який отримано раніше за допомогою *getImage()*. Далі *x* та *y* – координати лівого верхнього кута прямокутного регіону, в якому буде виводитись зображення. Якщо для рисування вибрано метод *drawImage()* з параметрами *width* (ширина) та *height* (висота), зображення буде виведено з масштабуванням. Зверніть увагу! **Помноживши ці параметри на коефіцієнти, можна розтягнути (стиснути) зображення по горизонталі та вертикалі.** Параметр *observer* – це посилання на об'єкт класу *ImageObserver*, який отримає звістку при завантаженні зображення. Звичайно таким об'єктом є сам клас, тому цей параметр вказується як *this*.

Коли викликається метод *drawImage()* зображення ще може бути не завантажено. Оскільки процес завантаження по мережі досить тривалий та непередбачуваний в часі, необхідно передбачити якісь засоби для контролю над цим процесом. Принаймні когось треба повідомити, коли зображення вже буде повністю завантажено, що і робиться в цих методах. Можна виводити зображення у міру їх готовності, а можна дочекатися повного завантаження, а вже потім виводити на екран.

Клас Image

Розглянемо детальніше методи класу Image.

Методи *getHeight()* та *getWidth()*, визначені в класі *Image*, дозволяють визначити відповідно висоту та ширину зображення:

```
public abstract int getHeight(ImageObserver observer);
public abstract int getWidth(ImageObserver observer);
```

Оскільки при виклику цих методів зображення ще може бути не завантажено, як параметр методам передається посилання на об'єкт *ImageObserver*.

Метод *getGraphics()* дозволяє отримати позаекранний контекст зображення для рисування зображення не у вікні додатка або аплета, а в оперативній пам'яті:

public abstract Graphics getGraphics();

Ця техніка використовується для того, щоб спочатку підготувати зображення в пам'яті, а потім за один прийом відобразити його на екрані.

Техніка анімації

Існують три можливості оживити Web-сторінку:

- створення AVI-файла;
- створення багатосекційного GIF-файла;
- використання кількох файлів в форматі GIF або JPEG як окремих кадрів відеофільма.

Звичайно, нас цікавить саме остання можливість. Отже, ідея проста. Завантажуємо за допомогою *getImage()* декілька файлів в масив, елементи якого мають тип *Image.* Потім, користуючись методами управління потоками, створюємо зміну кадрів. При цьому в методі *paint()* необхідно передбачити зміну індексу в масиві, щоб на екран виводився кожен раз новий кадр.

Усунення мерехтіння

Головним чинником цього неприємного явища є те, що зображення рисується безпосередньо перед очима користувача. Це перерисовування помітне оку та викликає ефект мерехтіння. Стандартний вихід з цієї ситуації – подвійна буферизація.

Основна ідея полягає в тому, що поза екраном (в оперативній пам'яті) створюється зображення, і все рисування відбувається саме на цьому зображенні. Коли рисування завершується, можна скопіювати зображення на екран за допомогою лише одного методу, таким чином поновлення екрана відбудеться миттєво.

Інше джерело мерехтіння – метод *update()*, який викликається методом *repaint()*. Стандартний метод *update()* спочатку очищує область рисування, а потім викликає метод *paint()*. Щоб позбавитися від цього, досить просто перевизначити метод *update()*, щоб він викликав метод *paint()*:

```
public void update(Graphics g)
{
        paint(g);
}
```

Але таке просте рішення містить одну небезпеку. Справа в тому, що зображення не обов'язково покриває повністю всю прямокутну область (наприклад, фігурка людини чи щось подібне). При наступному виведенні на екран ми побачимо сліди від попереднього рисунка, якщо нове зображення його повністю не покрило.

Подвійна буферизація, хоча й більш кропітка, дозволяє повністю позбавитись від мерехтіння. Спочатку треба визначити поле нашого класу типу *Image*, яке буде служити позаекранним зображенням:

```
private Image offScreenImage;
```

Далі в конструкторі класу додати ініціалізацію (створення) цього поля:

```
offScreenImage = createImage(size().width, size().height);
```

І, насамкінець, треба перевизначити метод *update()*, щоб він не очищував екран, а дозволяв методу *paint()* сформувати зображення, яке потім копіюється на екран:

```
public synchronized void update(Graphics g)
{
        if (offScreenImage == null)
                offScreenImage = createImage(size().width, size().height);
        Graphics offScreenGraphics = offScreenImage.getGraphics();
        offScreenGraphics.setColor(getBackground());
        offScreenGraphics.fillRect(0, 0, size().width, size().height);
        offScreenGraphics.setColor(g.getColor());
        paint(offScreenGraphics);
        g.drawImage(offScreenImage, 0, 0, width, height, this);
}
```

Контрольні питання

1. *Чому в Java не підтримується робота з файлами в форматі BMP?*
2. *В якому методі має знаходитись виклик методу drawImage()?*
3. *Як можна створити анімаційне зображення?*
4. *В чому полягає механізм подвійної буферизації і як він реалізований у мові Java?*
5. *Де, крім виведення зображень, ще можна використати механізм подвійної буферизації?*

3.6 Можливості Java 2D

У систему пакетів і класів Java 2D, основа якої – клас *Graphics2D* пакета *java.awt*, внесено декілька принципово нових положень.

1. Окрім координатної системи, прийнятої в класі *Graphics* і названої координатним простором користувача *(User Space)*, введено ще систему координат пристрою виведення *(Device Space)*: екрана монітора, принтера. Методи класу *Graphics2D* автоматично переводять систему координат користувача в систему координат пристрою при виведенні графіки.

2. Перетворення координат користувача в координати пристрою можна задати "вручну", причому перетворенням здатне служити будь-яке аффінне перетворення площини, зокрема, поворот на будь-який кут і стиснення. Воно визначається як об'єкт класу *AffineTransform*. Його можна встановити як перетворення за замовчуванням методом *setTransform()*. Можна виконувати перетворення "на льоту" методами *transform()* і *translate()* і робити композицію перетворень методом *concatenate()*. Причому для афінного перетворення координати задаються дійсними, а не цілими числами.

3. Графічні примітиви (прямокутник, овал, дуга і ін.) реалізують тепер новий інтерфейс *Shape* пакета *java.awt*. Для їх викреслювання можна використовувати новий єдиний для всіх фігур метод *draw()*, аргументом якого здатний служити будь-який об'єкт, що реалізував інтерфейс *Shape*. Введений метод *fill()*, що заповнює фігури – об'єкти класу, який реалізує інтерфейс *Shape*.

4. Для викреслювання ліній введене поняття пера (*pen*). Властивості пера описує інтерфейс *Stroke*. Клас *BasicStroke* реалізує цей інтерфейс. Перо має чотири характеристики: воно має товщину; може закінчити лінію будь-яким способом; з'єднувати лінії різними способами та креслити лінію різними пунктирами і штрих-пунктирами.

5. 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 іншої, клас *TexturePaint* – заповненням за заздалегідь заданим зразком (*pattern fill*).

6. Літери тексту розуміються як фігури, тобто об'єкти, що реалізовують інтерфейс *Shape*, і можуть викреслюватися методом *draw()* з використанням всіх можливостей цього методу. При їх викреслюванні застосовується перо, всі методи заповнення і перетворення.

7. Окрім імені, стилю і розміру, шрифт одержав багато додаткових атрибутів, наприклад, перетворення координат, підкреслення або перекреслювання тексту, виведення тексту справа наліво. Колір тексту і його фону є тепер атрибутами самого тексту, а не графічного контексту. Можна задати різну ширину символів шрифту, надрядкові і підрядкові індекси. Атрибути встановлюються константами класу *TextAttribute*.

9. Процес візуалізації (*rendering*) регулюється правилами (*hints*), визначеними константами класу *RenderingHints*.

З такими можливостями Java 2D стала повноцінною системою рисування, виведення тексту і зображень. Розглянемо, як реалізовані ці можливості, і як ними можна скористатися.

Перетворення координат. Клас *AffineTransform*

Правило перетворення координат користувача в координати графічного пристрою *(transform)* задається автоматично при створенні графічного контексту так само, як колір і шрифт. Надалі його можна змінити методом *setTransform()* так само, як змінюється колір або шрифт. Аргументом цього методу служить об'єкт класу *AffineTransform* з пакета *java.awt.geom*.

Перетворення координат задається двома такими конструкторами:

AffineTransform{double a, double b, double c, double d, double e, double f);
AffineTransform(float a, float b, float з, float d, float e, float f).

При цьому точка з координатами *(x,y)* в просторі користувача перейде в точку з координатами *(a·k+c·y+e, b·x+d·y+f)* в просторі графічного пристрою.

Таке перетворення не скривлює площину – прямі лінії переходять в прямі, кути між лініями зберігаються. Прикладами таких перетворень служать повороти навколо будь-якої точки на будь-який кут, паралельні зсуви, віддзеркалення від осей, стиснення і розтягування по осях.

Наступні конструктори використовують як аргумент масиву *{a,b,c,d,e,f}* або *{a,b,c,d}*, якщо *e=f=0*, складений з тих же коефіцієнтів в тому ж порядку:

AffineTransform (double[] arr);
AffineTransform (float[] arr).

П'ятий конструктор створює новий об'єкт за готовим об'єктом:

AffineTransform (AffineTransform at).

Шостий конструктор (спрацьовує за замовчуванням) створює тотожне перетворення:

AffineTransform ().

Всі ці конструктори математично точні, але не завжди зручні при конкретних перетвореннях. Тому у багатьох випадках зручніше створити перетворення статичними методами:

getRotateInstance (double angle) – повертає поворот на кут *angle*, заданий в радіанах, навколо початку координат. Додатний напрям повороту такий, що точки осі OX повертаються у напрямі до осі OY. Якщо осі координат користувача не змінювалися перетворенням віддзеркалення, то додатне значення angle задає поворот за годинниковою стрілкою;

getRotateInstance(double angle, double x, double y) – такий самий поворот навколо точки з координатами (x,y);

getScaleInstance (double sx, double sy) – змінює масштаб по осі OX в *sx* разів, по осі OY – в *sy* разів;

getShareInstance (double shx, double shy) – перетворить кожну точку *(x,y)* у точку *(x+shx·y, shy·x+y)*;

getTranslateInstance (double tx, double ty) – зсовує кожну точку (x,y) у точку *(x+tx, y+ty)*;

createInverse() – повертає перетворення, зворотне діючому.

Після створення перетворень його можна змінити методами:

setTransform(AffineTransform at);
setTransform(double a, double b, double з, double d, double e, double f);
setToIdentity();
setToRotation(double angle);
setToRotation(double angle, double x, double y);
setToScale(double sx, double sy);
setToShare(double shx, double shy);
setToTranslate(double tx, double ty).

Наступні методи виконуються перед поточними перетвореннями, утворюючи композицію перетворень:

concatenate (AffineTfansform 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).

Перетворення, задане методом *preConcatenate(AffineTransform at)*, навпаки, здійснюється після діючого перетворення.

Інші методи класу *AffineTfansform* здійснюють перетворення різних фігур в просторі користувача.

Приклад перетворення системи координат наведено на рисунку 6.

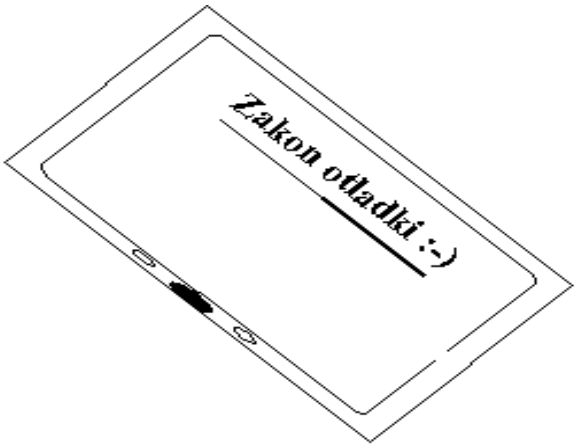

Рисунок 6 – Приклад змінення системи координат засобами Java 2D

Рисування фігур засобами Java 2D. Клас *BasicStroke*

Характеристики пера для рисування фігур описані в інтерфейсі *Stroke*. У Java 2D є поки тільки один клас, що реалізовує цей інтерфейс – клас *BasicStroke*, основний конструктор якого такий:

BasicStroke(float width,int cap, int join, float miter, float[] dash, float dashBegin),

де *width* – товщина пера в пікселах;

cap – оформлення кінця лінії, задане однією з констант:

CAP_ROUND – закруглений кінець лінії;

CAP_SQUARE – квадратний кінець лінії;

CAP_BUTT – оформлення відсутнє;

join – спосіб з'єднання ліній, заданий однією з констант:

JOIN_ROUND – лінії з'єднуються дугою кола;

JOIN_BEVEL – лінії з'єднуються відрізком прямої, перпендикулярним до бісектриси кута між лініями;

JOIN_MITER – лінії просто стикуються;

miter – відстань між лініями, починаючи з якої застосовується з'єднання JOIN_MITER;

dash – довжина штрихів і проміжків між штрихами – масив; елементи масиву з парними індексами задають довжину штриха в пікселах, елементи з непарними індексами – довжину проміжку; масив перебирається циклічно;

dashBegin – індекс, починаючи з якого перебираються елементи масиву dash.

Решта конструкторів задає деякі характеристики за замовчуванням:

BasicStroke (float width, int cap, int join, float miter); // суцільна лінія
BasicStroke (float width, int cap, int join);

Останній конструктор задає суцільну лінію зі з'єднанням, заданим константами JOIN_ROUND або JOIN_BEVEL; для реалізації з'єднання JOIN_MITER задається значення *miter=10.0f*;

BasicStroke (float width) – задає прямокутний кінець CAP_SQUARE і з'єднання JOIN_MITER із значенням *miter=10.0f*,

BasicStroke ( ) – ширина *miter=1.0f*.

Після створення пера одним з конструкторів і установлення 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 звичного набору фігур – прямокутників, прямих, еліпсів, дуг, точок – класами *Rectangle2D, RoundRectangle2D, Line2D, Ellipse2D, Arc2D, Point2D* пакета *java.awt.geom* (рис.7).

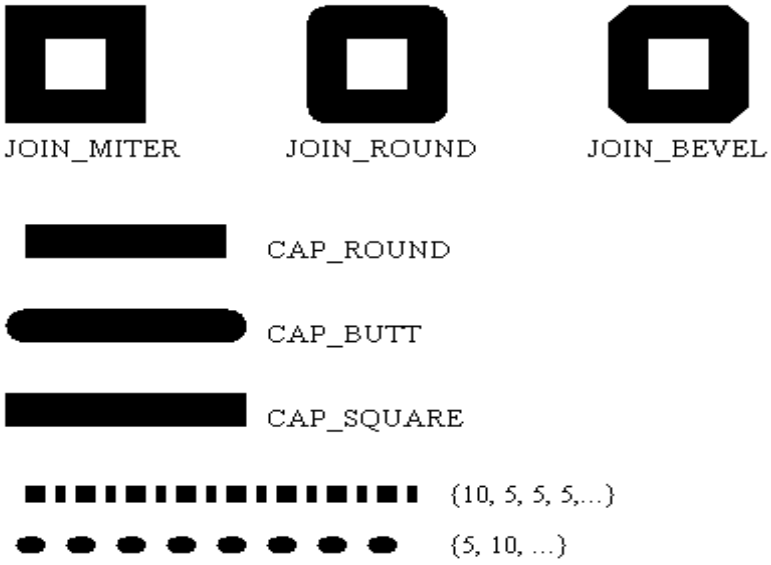

Рисунок 7 – Приклад виведення фігур засобами Java2D

У пакеті *java.awt.geom* є ще один цікавий клас – *GeneralPath*. Об'єкти цього класу можуть містити складні конструкції, складені з відрізків прямих або кривих ліній і інших фігур, з'єднаних або не з'єднаних між собою. Більш того, оскільки цей клас реалізує інтерфейс *Shape*, його екземпляри самі є фігурами і можуть бути елементами інших об'єктів класу *GeneralPath*.

Клас GeneralPath

Спочатку створюється порожній об'єкт класу *GeneralPath* конструктором за замовчуванням

GeneraiPath()

або об'єкт, що містить одну фігуру, конструктором

GeneraiPath (Shape sh).

Потім до цього об'єкта додаються фігури методом

append (Shape sh, boolean connect).

Якщо параметр *connect* дорівнює *true*, то нова фігура з'єднується з попередніми фігурами за допомогою поточного пера.

У об'єкті є поточна точка, спочатку її координати (0,0). Далі можна застосувати методи:

moveTo(float x, float y) – переміститься в точку (x, y);
lineTo(float x, float, y) – відрізок прямої від поточної точки до точки (x,y);
quadTo(float x1, float y1, float x, float, y) – відрізок квадратичної кривої;
curveTo(float x1, float y1, float x2, float y2, float x1, float y1) – крива Безьє.

Поточною після цього стає точка *(x,y)*. Початкову і кінцеву точки можна з'єднати методом *closePath()*. Наприклад, так 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 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*. В наш час Java 2D містить три реалізації цього інтефейсу – класи *Color, GradientPaint* і *TexturePaint.*

Класи GradientPaint і TexturePaint

Клас *GradientPaint* робить заливку таким чином. У двох точках *M* і *N* встановлюються різні кольори. У точці *M(x1, y1)* задається колір *c1*, в точці *N(x2, y2)* – колір *c2*. Колір заливки змінюється від *c1* до *c2* уздовж прямої, що з'єднує точки *M* і *N*, залишаючись постійним уздовж кожної прямої, перпендикулярної до прямої *MN*. Таку заливку створює конструктор

GradientPaint(float xl, float yl, Color cl, float x2, float y2, Color c2).

При цьому поза відрізком *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 cc).

При *cyclic=true* повторюється заливка смуги *MN* у всій фігурі.

Ще два конструктори задають точки як об'єкти класу *Point2D*.

Клас *TexturePaint* поступає складніше. Спочатку створюється буфер – об'єкт класу *BufferedImage* з пакета java.awt.image. Це складний клас, і поки нам знадобиться тільки його графічний контекст, керований екземпляром класу *Grapnics2D*. Цей екземпляр можна одержати методом *createGraphics()* класу *BufferedImage.*

Графічний контекст буфера заповнюється фігурою, яка служитиме зразком заповнення. Потім за зразком буфера 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 встановлюється в графічному контексті методом

setPaint (Paint p)

і використовується надалі методом *fill(Shape sh)*.

Виведення тексту засобами Java 2D

Шрифт – об'єкт класу *Font* – окрім імені, стилю і розміру має ще півтора десятки атрибутів: підкреслення, перекреслювання, нахил, колір шрифту і колір фону, ширину і товщину символів, афінне перетворення, розташування зліва направо або справа наліво.

Атрибути шрифту задають як статичні константи класу *TextAttribute.* Найчастіше використовують атрибути, які перераховані в таблиці 14.

На жаль, не всі шрифти дозволяють задати всі можливі атрибути. Подивитися список допустимих атрибутів для даного шрифту можна методом *getAvailableAttHbutes()* класу *Font*.

У класі Font є конструктор *Font(Map attrib)*, яким можна відразу задати потрібні атрибути створюваному шрифту. Це вимагає попереднього запису атрибутів в спеціально створений для цієї мети об'єкт класу, що реалізовує інтерфейс *Map*: класу *HashMap, WeakHashMap* або *Hashtable.* Наприклад,

```
HashMap hm = new HashMap();
hm.put (TextAttribute.SIZE, newFloat(60.0f));
hm.put (TextAttribute.POSTURE,
                    TextAttribute.POSTURE_OBLIQUE);
Font f = new Font(hm).
```

Можна створити шрифт і другим конструктором, яким ми вже користувалися, а потім додавати і змінювати атрибути методами *deriveFont()* класу *Font.*

Текст в Java 2D має власний контекст – об'єкт класу *FontRenderContext*, що зберігає всю інформацію, необхідну для виведення тексту. Одержати його можна методом *getFontRendexContext()* класу *Graphics2D*.

Вся інформація про текст і про його контекст, збирається в об'єкті класу *TextLayout*. Цей клас в Java 2D замінює клас *FontMetrics*.

У конструкторі класу *TextLayout* задаються текст, шрифт і контекст. Метод *paint()* з усіма визначеними параметрами може виглядати так:

```
public void paint(Graphics gr)
{       Graphics2D  g = (Graphics2D) gr;
        FontRenderGontext  fгc;
        fгc= g.getFontRenderGontext();
        Font  f = new Font( "Serif", Font .BOLD, 15);
        String  s = "Якийсь текст";
        TextLayout tl = new TextLayout(s, f, frc);
        // Продовження методу……
}
```

Таблиця 14 – Атрибути шрифтів

Атрибут	Значення
BACKGROUND	Колір фону. Об'єкт, що реалізовує інтерфейс Paint
FOREGROUND	Колір тексту. Об'єкт, що реалізовує інтерфейс Paint
BIDI_EMBEDDED	Рівень вкладеності проглядання тексту, ціле число у межах від 1 до 15
CHAR_ REPLACEMENT	Фігура, яка замінює символ. Об'єкт GraiphicAttribute
FAMILY	Сімейство шрифту. Рядок типу String
FONT	Шрифт. Об'єкт класу Font
JUSTIFICATION	Допуск при вирівнюванні абзацу. Об'єкт класу Float (від 0,0 до 1,0). Є дві константи: JUSTIFICATION__FULL і JUSTIFICATION_NONE
POSTURE	Нахил шрифту. Об'єкт класу Float. Є дві константи: POSTUREJDBLIQUE і POSTURE_REGULAR
RUN_DIRECTION	Проглядання тексту: RUN_DIRECTION_LTR – зліва направо, RUN_DIRECTION_RTL – справа наліво
SIZE	Розмір шрифту в пунктах. Об'єкт класу Float
STRIKETHROUGH	Перекреслювання шрифту. Константа STRIKETHROUGH_ON (за замовчуванням перекреслювання немає)
SUPERSCRIPT	Підрядкові або надрядкові індекси. Константи: SUPERSCRIPT_NO, SUPERSCRIPT_SUB, SUPERSCRT_SUPER
SWAP_COLORS	Заміна місцями кольору тексту і кольору фону. Константа SWAP_COLORSJDN, за замовчуванням заміни немає
TRANSFORM	Перетворення шрифту. Об'єкт класу AffineTransform
UNDERLINE	Підкреслення шрифту. Константи: UNDERLINE_ON, UNDERLINE_LOW_DASHED, UNDERLINE_LOW_DOTTED, UNDERLINE_LOW_GRAY, UNDERLINE LOW_ONE_PIXEL, UNDERLINE LOW TWO PIXEL
WEIGHT	Товщина шрифту. Константи: WEIGHT_ULTRA_LIGHT, EIGHT_EXTRA_LIGHT, WEIGHT_LIGHT і ін.
WIDTH	Ширина шрифту. Константи: WIDTH_CONDENSED, WIDTH_SEMI_CONDENSED, WIDTH_REGULAR, WIDTH_SEMI_EXTENDED, WIDTH_EXTENDED

У класі *TextLayout* є більше двадцяти методів *getXXX()*, які дозволяють отримати різні відомості про шрифт і контекст тексту, і метод

draw(Graphics2D g, float x, float y),

що викреслює вміст 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 за якимось зразком або вивести тільки контур.

Ще одна можливість створити текст з атрибутами – визначити об'єкт класу *AttributedString* з пакета *java.text*. Конструктор цього класу

AttributedString(String text, Map attributes)

задає відразу і текст, і його атрибути. Потім можна додати або змінити характеристики тексту одним із трьох методів *addAttibute()*.

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 відстежувати курсором поточну позицію в тексті. Це здійснюється методами класу *TextHitinfo*, а методи класу *TextLayout* дозволяють одержати позицію курсора, виділити блок тексту і підсвітити його.

Можна задати окремі правила для виведення кожного символу тексту. Для цього треба одержати екземпляр класу *GlyphVector* методом

createGlyphVector()

класу *Font*, змінити позицію символу методом *setGlyphPosition()*, задати перетворення символу, якщо це допустимо, методом *setGlyphTransform()* і вивести змінений текст методом *drawGlyphVector()* класу *Graphics2D*.

Приклади виведення тексту показано на рисунку 8.

Рисунок 8 – Приклад виведення тексту засобами Java 2D

Методи поліпшення візуалізації

Візуалізацію *(rendering)* створеної графіки можна удосконалити, встановивши один з методів *(hint)* поліпшення з класу *Graphics2D*:

setRenderingHints (RenderingHints.Key key, Object value);
setRenderingHints (Map hints).

Ключі (методи поліпшення) і їх значення задаються константами класу *RenderingHints*, перерахованими в таблиці 15.

Таблиця 15 – Методи візуалізації і їх значення

Методи (ключі)	Значення
KEY_ANTIALIASING	Розмивання крайніх пікселів ліній для гладкості зображення; задаються константами: VALUE_ANTIALIAS_DEFAULT, VALUE_ANTIALIAS_ON, VALUE_ANTIALIAS_OFF
KEY_TEXT_ANTIALIASING	Те ж саме 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 саме для кольору. Константи: VALUE_COLOR_RENDER_SPEED, VALUE_COLOR_RENDER_QUALITY, VALUE_COLOR_RENDER_DEFAULT
KEY_ALPHA_ INTERPOLATION	Плавне з'єднання ліній. Константи: VALUE_ALPHA_INTERPOLATION_SPEED, VALUE_ALPHA_INTERPOLATION_QUALITY VALUE_ALPHA_INTERPOLATION_DEFAULT
KEY_INTERPOLATION	Способи сполучення. Константи: VALUE_NTERPOLATIO_BILINEAR, VALUE_INTERPOLATION_BICUBIC, VALUE_INTERPOLATION_NEARESTNEIGHBOR
KEY_DITHERING	Заміна близьких кольорів. Константи: VALUE_DITHER_ENABLE, VALUE_DITHER_DI SABLE, VALUE_DITHER_DEFAULT

Не всі графічні системи забезпечують виконання цих методів, тому задання вказаних атрибутів не означає, що визначені ними методи застосовуватимуться насправді. От, наприклад, як може виглядати початок методу *paint()* із застосуванням методів поліпшення візуалізації:

```
public void paint(Graphics gr)
{       Graphics2D g = (Graphics2D) gr;
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
        g.setRenderingHint(RenderingHints/KEYJRENDERING,
                        RenderingHints.VALOE_RENDER_QUALITY);
        // Продовження методу
        . . . .
}
```

Контрольні питання

1. *Які нові можливості надають класи і пакети Java 2D?*
2. *Як можна здійснити перетворення системи координат у Java?*
3. *Назвіть основні методи класу AffineTfansform. Яке їх функціональне призначення?*
4. *Які особливості рисування фігур засобами Java 2D?*
5. *Чи використовує Java 2D градієнтні методи заливки? Якщо так, то яким чином і які класи для цього використовуються?*
6. *Які особливості виведення тексту засобами Java 2D?*
7. *Які методи поліпщення візуалізації ви знаєте? В чому їх сутність?*

4 ОБРОБКА ФАЙЛІВ І СТВОРЕННЯ МЕРЕЖНИХ ПРОГРАМ

4.1 Потоки та файли

Насамперед зазначимо, що два англомовних терміни "*thread*" та *"stream"* перекладаються українською як потік. Про потоки "*thread*" ми вже говорили, зараз будемо знайомитися з потоками *"stream"*. Щоб уникнути плутанини, надалі, говорячи про потоки *"thread"*, будемо додавати прикметник "обчислювальний".

Мова Java пропонує нам такі типи потоків:

- потоки, пов'язані з локальними файлами;
- потоки, пов'язані з даними в оперативній пам'яті;
- канальні потоки, тобто потоки для передачі даних між різними обчислювальними потоками;
- стандартні потоки введення-виведення.

Природно, що основну увагу нами буде приділено файловим потокам.

Як і можна було очікувати, всі потокові класи походять безпосередньо від класу *Object* (рис. 9):

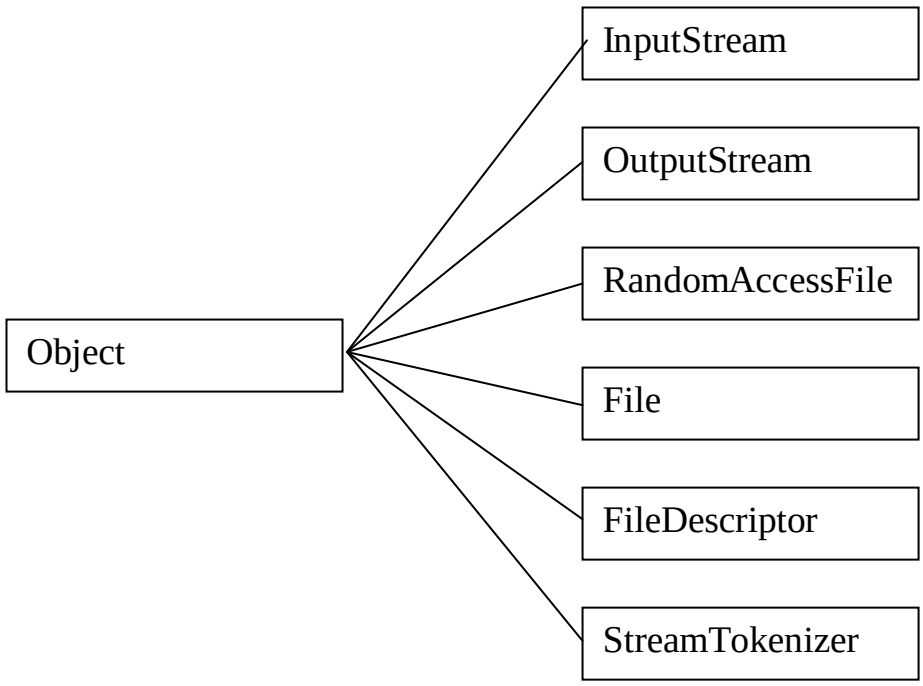

Рисунок 9 – Основні потокові класи

Клас *InputStream* є базовим для більшості класів, на основі яких створюються потоки введення. Саме ці похідні класи і використовуються на практиці. Щодо *InputStream* – це абстрактний клас, який містить декілька корисних методів, а саме:

int read() – зчитує з вхідного потоку окремі байти як цілі числа та повертає значення –1, коли більше нема чого читати;

int read(byte b[]) – зчитує множину байтів в байтовий масив, повертаючи кількість реально введених байтів;

int read(byte b[], int off, int len) – також зчитує дані в байтовий масив, але дозволяє крім того задати ще зсув в масиві та максимальну кількість зчитаних байтів;

long skip(long n) – пропускає n байтів в потоці;

int available() – повертає кількість байтів, які є в потоці в даний момент;

void mark(int readlimit) – помічає поточну позицію в потоці, *readlimit* – кількість байтів, які можна прочитати з потоку до моменту, коли помічена позиція втратить свою силу;

void reset() – повертається до поміченої позиції в потоці;

markSupported() – повертає бульове значення, яке вказує, чи можна в даному потоці відмічати позиції та повертатися до них;

void close() – закриває потік.

Як бачимо, список достатньо серйозний. З урахуванням того, що цей клас має аж шість прямих та чотири непрямих нащадки, детальне вивчення потокового введення потребує багато часу. Але ми нічого не сказали про ще один метод – про конструктор. Хоча саме тут немає нічого цікавого – він без параметрів, та й клас абстрактний.

Клас *OutputStream* утворює пару до класу *InputStream*. Основні методи:

void write(byte b)

void write(byte b[])

void write(byte b[], int off, int len)

void close()

void flush() – виконує примусовий запис всіх буферизованих вихідних даних.

Клас *RandomAccessFile* дозволяє організувати роботу з файлами в режимі прямого доступу, тобто вказувати зсув та розмір блока даних, над яким виконується операція введення-виведення.

Клас *File* призначений для роботи із заголовками каталогів та файлів. За допомогою цього класу можна отримати список файлів та каталогів, розташованих в заданому каталозі, створити або вилучити каталог, перейменувати файл або каталог і т. д.

За допомогою **класу *FileDescriptor*** можна перевірити ідентифікатор відкритого файла.

Клас *StreamTokenizer* дозволяє організувати виділення з вхідного потоку даних елементів, що відділяються один від одного заданими розділювачами.

Класи, похідні від InputStream

Ієрархію класів, похідних від *InputStream*, наведено на рисунку 10.

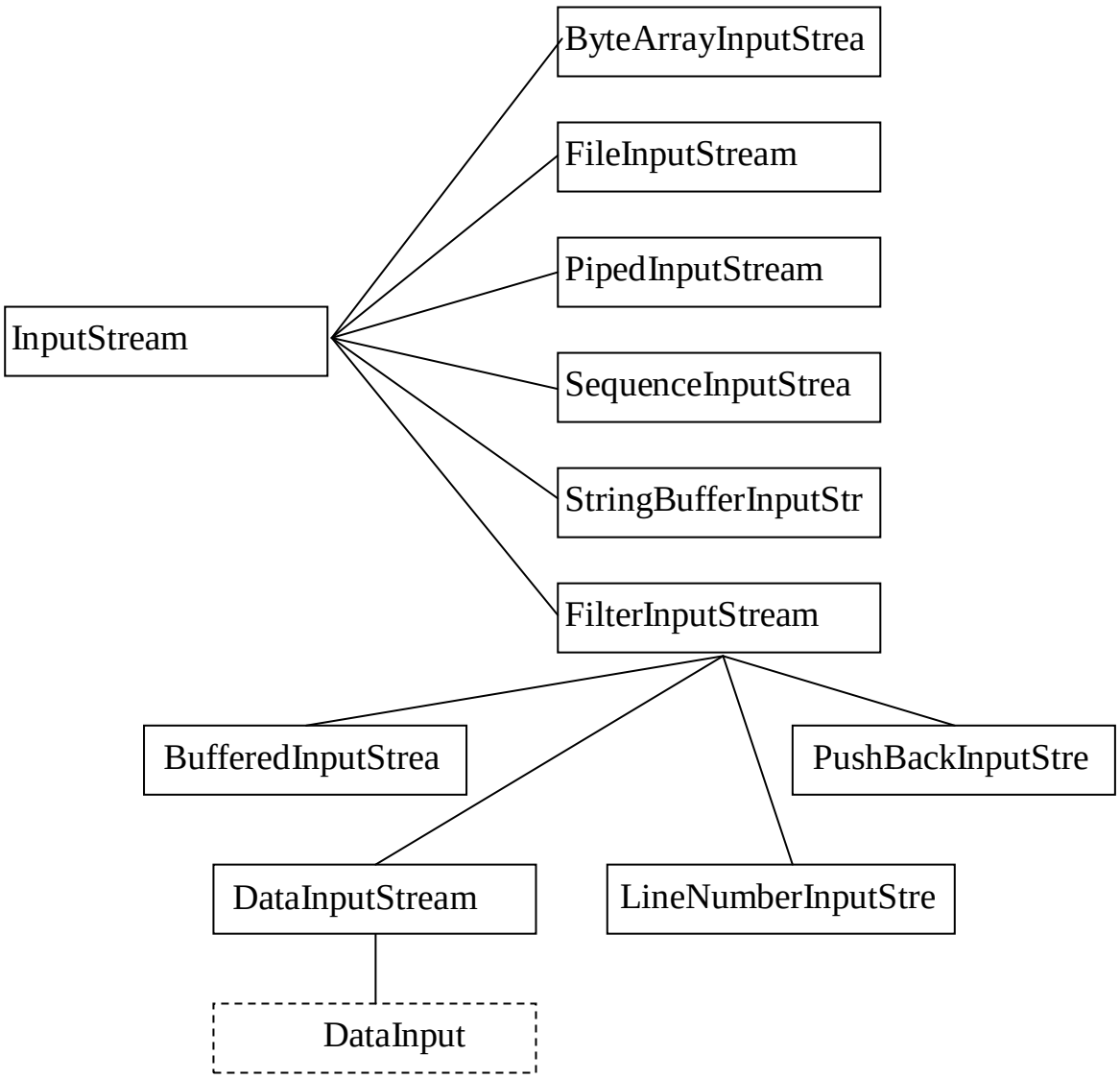

Рисунок 10 – Класи, похідні від *InputStream*

З усіх класів, показаних тут, нас найбільше цікавлять класи *FileInputStream* та *DataInputStream*.

Клас *FileInputStream* майже не має ніяких методів на додаток до тих, що визначені у його батька *InputStream*. Але у нього є три потужних конструктори:

FileInputStream(String name)
FileInputStream(File file)
FileInputStream(FileDescriptor fdObj)

Приклад використання класу *FileInputStream* – програма, що виводить на екран свій вихідний текст.

```
import java.io.*;
class FileApp
{       public static void main(String args[])
        {       byte buffer[] = new byte[2056];
                try
                {   FileInputStream fileIn=new FileInputStream("file1.java");
                    int bytes = fileIn.read(buffer, 0, 2056);
                   String str = new String(buffer, 0, 0, bytes);
                   System.out.println(str);
                }
                catch (Exception e)
                {       String err = e.toString();
                        System.out.println(err);
                }
        }
}
```

Зверніть увагу на цікавий момент. Здавалося б, щоб вивести на екран програму, яка складається з багатьох рядків, необхідне використання циклів. Але в Java, як і в C, рядок обмежується символом ‘\0’ і не збігається з його зображенням на екрані. Тому весь текст програми – це один рядок, який можна вивести на екран за допомогою лише одного методу *println()*.

Клас *FilterInputStream* надає можливість з’єднання потоків. Для чого це потрібно? Як ми бачили, базовий вхідний потік *InputStream* надає лише метод *read()* для читання байтів. Як бути, якщо треба читати рядки або цілі числа? А просто треба цей потік з’єднати з потоком спеціальних даних і таким чином отримати доступ до методів читання рядків, цілих чисел тощо. Але з’єднати – це термін для водопровідників, ми ж маємо вказати які методи для цього треба застосувати. І знову – ніяких нових методів, тільки конструктор. Він має вигляд

public FilterInputStream(InputStream in)

Зверніть увагу на дві речі. Єдиний параметр цього конструктора – посилання на об’єкт класу *InputStream* – насправді може бути і об’єктом

іншого класу, наприклад, тільки що розглянутого нами *FileInputStream*. А він вже знає, як отримати зв'язок з файлом. З іншого боку, сам клас *FilterInputStream* є похідним від класу *InputStream*, отже також може бути параметром в конструкторах інших корисних класів. І останнє. Клас *FilterInputStream* є абстрактним, отже екземплярів цього класу створювати не будемо, а будемо використовувати його класи-нащадки.

Зокрема **клас *DataInputStream*** є похідним від класу з назвою *FilterInputStream.* Він надає можливості для читання всіх вбудованих типів даних Java, а також рядків. Як створити екземпляр цього класу, щоб отримати доступ до його методів? Наприклад,

```
DataInputStream myStream = new DataInputStream
        (new FileInputStream("input.txt"));
```

і далі вже

```
String s = myStream.readLine();
```

щоб прочитати рядок.

Класи, похідні від OutputStream

Ієрархію класів, похідних від *OutputStream*, наведено на рисунку 11. Як бачимо, ці класи є віддзеркаленням щойно розглянутих нами класів. Отже, детально з їх роботою можете ознайомитися в літературі по Java, а ми розглянемо лише один з них – клас *PrintStream.*

З **класом *PrintStream*** ми вже зустрічалися. Коли ми створювали першу програму, записуючи

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

ми використали метод *println()*, визначений в класі *PrintStream*. Тепер можна прояснити деякі моменти, які тоді залишилися нез'ясованими. Справа в тому, що *out* – це екземпляр класу *PrintStream*. І цей екземпляр є одним з полів класу *System*, в якому є ще два поля – *in* і *err*. Залишилося тільки вияснити: в якому випадку ми, звертаючись до полів (або методів) якогось класу, пишемо ім'я класу, а не ім'я об'єкта. Відповідь – коли це поле *static*. Сам же клас *System* має описувач *final* і містить ще декілька цікавих методів (також статичних).

Сам же клас *PrintStream* містить декілька різновидів методів *print()* і *println()* і призначений для форматного виведення даних різних типів з метою їх візуального подання у вигляді текстового рядка.

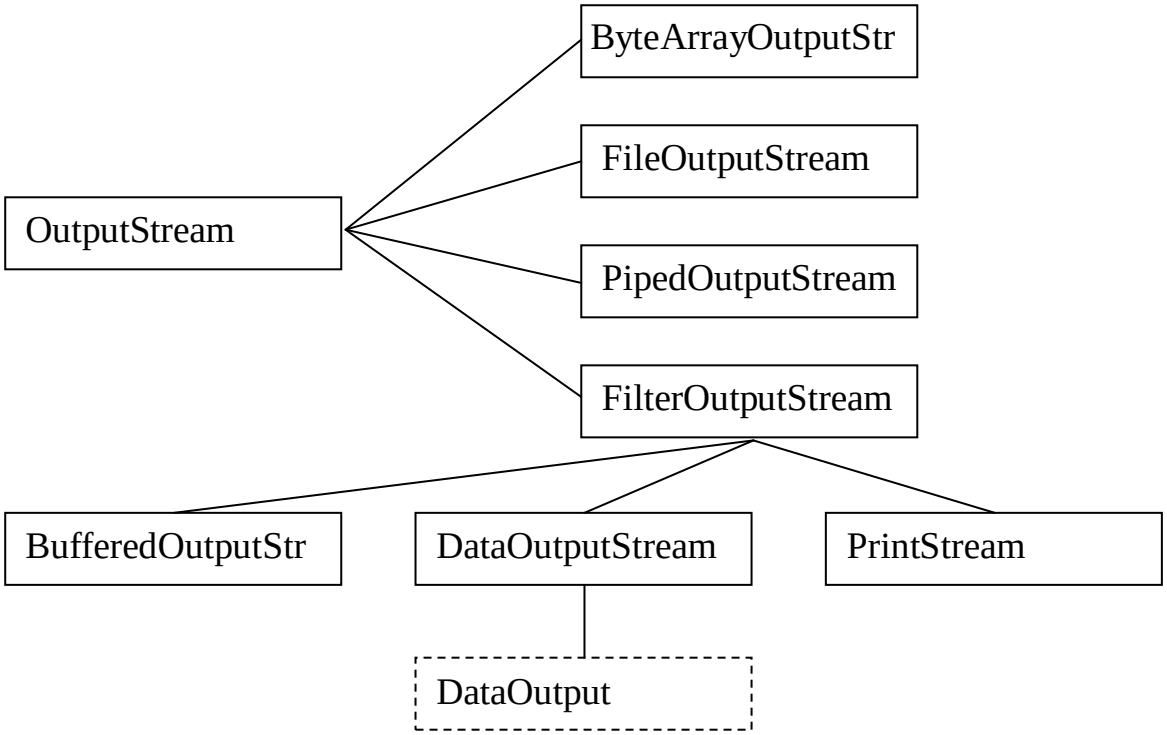

Рисунок 11 – Класи, похідні від *OutputStream*

Приклад. Фрагмент програми, яка при натисненні на кнопку *Save* записує в файл зміст текстового поля.

```
public boolean action(Event evt, Object arg)
{
    if (arg == "Save")
        {
          String str1 = textField1.getText();
          try
          {
               DataOutputStream myStream = new DataOutputStream(
                    new FileOutputStream("output.txt"));
                    myStream.writeBytes(str1);
                    myStream.close();
          }
          catch (Exception e) {}
          return true;
        }
        else
          return false;
}
```

Коротенький коментар до цього фрагмента. Метод *writeBytes(str1)* записує в файл один рядок у вигляді послідовності ASCII-символів. Саме він і викидає виняток, який ми мусимо перехопити в блоці *try-catch.*

Контрольні питання та завдання

1. *Наведіть коротку характеристику потокових класів, що мають своїм безпосереднім суперкласом клас Object.*
2. *Поясніть, як працює механізм потокового введення-виведення в Java.*
3. *Змініть останню програму так, щоб за допомогою неї можна було зберігати в файлі зміст не текстового поля, а текстової області.*

4.2 Створення мережних програм. Протоколи TCP/IP

Як основу для подальшої роботи розглянемо загальну схему побудови набору протоколів TCP/IP. Мережна модель TCP/IP може бути розділена на чотири рівні (згори – донизу): рівень додатків, транспортний рівень, мережний рівень, канальний рівень. На кожному рівні працюють свої протоколи, у протоколів кожного рівня є своє чітко визначене коло обов'язків.

Нас, як програмістів, цікавлять три протоколи: IP, TCP та UDP.

IP – це протокол мережного рівня, який визначає куди саме будуть передаватись дані. IP характеризується як ненадійний протокол, тому що він не знаходить помилок і не виконує повторне пересилання даних. Що таке IP- та DNS-адреса можна прочитати в спеціальній літературі.

Протоколи TCP та UDP відносяться до транспортного рівня. Протокол TCP забезпечує надійний зв'язок на основі логічного з'єднання з неперервним потоком даних. Доставка TCP-пакета схожа з доставкою листа за замовленням.

З неперервним потоком даних. TCP забезпечує механізм передачі, що дозволяє пересилати довільну кількість байтів. Дані нарізаються на сегменти певної довжини, завдяки чому для рівня додатків емулюється неперервний потік даних.

На основі логічного з'єднання. Перед початком передачі даних TCP встановлює з віддаленою машиною з'єднання, обмінюючись службовою інформацією. Цей процес носить назву handshaking – рукостискання. Комп'ютери «домовляються» між собою про основні параметри зв'язку.

Надійний. Якщо сегмент TCP втрачено або зіпсовано, реалізація TCP це знайде та повторно передасть необхідний сегмент.

Доставка UDP-пакета схожа з доставкою поштової картки, яку кидають у скриньку сподіваючись, що хтось її звідти рано чи пізно дістане.

Протокол UDP має такі особливості:

1) оснований на повідомленнях, тобто UDP дозволяє додаткам пересилати інформацію у вигляді повідомлень (дейтаграм), які є одиницями передачі даних в UDP, а розподіл даних по окремих дейтаграмах має виконувати сам додаток;
2) працює без встановлення логічного з'єднання, тобто ніякого обміну службовою інформацією на початку з'єднання (інформація передається в припущенні, що приймальна сторона її очікує);
3) є ненадійним. Це означає, що UDP не має ані вбудованого механізму знаходження помилок, ані повторного пересилання спотворених або втрачених даних.

Як бачимо, майже все говорить на користь TCP. Тоді для чого ж нам UDP? Може тому що програмісту непросто реалізувати усі ті плюси, які ми виставили TCP? Навпаки, все це вже реалізовано і все це у нас є. Але справа в тому, що протокол TCP потребує великих додаткових витрат, а це не завжди виправдано. Приклад використання UDP – служба часу.

Універсальна адреса ресурсів URL

Адреса IP дозволяє ідентифікувати вузол, але його недостатньо для ідентифікації ресурсів, що є на цьому вузлі (працюючі додатки або файли). Причина очевидна – на вузлі, що має одну адресу IP, може існувати багато різних ресурсів.

Для посилання на ресурси мережі Інтернет застосовується так звана універсальна адреса ресурсів *URL (Universal Resource Locator*). У загальному вигляді ця адреса виглядає таким чином:

[protocol]://host[:port][path]

Рядок адреси починається з протоколу *protocol*, який повинен бути використаний для доступу до ресурсу. Документи HTML, наприклад, передаються з серверу Web видаленим користувачам за допомогою протоколу HTTP. Файлові сервери в мережі Інтернет працюють з протоколом FTP.

Для посилання на мережні ресурси через протокол HTTP використовується така форма універсальної адреси ресурсів URL:

http://host[:port][path]

Параметр host обов'язковий. Він повинен бути вказаний як доменна адреса або як адреса IP (у вигляді чотирьох десяткових чисел). Наприклад:

http://www.sun.com
http://157.23.12.101

Необов'язковий параметр *port* задає номер порту для роботи з сервером. За замовчуванням для протоколу HTTP використовується порт з номером 80, проте для спеціалізованих серверів Web це може бути і не так.

Номер порту ідентифікує програму, що працює у вузлі мережі TCP/IP і взаємодіє з іншими програмами, розташованими на тому ж або на іншому вузлі мережі. Якщо ми розробляємо програму, що передає дані через мережу TCP/IP з використанням, наприклад, інтерфейсу сокетів Windows Sockets, то при створенні каналу зв'язку з виділеним комп'ютером ви повинні вказати не тільки адресу IP, але і номер порту, який буде використаний для передачі даних. Наприклад, так потрібно указувати в адресі URL номер порту:

http://www.myspecial.srv:82

Тепер займемося параметром *path*, що визначає шлях до об'єкта.

Зазвичай будь-який сервер Web або FTP має кореневий каталог, в якому розташовані підкаталоги. Як у кореневому каталозі, так і в підкаталогах сервера Web можуть знаходитися документи HTML, двійкові файли, файли з графічними зображеннями, звукові і відео-файли, розширення сервера у вигляді програм CGI або бібліотек динамічного компонування, що доповнюють можливості сервера.

Якщо як адреси URL вказати навігатору тільки доменне ім'я сервера, то сервер перешле навігатору свою головну сторінку. Ім'я файла цієї сторінки залежить від сервера. Більшість серверів на базі операційної системи UNIX посилає за замовчуванням файл документа з ім'ям *index.html*. Інші сервери Web можуть використовувати для цієї мети ім'я *default.htm* або будь-яке ще, визначене при установленні сервера, наприклад, *home.html* або *home.htm*.

Для посилання на конкретний документ HTML або на файл будь-якого іншого об'єкта необхідно вказати в адресі URL його шлях, що включає ім'я файла, наприклад:

http://www.glasnet.ru/~frolov/index.html
http://www.dials.ccas.ru/frolov/home.htm

Кореневий каталог сервера Web позначається символом /. Якщо для протоколу HTTP шлях не заданий, то використовується кореневий каталог.

Клас URL в бібліотеці класів Java

Для роботи з ресурсами, заданими своїми адресами URL, в бібліотеці класів Java є дуже зручний і могутній клас з назвою *URL*. Простота створення мережних додатків з використанням цього класу значною мірою спростовує поширену думку про складність мережевого програмування. Інкапсулюючи в собі складні процедури, клас *URL* надає в наше розпорядження невеликий набір простих у використовуванні конструкторів і методів.

Конструктори класу URL:

public URL(String url);

public URL(String prot,String host,int port,String file);
public URL(String prot,String host,String file);
public URL(URL context, String spec);

Перший з них створює об'єкт URL для мережного ресурсу, адреса URL якого передається конструктору у вигляді текстового рядка через єдиний параметр *url*, в процесі створення об'єкта перевіряється задана адреса URL, а також наявність вказаного ресурсу. Якщо адреса вказана неправильно або заданий ресурс відсутній, виникає виняток *MalformedURLException*.

Другий варіант конструктора класу URL допускає роздільне вказання протоколу, адреси вузла, номера порту і імені файла.

Третій варіант припускає використання номера порту за замовчуванням. Наприклад, для протоколу HTTP це порт з номером 80.

І, нарешті, четвертий варіант конструктора допускає вказання контексту адреси URL і рядка адреси URL. Рядок контексту дозволяє вказати компоненти адреси URL, що відсутні в рядку *url*, такі як протокол, ім'я вузла, файла або номер порту.

Методи класу URL:

openStream() – дозволяє створити вхідний потік для читання файла ресурсу, пов'язаного із створеним об'єктом класу URL. Для виконання операції читання із створеного таким чином потоку ми можемо використовувати будь-який метод *read()*, визначений в класі *InputStream*. Дану пару методів (*openStream()* з класу *URL* і *read()* з класу *InputStream*) можна застосувати для вирішення задачі отримання вмісту двійкового або текстового файла, що зберігається в одному з каталогів сервера Web. Зробивши це, звичний додаток Java або аплет може виконати локальну обробку одержаного файла на комп'ютері віддаленого користувача;

getContent() – визначає і одержує вміст мережного ресурсу, для якого створений об'єкт URL. Практично можна використовувати цей метод для отримання текстових файлів, розташованих в мережних каталогах. На жаль, даний метод непридатний для отримання документів HTML, оскільки для даного ресурсу не визначений обробник вмісту, призначений для створення об'єкта. Метод *getContent()* не здатний створити об'єкт ні з чого іншого, окрім текстового файла. Дана проблема, проте, розв'язується дуже просто – достатньо замість методу *getContent()* використовувати описану вище комбінацію методів *openStream()* з класу *URL* і *read()* з класу *InputStream*;

getHost() – визначає ім'я вузла, що відповідіє даному об'єкту URL;

getFile() – дозволяє одержати інформацію про файл, пов'язаний з даним об'єктом URL;

getPort() – визначає номер порту, на якому виконується зв'язок;

getProtocol() – визначає протокол, з використанням якого встановлено з'єднання з ресурсом, заданим об'єктом URL;

getRef() – повертає текстовий рядок посилання на ресурс, що відповідає даному об'єкту URL;

hashCode() – повертає хеш-код об'єкта URL:

sameFile() – визначає, чи посилаються два об'єкти класу URL на один і той самий ресурс. Якщо об'єкти посилаються на один ресурс, метод повертає *true*, якщо ні – *false*.

equals() – визначає ідентичність адрес URL, заданих двома об'єктами. Якщо адреси URL ідентичні, метод повертає true, якщо ні - значення false.

toExternalForm() – повертає текстовий рядок зовнішнього подання адреси URL, визначеної даним об'єктом класу URL;

toString() – повертає текстовий рядок, що описує даний об'єкт;

openConnection() – призначений для створення каналу між додатком і мережним ресурсом, поданим об'єктом класу URL. Якщо ми створюємо додаток, який дозволяє читати з каталогів сервера Web текстові або двійкові файли, можна створити потік методом *openStream()* або одержати вміст текстового ресурсу методом *getContent()*. Проте, є і інша можливість. Спочатку можна створити канал, як об'єкт класу *URLConnection*, викликавши метод *openConnection()*, а потім створити для цього каналу вхідний потік методом *getInputStream()* з класу *URLConnection*. Така методика дозволяє визначити чи встановити перед створенням потоку деякі характеристики каналу, наприклад, задати кешування. Проте найцікавіша можливість, яку надає цей метод, полягає в організації взаємодії додатка Java і сервера Web.

Приклад програми використання адреси URL для отримання Web-сторінки:

```
import java.net.*;
import java.io.*;

public class ReadURL
{   public static void main( String agr[])
   { try {                    //Об'являємо адресу URL
            URL myURL = new URL("http://www.yahoo.com/");
                              //Створюємо потік введення
            InputStream in = myURL.openStream();
            int ch;
                              //Читаємо з потоку і виводимо на екран
            while((ch=in.read())!=-1)
            System.out.print((char)ch);
            in.close();
      }
```

```
    catch(MalformedURLException me)
      {  System.out.println(me.getMessage());
           System.exit(0);
       }
    catch(IOException e)
      {  System.out.println(e.getMessage());
           System.exit(0);
      }
  }
}
```

Клас URLConnection в бібліотеці класів Java

Якщо треба не тільки отримати інформацію з хоста, але дізнатись і про її тип (текст, гіпертекст, архівний файл, зображення, звук), або дізнатись про довжину файла, або передати інформацію на хост, то спочатку необхідно методом *openConnection()* створити об'єкт класу *URLConnection.*

Але після створення об'єкта з'єднання ще не встановлено, і можна задати параметри зв'язку. Наведемо деякі з методів, призначених для цього.

Методи для встановлення параметрів з'єднання:

setDoInput(boolean doinput) – встановлює можливість використання потоку для введення;

setDoOutput(boolean dooutput) – встановлює можливість використання потоку для виведення;

setUseCaches(boolean usecaches) – включення або відключення кешування;

setDefaultUseCaches(boolean defaultusecaches) – включення або відключення кешування за замовчуванням;

setIfModifiedSince(long ifmodifiedsince) – установлення дати модифікації документа.

connect() – установлення з'єднання з об'єктом, на який посилається об'єкт класу URL.

Методи для отримання параметрів з'єднання:

```
public boolean getDefaultUseCaches();
public boolean getUseCaches();
public boolean getDoInput();
public boolean getDoOutput();
public long getIfModifiedSince();
```

Методи для витягання інформації із заголовка протоколу HTTP:

getContentEncoding() – повертає вміст *content-encoding* (кодування ресурсу, на який посилається URL), або *null*, якщо сервер його не вказав;

getContentLength() – повертає довжину отриманої інформації (розмір документа в байтах), або значення –1, якщо сервер її не вказав;
getContentType() – повертає тип інформації *content-type* (тип вмісту), тобто рядок типу *"text/html"* або *null* (якщо сервер не вказав);
getDate() – повертає дату посилки ресурсу в секундах з 01.01.1970;
getLastModified() – повертає дату зміни ресурсу в секундах з 01.01.1970;
getExpiration() – повертає дату застарівання ресурсу в секундах з 1.01.1970.

Інші методи в класі *URLConnection* дозволяють одержати всі заголовки або заголовки із заданим номером, а також іншу інформацію про з'єднання.

Приклад програми, що пересилає рядок текста за адресою URL.

```
import java.net.*;
import java.io.*;
class PostURL
{ public static void main(String[] args)
  { String req = "This text is posting to URL";
        try                 // Вказуємо URL потрібної програми
        {     URL url = new URL("http://www.bmv.ru/cgi-bin/soe.pl");
                            // створюємо об'єкт
              URLConnection uc=url.openConnection();
                            // Встановлюємо параметри з'єднання
              uc.setDoOutput(true);    uc.setDoInput(true);
              uc.setUseCaches(false);
              uc.connect();       // Встановлюємо зв'язок
                                  // Відкриваємо вихідний потік
              DataOutputStream dos = new
                            DataOutputStream(uc.getOutputStream());
              dos.writeBytes(req);      // записуємо в нього рядок
              dos.close();              // закриваємо потік
                            // Відкриваємо вхідний потік для
              відповідіBufferedReader br = new BufferedReader(new
                                  InputStreamReader(uc.getInputStream()));
              String res = null;  // читаємо з потоку, поки є що читати
              while ((res = br.readLine()) != null)
                    System.out.println(res);
              br.close();         // закриваємо потік
        }
      catch(MalformedURLException me)  {System.err.println(me);}
      catch(UnknownServiceException se)  {System.err.println(se);}
      catch(IOException ioe)             {System.err.println(ioe);}
       }
}
```

Сокети TCP

У бібліотеці класів Java є дуже зручний засіб, за допомогою якого можна організувати взаємодію між додатками Java і аплетами, що працюють як на одному і тому ж, так і на різних вузлах мережі TCP/IP. Це засіб, що «народився» в світі операційної системи UNIX, – так звані сокети (sockets).

Що таке сокети? Можна уявити собі сокети у вигляді двох розеток, в які включений кабель, призначений для передачі даних через мережу. Переходячи до комп'ютерної термінології, скажемо, що сокети – це програмний інтерфейс, призначений для передачі даних між додатками.

Перш ніж додаток зможе виконувати передачу або прийом даних, він повинен створити сокет, вказавши при цьому адресу вузла IP, номер порту, через який передаватимуться дані, і тип сокета.

З адресою вузла IP ми вже стикалися. Номер порту служить для ідентифікації додатка. Зауважимо, що існують так звані "добре відомі" (well known) номери портів, зарезервовані для різних додатків. Так, порт з номером 80 зарезервований для використання серверами Web при обміні даними через протокол HTTP.

Що ж до типів сокетів, то їх два – потокові і дейтаграмні.

За допомогою ***потокових сокетів*** ми можемо створювати канали передачі даних між двома додатками Java у вигляді потоків, які ми вже розглядали раніше. Потоки можуть бути:

- вхідними або вихідними,
- звичними або форматованими,
- з використанням або без використання буферизації.

Організувати обмін даними між додатками Java з використанням потокових сокетів не важче, ніж працювати через потоки зі звичними файлами. Помітимо, що потокові сокети дозволяють *передавати дані тільки між двома додатками*, оскільки вони припускають створення каналу між цими додатками.

Проте іноді потрібно забезпечити взаємодію декількох клієнтських додатків з одним серверним або декількох клієнтських додатків з декількома серверними додатками. В цьому випадку можна або створювати в серверному додатку окремі задачі і окремі канали для кожного клієнтського додатка, або скористатися дейтаграмними сокетами. ***Дейтаграмні сокети*** дозволяють передавати дані відразу всім вузлам мережі, хоча така можливість рідко використовується і часто блокується адміністраторами мережі.

Для передачі даних через дейтаграмні сокети не потрібно створювати канал – дані посилаються безпосередньо тому додатку, для якого вони призначені з використанням адреси цього додатка у вигляді сокета і номера порту. При цьому один клієнтський додаток може

обмінюватися даними з декількома серверними додатками або навпаки, один серверний додаток – з декількома клієнтськими.

На жаль, дейтаграмні сокети не гарантують доставку переданих пакетів даних. Навіть якщо пакети даних, передані через такі сокети, дійшли до адресата, не гарантується, що вони будуть одержані в тій самій послідовності, в якій були передані. Потокові сокети, навпаки, гарантують доставку пакетів даних, причому в правильній послідовності.

Причина відсутності гарантії доставки даних при використанні дейтаграмних сокетів полягає у використанні такими сокетами протоколу UDP, який, у свою чергу, базується на протоколі з негарантованою доставкою IP. Потокові сокети працюють через протокол гарантованої доставки TCP.

***Сокет (socket*) –** це описувач мережного з'єднання. Сокет TCP використовує протокол TCP, успадковуючи всі характеристики цього протоколу. Для створення сокета TCP необхідно мати таку інформацію:

- IP-адреси клієнта та сервера;
- порти, які використовують додатки на клієнтському та серверному боці.

Сервер – це комп'ютер, який очікує звертань від різних машин із запитом конкретних ресурсів. Відповідно ***клієнти*** – це комп'ютери, які звертаються з цими запитами до сервера. Не слід думати, що сервер – це головний, а клієнт – це підлеглий. В принципі і клієнт, і сервер – рівноправні в тому сенсі, що і той, і інший можуть і пересилати, і отримувати дані. Але щоб відбулась телефонна розмова хтось має подзвонити (клієнт), а хтось має чергувати біля телефону і своєчасно зняти слухавку (сервер).

Щоб почати обмін через сокет, додаток-клієнт має прив'язатися до конкретного порту. Порт – це абстракція (до речі, як і сокет), яка дозволяє розділити різні додатки, що можуть входити до мережі з одного і того самого комп'ютера. Іншими словами, уявіть ситуацію, коли в багатозадачній системі з одного комп'ютера різні програми шлють запити і очікують відповіді. Як відрізнити, кому яка відповідь призначена? Щоб уникнути плутанини, різні додатки мають прив'язуватися до різних портів.

Ось ми і плавно переходимо від теорії до практики. Хто і як має визначати номери портів? Відповідь – сервер, а він має знати, як розподіляються номери портів. За загальноприйнятими погодженнями, це мають бути номери, більші за 1024.

4.3 Робота з потоковими сокетами

Всю роботу в мережі інкапсульовано в пакеті *java.net*. В Java є два класи, які дозволяють створювати мережні програми на основі сокетів: *Socket* та *ServerSocket*.

Клас Socket

Клас *Socket* використовується для звичайного двобічного обміну даними. Він має чотири конструктори:

public Socket (String host, int port) throws UnknownHostException,
IOException,

де *host* – це адреса комп'ютера, з яким треба установити зв'язок; *port* – номер порту на комп'ютері, з яким треба установити зв'язок. Наприклад,

```
try {
        Socket socket = new Socket("10.0.9.1", 4444);
        . . .
}
catch (. . .) {   }
catch (. . .) {   }
```

Другий конструктор першим параметром отримує змінну класу *InetAddress*:

public Socket (InetAddress address, int port) throws
UnknownHostException, IOException

Об'єкт класу *InetAddress* містить IP-адресу комп'ютера в мережі. Цікаво, що цей клас не має конструкторів, але має цілу низку так званих фабричних (статичних) методів, які повертають екземпляри класу *InetAddress*. Наприклад,

```
try {
        InetAddress address=InetAddress.getByName("10.0.9.1");
        Socket socket = new Socket(address, 4444);
}
catch (. . .) {  }
catch (. . .) {  }
```

Звичайно, в цьому прикладі створення сокета можна було б зробити в одному операторі. Також заміть IP- можна використовувати DNS-адресу.

Третій та четвертий конструктори аналогічні першим двом, але мають ще додатковий третій параметр, який дозволяє задати бульове значення. Якщо воно встановлюється в *true*, використовується протокол на основі потоків даних (наприклад, TCP, як і за замовчуванням в перших двох конструкторах), якщо в *false* – протокол на основі дейтаграм (наприклад, UDP).

Клас *Socket* також має методи, що дозволяють читати та писати в нього: *getInputStream()* і *getOutputStream()*, які повертають потоки введення та виведення. Для зручності використання ці потоки звичайно надбудовуються добре відомими вам класами *DataInputStream* і *PrintStream,*

відповідно. І метод *getInputStream()*, і метод *getOutputStream()* генерує виняток *IOException*, який треба перехопити та обробити.

```
try
{
   Socket socket = new Socket(“10.0.9.1”);
  DataInputStream input =
                        new DataInputStream(socket.getInputStream());
  PrintStream output = new PrintStream(socket.getOutputStream());
}
catch (UnknownHostException e)
 {
        System.err.println(“Unknown host: “ + e.toString());
        System.exit(1);
}
catch (IOException e)
 {
        System.err.println(“Failed I/O: “ + e.toString());
        System.exit(1);
}
```

Наразі, щоб послати повідомлення та отримати відповідь на один рядок, достатньо використати надбудовані потоки

```
output.println(“test”);
String response input.readLine();
```

Методи *getInetAddress()* і *getPort()* дозволяють визначити адресу IP і номер порту, пов'язані з даним сокетом (для видаленого вузла):

```
public InetAddress getInetAddress();
public int getPort();
```

Метод getLocalPort() повертає для даного сокета номер локального порту:

```
public int getLocalPort();
```

Завершуючи обмін даними треба закрити потоки, а потім і сам сокет:

```
output.close();
input.close();
socket.close();
```

Клас ServerSocket

Щоб створити сокет TCP, необхідно скористатися класом, який дозволяє прив’язатися до порта та очікувати підключення клієнтів. При кожному підключенні буде створено екземпляр класу *Socket*. *ServerSocket* має два конструктори:

public ServerSocket(int port) throws IOException;
public ServerSocket(int port, int count) throws IOException;

Перший з них створює сокет, підключений до вказаного порту. За замовчуванням в черзі очікування підключення може знаходитися до 50 клієнтів. Другий конструктор дозволяє задавати довжину черги.

Після створення об'єкта *ServerSocket* можна використовувати метод *accept()* для очікування підключення клієнтів. Цей метод блокується до моменту підключення клієнта, а потім повертає об'єкт *Socket* для зв'язку з клієнтом. Блокування означає, що програма входить у внутрішній нескінченний цикл, який завершується за певних умов. Іншими словами, поки клієнт не підключиться, наступний оператор виконуватися не буде.

Наступний текст створює об'єкт *ServerSocket* з портом 4444, очікує підключення і далі створює потоки, через які буде виконуватись обмін даними з клієнтом, що підключився:

```
try {
    ServerSocket server = newServerSocket(4444);
    Socket con = server.accept();
    DataInputStream inp=new DataInputStream(cont.getInputStream());
    PrintStream output = new PrintStream(cont.getOutputStream());
}
catch (IOException e)
{
    System.err.println("Failed I/O: " + e.toString());
    System.exit(1);
}
```

Приклади програм з використанням потокових сокетів

Приклад серверного додатка, що реалізує потоковий сокет.

```
import java.net.*;
import java.io.*;
import java.ntil.*;

public class MyServer extends Tread
 {
  ServerSocket srvsocket;
  boolean flag=true;
  int port= 8888;

     public MyServer()
     { try
          {              // Об'являємо порт прослуховування
```

```
                srvsocket=new ServerSocket(port);
                System.out.println("My-Server running...");
            }
        catch(IOException e)
            {
                System.err.println(e.getMessage());
                 System.exit(0);
            }
         }
public void run()
{
    Socket clientsocket;
   while(flag)
   { try
       {       // Прослуховуємо порт
            clientsocket=srvsocket.accept();
                    //Записуємо в нього дані
             PrintWriter out =
                        new PrintWriter(clientsocket.getOutputStream());
            out.println("MY-SERVER RESPONSE!!!");
            out.println("Connection at:"+new Date().toString());
            out.flush();  //Закриваємо об'єкти
            out.close();
            clientsocket.close();   // і завершуємо роботу потоку
            srvsocket.close();
            flag=false;
       }
      catch(IOException e)
       {
            System.err.println(e.getMessage());
            System.exit(0);
       }
    }
  }
 public static void main(String[] agrs)
 {
       MySrver msrv = new MyServer();
       msrv.start();
 }
}
```

Приклад клієнтського додатка, що реалізує потоковий сокет.

```
import java.net.*;
import java.io.*;

public class MyClient
 {  public static void main(String[ ] agrs)
   {
    System.out.println("Client running...");
    try
     {              //Визначаємо адресу і порт підключення
            InetAddress local = InetAddress.getlocalHost();
            int port=8888;
                   //Виконуємо підключення
            Socket mysocket = new Socet(local,port);
                   //Зчитуємо потік даних по мережі
            InputStream in = mysocket.getInputStream();
            int ch;
            while ((ch=in.read())!=-1)
                   System.out.print((char)ch);
             mysocket.close();
            in.close();
     }
    catch(IOException e) {}
   }
 }
```

Контрольні питання

1. *Дайте порівняльну характеристику протоколів TCP і UDP.*
2. *Наведіть приклади практичних задач, де можна було б використати протоколи TCP і UDP.*
3. *Що спільного та відмінного в роботі серверної та клієнтської частини?*
4. *Як уникнути блокування наступної за методом accept() частини програми, якщо необхідно, щоб цей код продовжував виконуватися?*

4.4 Використання сокетів UDP

Для роботи з сокетами UDP додаток має створити сокет на базі класу *DatagramSocket*, а також підготувати об’єкт класу *DatagramPacket*, в який буде занесено блок даних для прийому/передавання.

Канал, а також вхідні та вихідні потоки створювати не треба. Дані передаються та приймаються методами *send()* і *receive()*, визначеними в класі *DatagramSocket*.

Клас DatagramSocket

Розглянемо конструктори та методи класу *DatagramSocket*, призначеного для створення та використання сокетів UDP або дейтаграмних сокетів.

В класі *DatagramSocket* визначено два конструктори:

```
public DatagramSocket(int port);
public DatagramSocket();
```

Перший з цих конструкторів дозволяє визначити порт для сокета, інший припускає використання будь-якого вільного порту.

Звичайно серверні додатки працюють з використанням заздалегідь визначеного порту, номер якого є відомим для додатків-клієнтів. Тому для серверних додатків ліпше використовувати перший з наведених вище конструкторів.

Клієнтські додатки, навпаки, часто-густо використовують будь-які вільні на локальному вузлі порти, тому для них ліпшим є конструктор без параметрів.

Звичайно перший з цих конструкторів використовують для серверів, які, як правило, знають самі і мають повідомити клієнтам номер свого порту, другий – для клієнтів.

До речі, за допомогою методу *getLocalPort()* додаток завжди може довідатися номер порту, що його закріплено за даним сокетом:

```
public int getLocalPort();
```

Прийом і передавання даних на дейтаграмному сокеті виконується за допомогою методів *receive()* і *send(),* відповідно:

```
public void receive(DatagramPacket p);
public void send(DatagramPacket p);
```

Як параметр цим методам передається посилання на пакет даних, визначений як об’єкт класу *DatagramPacket*.

Ще один метод в класі *DatagramSocket*, яким ви маєте скористатися, це метод *close()*, призначений для закриття сокета:

```
public void close();
```

Клас DatagramPacket

Перед тим, як приймати або передавати дані з використанням методів *receive()* і *send(),* ви маєте підготувати об’єкти класу *DatagramPacket*. Метод *receive()* запише в такий об’єкт прийняті дані, а метод *send()* – перешле дані з об’єкта класу *DatagramPacket* вузла, адреса якого вказана в пакеті.

Підготовка об’єкта класу *DatagramPacket* для прийому пакетів виконується за допомогою такого конструктора:

public DatagramPacket(byte buf[], int length);

Цьому конструктору передається посилання на масив *buf*, в який треба буде записати дані, та розмір цього масиву *length*.

Якщо вам треба підготувати пакет для передавання, скористайтеся конструктором, який додатково дозволяє задати адресу IP *addr* і номер порту *port* вузла призначення:

public DatagramPacket(byte buf[], int length, InetAddress addr, int port);

Таким чином, інформація про те, на який вузол і на який порт необхідно доставити пакет даних, зберігається не в сокеті, а в пакеті, тобто в об'єкті класу *DatagramPacket*.

Крім цих конструкторів, в класі *DatagramPacket* визначено чотири методи, що дозволяють отримати дані та інформацію про адресу вузла, з якого прийшов пакет, або для якого призначено пакет.

Метод *getData()* повертає посилання на масив даних пакета:

public byte[] getData();

Розмір пакета, дані з якого зберігаються в цьому масиві, легко визначити за допомогою методу *getLength()*:

public int getLength();

Методи getAddress() і getPort() дозволяють визначити адресу та номер порту вузла, звідки прийшов пакет, або вузла, для якого призначено пакет:

public InetAddress getAddress();
public int getPort();

Якщо ми створюємо клієнт-серверну систему, в якій сервер має заздалегідь відому адресу та номер порту, а клієнти – довільні адреси та різні номери портів, то після отримання пакета від клієнта сервер може визначити за допомогою методів *getAddress()* і *getPort()* адресу клієнта для встановлення з ним зв'язку.

Приклад серверного додатка, що реалізує дейтаграмний сокет

```
import java.net.*;

public class DatagramServsr extends Thread
{       byte[] buf = new byte[100];
        int port=8008;

        public void run()
        {   try         //Визначаємо локальну адресу
            {     InetAddress iosal = InetAddress.getLocalHost();
                        //Визначаємо сокет дейтаграми
                DatagramSocket srvsocket=new DatagramSocket(port,local);
```

```
                //Визначаємо пакет дейтаграми
            DatagramPacket pack=new DatagramPacket(buf,buf.length);
            while (true)
            {       srvsocket.receive(pack); //Отримуємо дейтаграму
                    String srt=new String(pack.getData(),0,pack.getLength());
                    System.out.println(str);
            }
        }
      catch(Exception e)
        { System.err.println(e.getMessage());}
      }
  public static void main(String agr[])
  {
       System.out.println("Datagram server running...");
      DatagramServsr my= new DatagramServsr();
      my.start();
  }
}
```

Приклад клієнтського додатка, що реалізує дейтаграмний сокет.

```
import java.net.*;
public class DatagramClient
{
      public static void main(String agr[])
      {             // Об'являємо масив байтів для пакета дейтграми
            byte[ ] buf= new bute[100];
                  // Ініціалізація порту
            int port=8008;
            int cur=0;
            int ch;
            boolean flag = true;
             while(flage)
            { try
              {           // Отримуємо символ з клавіатури
                   ch=System.in.read();
                        // /Визначаємо локальну адресу
                  InetAddress local = InetAddress.getLocalHost();
                        // Записуємо символ в масив
                  buf[cur]=(byte)ch;
                  if(ch==-1) flag=false;
                  if(ch=='\n')
                  {             // Визначаємо сокет дейтаграми
```

```
                    DatagramSocket srvsocket = new DatagramSocket();
                        // Визначаємо пакет для відправки
                        // за адресою local і портом port
                    DatagramPacket pack =
                        new DatagramPacket(buf,buf.length,local,port);
                        //Посилаємо пакет через сокет
                    srvsocket.send(pack);
                    cur=-1 ;
                        //Обнуляємо масив
                    for(int i=0;i<buf.lenght;i++) buf[i]=0;
                }
            cur++;
            }
        catch(Exception e)
            {
                System.err.println(e.getMessage());
                break;
            }
        }
    }
}
```

Широкомовні пакети

Якщо адреса сервера невідома, клієнт може посилати широкомовні (рос. широковещательные) (broadcast) пакети, вказавши в об'єкті класу *DatagramPacket* адресу мережі. Така методика звичайно використовується в локальних мережах. Більш детально про широкомовні пакети можна прочитати в спеціальній літературі.

Наведемо фрагмент програми-сервера, який, отримавши запит від клієнта, витягує з нього всю необхідну інформацію та посилає відповідь:

```
public void startServing()
{
    DatagramPacket datagram;      // For a UDP datagram.
    InetAddress clientAddr;       // Address of the client.
    int clientPort;               // Port of the client.
    byte[] dataBuffer;            // To construct a datagram.
    String timeString;            // The time as a string.
    DatagramSocket timeSocket =
                    new DatagramSocket(TIME_PORT);
                        // Keep looping while we have a socket.
    while(keepRunning)
    {
```

```
            try         // Create a DatagramPacket to receive query.
            {
                dataBuffer = new byte[SMALL_ARRAY];
                datagram = new DatagramPacket
                        (dataBuffer, dataBuffer.length);
                timeSocket.receive(datagram);
                    // Get the meta-info on the client.
                clientAddr = datagram.getAddress();
                clientPort = datagram.getPort();
                    // Place the time into byte array.
                dataBuffer = getTimeBuffer();
                    // Create and send the datagram.
                datagram = new DatagramPacket(dataBuffer,
                     dataBuffer.length, clientAddr, clientPort);
                timeSocket.send(datagram);
            }
            catch(IOException excpt) {
                System.err.println("Failed I/O: " + excpt);
            }
        }
        timeSocket.close();
    }
```

Контрольні питання

1. *Дайте порівняльну характеристику класів Socket і DatagramSocket.*
2. *Наведіть сигнатуру основних методів, визначених в класі DatagramSocket.*
3. *Перерахуйте методи класу DatagramPacket та поясніть їх призначення.*

ЛІТЕРАТУРА

1. Г. Буч. Объектно-ориентированное проектирование с примерами применения: пер. с англ. – М.: Конкорд, 1992. – 512 с.
2. Д. Вебер, Технология JAVA в подлиннике: пер. с англ. – СПб.: БХВ-Петербург, 2001. – 1104 с.
3. Хабибуллин И.Ш. Самоучитель Java. – СПб.: БХВ-Петербург, 2002. – 464 с.
4. Бишоп Д. Эффективная работа: Java 2. – СПБ.: Питер Л.: Издательская группа BHV, 2002. – 592 с.
5. Глушаков С.В. Программирование на Java 2: Учебный курс. – Харьков: Фолио; М.: ООО «Издательство АСТ», 2001. – 536 с.
6. Майкл Морган. Java 2. Руководство разработчика.: Пер. с англ..: Уч. пос. – М.: Издательский дом “Вильямс”, 2000. – 720 с.
7. Джим Яворски, Пол Дж. Перроун. Система безопасности Java. Руководство разработчика.: Пер. с англ..: Уч. пос. - М.: Издательский дом “Вильямс”, 2001. – 528 с.
8. П. Ноутон, Г. Шилдт. Java 2: пер. с англ. – М.: Издательский дом “Вильямс”, 2002. – 1036 с.