ВСТУП

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.

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 самих значних кроків вперед в області розробки середовищ програмування за останні 20 років. HTML (Hypertext Markup Language – мова розмітки гіпертексту) був необхідний для статичного розміщення сторінок у "Всесвітній павутині" WWW (World Wide Web). Мова Java потрібна була для якісного стрибка в створенні інтерактивних продуктів для Internet.

Багато властивостей Java отримав від C і C++. Проектувальники Java свідомо пішли на це, оскільки знали, що знайомий синтаксис зробить 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.

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.

2 ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ JAVA

2.1 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 функцій84. 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 представляються 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 модель85. 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 програм86. ООП організує програму навколо її даних (тобто навколо об’єктів) і набору чітко визначених інтерфейсів з цими даними87. Об’єктно-орієнтовану програму можна характеризувати як *доступ до коду, що управляється даними* (data controlling access to data)88. Як ми побачимо далі, такий підхід має деякі організаційні переваги, а саме 89:

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 модель8A8B. До них відносять абстракцію, інкапсуляцію, успадкування і поліморфізм8C. Також їх часто називають основними принципами ООП.

Абстракція та інкапсуляція

Абстракція даних – введення типів даних, визначених користувачем і відмінних від базових8D. Ця концепція полягає у можливості визначати нові типи даних, з якими можна працювати так само, як і з основними типами даних8E. 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 тип8F.

Інкапсуляція – це механізм, який пов'язує код з даними, що ним обробляються, та зберігає їх як від зовнішнього впливу, так і від помилкового використання90. Інкапсуляцію можна уявити як захисну оболонку, яка запобігає доступу до коду та даних з іншого коду, що знаходиться зовні цієї оболонки91. Доступ до коду та даних в середині оболонки відбувається через чітко визначений інтерфейс92. 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 наслідків93.

Основою абстракції та інкапсуляції в Java є **клас**94. Клас визначає структуру та поведінку (дані і код) деякого набору об'єктів95. 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)96. Тому об'єкт іноді ще називають **екземпляром класу**97. Таким чином, клас – це логічна конструкція, а об'єкт – це фізична реальність98.

При створенні класу необхідно специфікувати код і дані, що складають цей клас99. Разом всі ці елементи називають *членами* (members) класу9A. Дані, що визначаються в класі, називають *змінними-членами* (member variables), або *полями* класу9B. Код, який оперує з цими даними (тобто функції, що знаходяться в середині класу), називають *методами-членами* (member methods), або просто *методами*9C. В правильно записаних Java-програмах методи визначають, як можна використовувати змінні-члени. Отже, поведінку та інтерфейс класу визначають методи, що оперують з даними його екземплярів9D.

Оскільки мета створення класу – це інкапсуляція складності, існують механізми приховування її реалізації в середині класу9E. Методи і поля класу можуть бути помічені як *private* (приватний, локальний) або *public* (загальний)9F. Модифікатор *public* вказує на все, що потрібно знати зовнішнім користувачам класуA0. Методи та поля з модифікатором *private* є доступними лише для методів даного класуA1. Будь-який код, що не є членом даного класу, не має доступу до *private*-методів і полівA2. Оскільки *private*-члени класу є доступними для інших частин програми тільки через *public*-методи класу, можна бути впевненим, що ніякі непотрібні дії не виконуютьсяA3.

Успадкування

Успадкування (наслідування) – це процес, за допомогою якого один об'єкт отримує властивості іншого об'єктаA4. Воно важливе, тому що підтримує концепцію ієрархічної класифікації. Переважною частиною знань можна управляти лише за допомогою ієрархічних (тобто організованих "згори донизу") класифікаційA5. Без застосування класифікацій кожен об'єкт потребував би явного визначення всіх своїх характеристикA6. Завдяки використанню успадкування об'єкт потребує визначення лише тих якостей, які роблять його унікальним у власному класіA7. Тому саме механізм успадкування дає можливість одному об'єкту бути специфічним екземпляром загального випадкуA8.

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** тварин, тоді як клас тварин називають **суперкласом** (батьківським класом, класом-пращуром).

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 класівA9.

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 спеціалізаціїAA. 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 батьківAB. Він не має не передбачуваних зв'язків з рештою коду в програміAC.

Крім того, у похідному класі успадковані функції можуть бути перевизначеніAD. Таким чином будують ієрархії класів, пов'язаних між собою. Якщо об'єкт наслідує свої атрибути від одного базового класу, це буде *просте успадкуванняAE*. Якщо об'єкт успадковує атрибути від кількох батьків, це буде *множинне успадкуванняAF*. 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 крапкою, 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 – обрати 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 кильку, то гавкає та біжить за нею. Якщо чує їжу, виділяє слину та біжить до миски. В обох ситуаціях працює одне й те саме почуття – нюх. Різниця полягає в тому, що саме він нюхає, тобто в типі даних, з якими оперує ніс собаки. Ту ж спільна 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?*

2.2 Основні поняття мови JAVA. Перша програма

Мова Java – це об’єктно-орієнтована мова програмування, що веде 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 на ній додатки. Java свідомо уникає арифметики з покажчиками й іншими ненадійними елементами, якими буяє 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**B7. Власне кажучи, створення аплетів – основне застосування для Java. Аплети здобули 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 на Java не можна писати нормальні додатки з вікнами. Ця мова програмування споконвічно була створена для звичайних додатків, що виконуються в Internet і в інтрамережах, і вже потім стала використовуватися для виготовлення 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 по Java:

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

На рисунку 1 схематично представлено основні етапи створення додатку 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

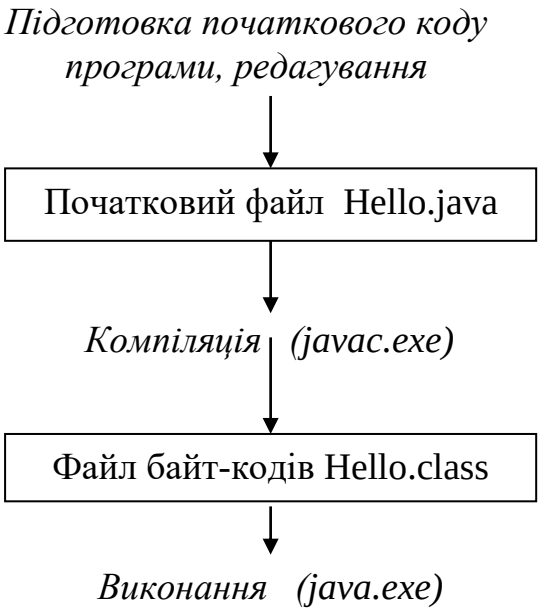

Рисунок 1 – Редагування, компіляція та запуск на виконання Java-програми

вихідного текстуB8.

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-кодB9. Таким чином досягається компроміс між ефективністю виконання Java-програм та їх незалежністю від платформиBA.

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 помилкуBB.

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

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 C і C++, який і визначає місце, з якого програма починає виконуватися (так звана точка входу)BD. 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 розташуванняBE. Модифікатор *static* говорить про те, що для всіх екземплярів класу Hello і в наслідуваних від нього класах існує лише один метод *main()*, поділюваний між усіма класами, що, можливо, будуть успадковані від HelloBF. Це допомагає уникнути появи безлічі точок входу в програму, що викликає помилку.

Через змінну-масив *args* типу *String* (рядок) передаються параметри командного рядка класуC0. У Java перший елемент списку параметрів відповідає першому параметру, а не імені програми, що запускається, як це прийнято в мовах C і C++. Доступ до нього можна здійснити через вираз *args[0]*C1. Рядок *System.out.println ("Hello, World!")* посилає рядок тексту в стандартний потік виведення, тобто на екранC2. 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, по завершенню чого переводить курсор на наступний рядок.

У Java **всі методи класу описуються тільки в середині цього класу**C3. 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 класC4.

Якщо засвоїти вміст пакету Java з ім'ям *java.awt*, що розшифровується як Abstract Windowing Toolkit C5 (Набір абстрактної роботи з віконною системою), то відкриються незлічимі можливості по створенню інтерфейсів і віконної графікиC6. Цей пакет забезпечує машинно-незалежний інтерфейс керування віконною системою будь-якої віконної операційної системиC7. До складу *java.awt* входять більше 40 класів, що відповідають за елементи графічного середовища користувача (GUI)C8. В основному awt застосовується при написанні аплетів для сторінок WebC9. При перегляді сторінки на Web-сервері аплет передається на машину користувача, де і запускається на виконанняCA.

Тепер ми можемо розглянути аплет, що 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 пакету *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 з 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 перемалювати дані на екрані. Тут за допомогою методу *drawString()* стандартного класу *Graphics* малюється 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?*

2.3 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 базовимиCB. Інша категорія – об'єктні типи даних, до яких належать класи, масиви й інтерфейсиCC. 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 "$"CD. Далі можуть йти і цифриCE. Наприклад, _Java - правильний ідентифікатор, а 1_$ - ніCF. Ще одне обмеження Java виникає з його властивості використовувати для збереження символів кодування Unicode, тобто можна застосовувати тільки символи, що мають порядковий номер більш 0x0 у розкладці символів UnicodeD0.

КоментаріD1. 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 документуванняD2. Після написання вихідного тексту утиліта автоматичної генерації документації збирає тексти таких коментарів в один файлD3.

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

- якщо в цифри немає суфікса і префікса, то це *десяткове число*D4;
- у *вісімкових числах* перед цифрою стоїть нульD5;
- для *шістнадцяткових чисел* префікс складається з нуля і букви X (0x чи 0X)D6.
- при додаванні до цифри букви L числу присвоюється тип *long (довге ціле)*D7**.**

Приклади: 23 (десяткове),
0675 (вісімкове),
0x9FA (шістнадцяткове),
456L (довге ціле)D8.

Числа із плаваючою крапкою. Для них передбачено два види описів D9:

- звичайне й
- експонентне.

При звичайному описі числа з плаваючою крапкою 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 DA. При додаванні суфіксів D і F виходять числа типів *double* і *float* DB. Наприклад, 2.71D і 0.981F.

Цілочисельні типиDC. У мові Java з'явився новий 8-бітний тип *byte*DD. Тип *int*, на відміну від аналогічного в C++, має довжину 32 біти DE. А для 16-бітних чисел передбачений тип *short*DF. У відповідності з усіма цими змінами тип *long* збільшився, ставши 64-бітним E0.

Чисельні типи даних наведено в таблиці 1:

Таблиця 1 – Чисельні типи даних

	Тип	Форма представлення	Значення за замовчу-ванням	Довжина (в бітах)	Максимальне значення
E1	*byte*	Ціле число зі знаком	0	8	127
E2	*short*	Ціле число зі знаком	0	16	32767
E3	*int*	Ціле число зі знаком	0	32	2147483647
E4	*long*	Ціле число зі знаком	0	64	порядку 10^{18}
E5	*float*	Число з плаваючою точкою	0	32	порядку 10^{38}
E6	*double*	Число з плаваючою точкою	0	64	порядку 10^{308}

Примітка. Всі чисельні типи в Java знакові E7.

У стандарт Java був введений тип *boolean*, якого так довго чекали програмісти, що використовують C++. Він може приймати лише два значення: *true і false*E8.

У порівнянні з C++ **масиви** Java перетерпіли значні зміни. По-перше, змінилися правила їхнього опису. Масив тепер може бути описаний двома такими способами E9:

type name[];

type[] name;

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

char[] arrayName;

arrayName[] = new char[100];

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

char array[] = new char[100];

Багатомірних масивів у Java немає ED, тому доводиться вдаватися до хитрощів. Наприклад, створити багатомірний масив можна як масив масивів EE:

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 елементівEF. Надалі цей клас може бути успадкований іншими, більш специфічними класами. Кожен з них додає ті властивості, які є унікальними для нього F0. В термінології Java клас, який успадковується, називається **суперкласом** (superclass)F1. Клас, який виконує успадкування, називається **підкласом** (subclass)F2. Тому підклас – це спеціалізована версія суперкласуF3. Він успадковує всі поля та методи суперкласу, та додає до них свої власні унікальні елементиF4. Щоб успадкувати клас, необхідно просто ввести визначення одного класу в інше, використовуючи ключове слово *extends*F5.

Розглянемо тепер, як описуються основні базові будівельні блоки мови Java – класи. Схема синтаксису опису класу такаF6:

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

де *Модифікатори* – ключові слова типу *public* 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 класуF7. При розробці самого класу вони не мають особливого значення, але вони дуже важливі при створенні інших класів або інтерфейсів на базі даного класуF8.

Модифікаторів доступу є три плюс ще один за замовчуваннямF9.

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

friendly **–** значення за замовчуванням (тобто слово *friendly* в описі класу ніколи не пишеться!)FC. *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 класуFD.

final-класи не можуть мати підкласів-нащадківFE. Тим самим ми втрачаємо одну з головних переваг ООПFF. Але іноді треба заборонити іншим класам змінювати поведінку розроблюваного класу (наприклад, якщо даний клас буде використаний як стандарт для обслуговування мережних комунікацій)100.

abstract **–** це клас, в якому є хоча б один абстрактний метод, тобто метод, для якого описаний лише заголовок (прототип функції), а саме тіло методу відсутнє101. Зрозуміло, що реалізацію цього відсутнього методу покладено на класи-нащадки102. Наприклад, створюється деякий клас для перевірки орфографії. Замість того, щоб закладати в нього перевірку української, російської, англійської орфографії, створюючи 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)103. Наприклад,

*public class MyClass extends Frame*104.

В основі ієрархії в java знаходиться клас *Object*105. Тому якщо навіть не використовується слово *extends* в описі класу, то створюється нащадок класу *Object*106.

Нагадаємо, що клас-нащадок успадковує всі дані та методи суперкласу107.

В мові Java відсутнє множинне успадкування (але є інтерфейси, які з успіхом замінюють його)108.

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 спадковість109. Про проблеми, що пов'язані з множинною спадковістю, можна прочитати в [1].

Конструктори. Створення екземплярів класу

Конструктор – це метод класу, що має особливе призначення10A. Зокрема він використовується для установки деяких параметрів

(наприклад, ініціалізації змінних) та виконання певних функцій (наприклад, виділення пам'яті)10B.

Конструктор має те ж саме ім'я, що і клас10C. Наприклад, 10D

MyClass(String name) {myName=name;}

Як відрізнити – це клас чи конструктор?

MyClass *MyClass()*

Так саме, як в C++, один клас може мати декілька конструкторів10E. У цьому випадку вони мають відрізнятися за своїми параметрами10F. Також в описі конструктору 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 класу110. Створимо екземпляр класу *MyClass*:

MyClass myClass1= new MyClass();

Екземпляр класу може бути полем іншого класу111. Так утворюється ієрархія за складом (спадковість реалізує ієрархію за номенклатурою)112.

```
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*113. При цьому відбувається 114:

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'єкт115. Правильно – в описі конструктора відсутній тип даних, що повертається116.

У чому полягає відмінність об'єктів від змінних базових типів? До базових типів завжди звертаються за значенням117. До об'єктних типів (до яких відносяться масиви, класи, інтерфейси) – завжди за посиланням118. Проілюструємо цю різницю на прикладі.

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

Чому дорівнює *x*119? 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’яті11A. Це ілюструється на рисунку 211B.

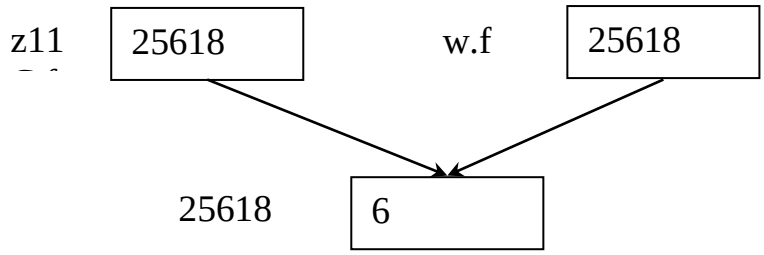

Рисунок 2 – Звертання до об’єктних типів

Контрольні питання

1. *Наведіть порівняльну характеристику стандартних типів даних мови C і базових типів даних мови Java.*
2. *Чи можуть модифікатори доступу сполучатися 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?*

2.4 Поля та методи класів

В класах існують два типи змінних: одні належать самому класу, інші ж – різним методам класу11D.

Ті змінні, що описані не в методах, але в середині даного класу, називають **полями** класу11E. Вони доступні для всіх методів даного класу (це як мінімум)11F.

Крім того, можна описувати змінні в середині метода класу120. Такі змінні є локальними для метода і доступні тільки для цього метода121.

Загальне правило: кожна змінна доступна лише в середині того блока (обмеженого фігурними дужками), в якому її описано122.

І останнє. Як поля, так і локальні змінні можуть бути як базового типу, так і екземплярами класу123.

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 місця124. Все, що описано *private*, доступно лише в середині класу125. Якщо у елемента взагалі не указаний модифікатор рівня доступу, то такий елемент буде видно з підкласів-нащадків і класів того ж пакета126. Саме цей рівень доступу використовується в Java за замовчуванням127. Якщо треба зробити, щоб елемент крім того був доступний зовні пакета, але тільки підкласам того ж класу, якому він належить, його слід описати як *protected*128. І, на сам кінець, якщо треба щоб елемент був доступний тільки підкласам, незалежно від того, чи знаходяться вони в даному пакеті, використовуйте комбінацію *private protected*129.

Таблиця 2 – Доступ до полів і методів для різних модифікаторів

	private	friendly	private protected	protected	public
той же клас	+	+	+	+	+
підклас в тому ж пакеті	-	+	+	+	+
незалежний клас в тому ж пакеті	-	+	-	+	+
підклас в іншому пакеті	-	-	+	+	+
незалежний клас	-	-	-	-	+

З полями та методами можуть використовуватись всі специфікатори доступу, з класами – лише *public* і значення за замовчуванням12A.

Інші модифікатори

Модифікатор ***static*. Ц**ей описувач може використовуватись як з полями, так і з методами12B. Для поля описувач *static* означає, що таке поле створюється в єдиному екземплярі незалежно від кількості об'єктів даного класу (звичайні поля – для кожного екземпляру класу)12C. Статичне поле існує навіть тоді, коли немає жодного екземпляра класу12D. Статичні поля розташовуються Java-машиною окремо від об'єктів класу в момент першого звертання до цього класу12E. 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*12F. Це пояснюється тим, що воно існує в одному екземплярі130.

По аналогії зі статичними полями статичні методи не прив'язані до конкретного об'єкта класу131. Коли викликається статичний метод, перед ним можна вказувати не посилання, а ім'я класу132. 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();133
    SomeClass.f();134
. . .
```

І, на сам кінець, оскільки статичний метод не пов'язаний з конкретним об'єктом, в середині такого метода не можна звертатися до нестатичних полів класу без посилання на об'єкт перед ім'ям поля 135 (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 описувач може використовуватись як з полями, так і з методами136. Для полів – це простий засіб створення констант (в Java відсутня директива *#define*)137. Наприклад, *final int SIZE = 5;*

За замовчуванням константи записуються великими літерами138. Крім того, для економного використання пам'яті константи звичайно роблять статичними139. Що стосується *final*-методів, то це такі методи, які не можуть бути перевизначеними в класах-нащадках13A.

Передача параметрів в Java

Як відомо, існують два способи передачі параметрів: за значенням; за посиланням13B. Правило таке: якщо передається базовий тип (*int, char*), то результат передається за значенням13C. Якщо передається об'єкт (наприклад, клас), то він передається за посиланням13D.

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?

2.5 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**13E. Кожен пакет можна розглядати як підкаталог на диску. Пакети – це набори класів13F. Вони нагадують бібліотеки, які існують в багатьох мовах. Звичайно пакети містять класи, логічно пов'язані між собою.

Щоб внести клас в пакет, необхідно скористатися оператором *package* 140, наприклад:

```
package game;
```

При цьому необхідно витримати дві умови 141:

1. Вихідний текст класа 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 класа складається з імені пакета, крапки та імені класа142. Тому, якщо Ви внесли клас *Animator* в пакет *game*, щоб імпортувати його слід вказати повне ім'я класу 143:

```
import  game.Animator;
```

Якщо ж вам необхідно використати в своїй програмі декілька класів з пакета *game* і ви не бажаєте імпортувати кожен клас окремо, виписуючи всі їх імена, ви можете імпортувати весь пакет, отримавши доступ одразу до всіх класів даного пакета 144:

```
import  game.*;
```

І на сам кінець. Навіть нічого не імпортуючи, ви можете отримати доступ до деякого класу з пакету, вказуючи його повне ім'я при об'яві екземпляра класу 145:

```
game.Animator a;
```

Загальні відомості про інтерфейси

Інтерфейси – це варіант множинного успадкування, яка є в C++, але відсутня в Java146. Іншими словами, клас в Java не може успадкувати поведінку одразу кількох класів, але може реалізовувати одразу декілька інтерфейсів147. Також клас може бути одночасно і нащадком одного класу, і реалізовувати один або кілька інтерфейсів148.

В чому відмінність інтерфейсів від класів? Класи описують об'єкт, а інтерфейси визначають набір методів і констант, які реалізуються іншим об'єктом149. Інтерфейси мають одне головне обмеження: вони можуть описувати абстрактні методи та поля *final*, але не можуть мати жодної реалізації цих методів14A. В прикладному відношенні інтерфейси дозволяють програмісту визначити деякі функціональні характеристики, не турбуючись про те, як потім ці характеристики будуть описані14B.

Наприклад, якщо деякий клас реалізує інтерфейс *java.lang.Runnable* він має містити метод *run()* 14C. 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 інтерфейса

З точки зору синтаксису інтерфейси дуже схожі на класи14D. Головна відмінність полягає в тому, що жоден метод в інтерфейсі не має тіла та в ньому не можна об'являти змінні, а тільки константи14E.

В загальному випадку об'ява інтерфейса має вигляд14F:

[public] interface ИмяІнтерфейса [extends СписокІнтерфейсів]

Приклад інтерфейсу150:

```
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 151:

```
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 інтерфейса – об'являти абстрактні методи, які будуть реалізовані в інших класах152.

Зазначимо, що інтерфейси також можна розширяти, створюючи нащадків від вже існуючих інтерфейсів 153, наприклад:

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

Реалізація інтерфейсів

В класі, що реалізує інтерфейс, мають бути перевизначені всі методи, які були об'явлені в інтерфейсі, інакше клас буде абстрактним154. Звичайно, в класі можуть бути присутні інші, власні методи, не описані в інтерфейсах155. Приклад класу, що реалізує інтерфейс 156:

```
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*157.

Контрольні питання

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

3 ПАКЕТ JAVA.AWT: ІНТЕРФЕЙС КОРИСТУВАЧА ТА ОБРОБКА ПОВІДОМЛЕНЬ

3.1 Реалізація користувацького інтерфейсу

Розглянемо найбільший і, напевно, самий корисний розділ мови Java, зв'язаний з реалізацією користувацького інтерфейсу. Для цього необхідно вивчити базові класи пакету ***java.awt*** (Abstract Window Toolkit), представлені на рисунку 3.

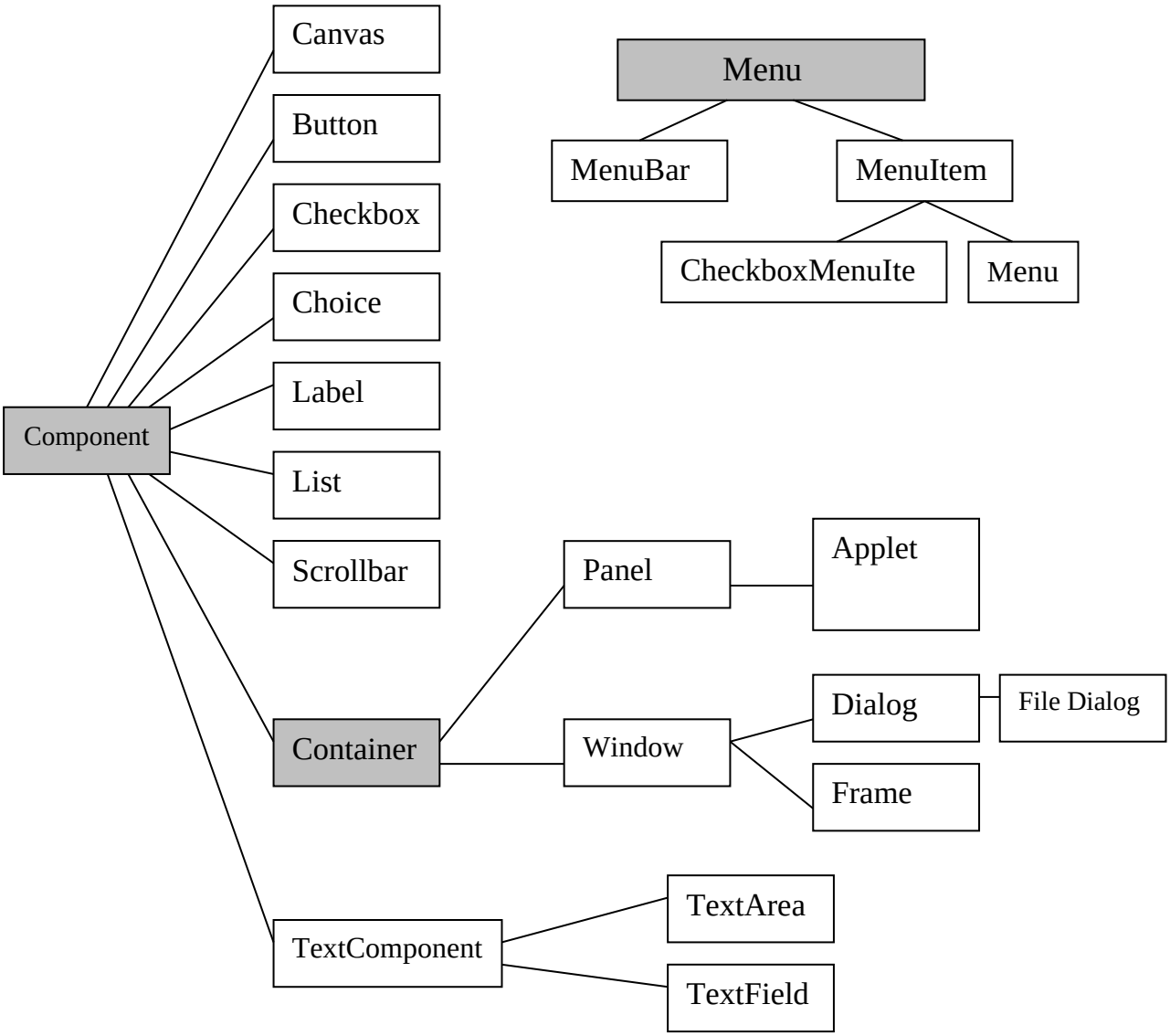

Рисунок 3 – Ієрархія класів пакету *java.awt*[158]

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)[159]. Практично всі класи візуальних

компонентів є нащадками абстрактного класу *Component*15A. Лише візуальні елементи меню успадковуються від іншого класу – *MenuComponent*15B. Керуючі елементи представлені такими класами: *Button* (кнопка), *Checkbox* (кнопка з незалежною фіксацією), *Choice* (список Windows), *Label* (рядок), *List* (список вибору Windows) і *Scrollbar* (смуга прокручування)15C. Це досить прості класи, успадковані від абстрактного класу *Component* безпосередньо15D.

Однак у складі *java.awt* є класи інтерфейсних елементів, що мають проміжного пращура15E. Прикладом тому є клас *Panel* для створення різних панелей15F. У нього є проміжний абстрактний клас-пращур *Container*, що слугує родоначальником багатьох класів-контейнерів, здатних містити в собі інші елементи інтерфейсу160. Від цього ж класу успадковується клас вікна *Window*, що представляє на екрані найпростіше вікно без меню і рамки161. У цього класу є два часто використовуваних нащадки: *Dialog*, назва якого говорить сама за себе, і *Frame* – стандартне вікно Windows162. Ще один проміжний клас *TextComponent* породжує два корисних у роботі класи – *TextField* (аналог рядка введення Windows) і багаторядкове вікно текстового введення *TextArea*163. Особняком від всіх елементів стоїть клас *Canvas*. Його візуальне представлення – порожній квадрат, на якому можна виконувати малювання і який може обробляти події натиснення кнопок миші164.

Від свого батьківського класу *Component* всі візуальні елементи переймають загальну для них усіх поведінку, пов'язану з їх візуальною та функціональною сторонами165. 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	166
setForeground() getForeground()	установлення і зчитування кольору компонента	167
setBackground() getBackground()	установлення і зчитування кольору тіла компонента	168
preferredSize() minimumSize()	повертають менеджеру розкладок інформацію про кращий і мінімальний розміри компонента, відповідно	169
resize() size()	встановлює і визначає розміри компонента	16A
show() hide()	показує та приховує компонент	16B
isVisible()	повертає true, якщо компонент відображений, і	16C

isShowing()	значення false, якщо компонент прихований	

Продовження таблиці 3

disable() enable()	забороняє або дозволяє компонент	16D
isEnable()	повертає true, якщо компонент дозволений	16E
paint() update() repaint()	відображення компонента	16F
handleEvent() action()	обробка повідомлень	170
keyDown() keyUp()	обробка повідомлень клавіатури	171
mouseDown() mouseUp() mouseDrag() mouseMove() mouseEnter() mouseExit()	обробка повідомлень миші	172

Використання компонентів користувацького інтерфейсу

Label (Текст)

За допомогою класу *Label* можна створювати текстові рядки у вікні Java-програм і аплетів[173]. Для створення об'єкту цього типу існують три види конструкторів[174]:

```
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 таблиці4.

Таблиця 4 – Основні методи класу *Label*[175]

SetText(String str)	змінює текст в рядку[176]
setAlignment(int align)	змінна вирівнювання[177]
String getText()	повертає текст рядка[178]
int getAlignment()	повертає значення вирівнювання[179]

Приклад.

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

Button (Кнопка)

Клас *Button* представляє на екрані кнопку17A. У цього класу є два типи конструктора17B. Перший з них створює кнопку без надпису, другий – з надписом 17C:

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

Корисні методи 17D:

setLabel() – створити або замінити надпис на кнопці;

getLabel() – дізнатись про надпис на кнопці.

Обробка кнопок. При натисненні на кнопку викликається метод *action()*17E:

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

де *evt* – подія, яка відбулася (*evt.target* – до якого об'єкту відноситься подія, *evt.when* – час виникнення події) 17F;

wA – текст надпису на кнопці 180.

Приклад 181.

```
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 фіксацією182. Це кнопки, що мають два стани: "увімкнене" і "вимкнене"183. Клік на такій кнопці приводить до того, що її стан міняється на

протилежний. В початковий момент прапорець скинутий (тобто має стан *false*).

Якщо розмістити кілька кнопок з незалежною фіксацією в середині елемента класу *CheckboxGroup*, то замість них ми одержимо кнопки з залежною фіксацією – перемикачі (radio button), тобто групу кнопок, серед яких у той самий момент може бути включено тільки одну. Якщо натиснути будь-яку кнопку з групи, то раніше натиснуту кнопка буде відпущено (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 наведено 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 по прапорцю або перемикачу 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 до допомоги класу *Choice*191. Створити його можна за допомогою конструктора 192

Choice();

Корисні методи класу *Choice* наведено в таблиці 6:

Таблиця 6 – Основні методи класу *Choice*193

addItem(String str)	Додати елемент в список194
select(int n)	Вибрати рядок з визначеним номером195
select(String str)	Вибрати визначений рядок тексту зі списку196
int countItems()	повернути кількість пунктів у списку197
int getSelectIndex()	повертає номер вибраного рядка (нумерація починається з 0) 198
String getItem(int n)	повернути рядок з визначеним номером у списку199
String getItem()	повернути вибраний рядок19A

Обробка списків, що розкриваються. При виборі елемента списку викликається метод *action()*, другий параметр *wA* якого містить в собі назву вибраного елемента19B.

Приклад 19C.

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, в середині якого знаходяться пункти вибору19D. Будь-який з цих пунктів можна вибрати подвійним щигликом або, вказавши на нього мишею, натиснути клавішу <Enter>19E. Причому можна зробити так, що стане можливим вибір декількох пунктів одночасно19F.

Корисні методи класу *List* наведено в таблиці 7.

1440 Таблиця 7 – Основні методи класу *List*1A0

addItem(String str)	додати рядок до списку;	1A1
addItem(String str, int index)	вставити рядок в список в позицію index (якщо index = -1, то додається в кінець списку);	1A2
replaceItem(String str, int index)	замінити елемент вибору в зазначеній позиції	1A3
delItem(int index)	видалити зі списку визначений пункт	1A4
delItems(int start, int end)	видалити елементи вибору з номерами, що входять в інтервал від номера start до номера end;	1A5
getItem(int n)	текст пункту вибору	1A6
clear()	очистити список (знищити всі елементи списку відразу)	1A7
select(int n)	виділити пункт із визначеним номером	1A8
deselect(int n)	зняти виділення з визначеного пункту	1A9
isSelected(int n)	повернути значення true, якщо пункт із зазначеним номером виділений, інакше повернути false	1AA
countItems()	порахувати кількість пунктів вибору в списку	1AB
getRows()	повернути кількість видимих у списку рядків вибору	1AC
getSelectedIndex()	довідатися порядковий номер виділеного пункту; якщо повертається -1, те обрано кілька пунктів (метод для списків I типу)	1AD
getSelectedItem()	прочитати текст виділеного пункту вибору (для списків I типу);	1AE
int[] getSelectedIndexes()	повернути масив індексів виділених пунктів (для списків II типу);	1AF
String[] getSelectedItems()	повернути масив рядків тексту виділених пунктів (для списків II типу)	1B0
allowsMultipleSelections()	повернути true, якщо список дозволяє множинний вибір	1B1
setMultipleSelections()	включити або виключити режим дозволу множинного вибору	1B2
makeVisible(int n)	зробити елемент із визначеним номером видимим у вікні списку	1B3
getVisibleIndex()	повернути індекс елемента вибору, що останнім після виклику методу makeVisible() став видимим у вікні списку	1B4

Створення об'єкта класу *List* може відбуватися двома способами1B5. Ви можете створити порожній список і додавати в нього пункти, викликаючи метод *addItem()*. При цьому розмір списку буде рости при додаванні пунктів1B6. Інший спосіб дозволяє відразу обмежити кількість видимих у вікні списку пунктів. Інші пункти вибору можна побачити, прокрутивши список.

Конструктори1B7:

```
List ();          // список для вибору тільки одного параметра1B8
List (int row, boolean mult);      // створення списку,  в якому
                  // row – кількість елементів, які можна одночасно
                  // бачити, mult – дозвіл вибіру більше одного ел-
              ma1B9
```

Обробка списків. Клас *List* не використовує метод *action()*1BA. Замість нього використовується метод 1BB

handleEvent(Event evt);

де *evt.id* – статична константа, що приймає одне з двох значень 1BC:

LIST_SELECT – елемент вибраний;
LIST_DESELECT – не вибраний;

evt.arg – змінна, яка містить індекс вибраного елемента.

Приклад 1BD.

```
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* представляє на екрані знайому усім смугу прокручування1BE. За допомогою цього елемента можна прокручувати зображення і текст у вікні або встановлювати деякі значення1BF. Щоб створити смугу прокручування, необхідно викликати конструктор об'єкта класу *Scrollbar*1C0. Це можна зробити трьома способами 1C1:

*Scrollbar() // смуга прокручування з параметрами за замовчуванням*1C2

Scrollbar(Scrollbar.VERTICAL); //смуга прокручування
*// з вертикальною орієнтацією*1C3

Scrollbar(<орієнт.>, <поч. зн.>, <видно>, <мін. зн.>, <макс. зн.>);
1C4

Де 1C5 <орієнт.> – орієнтація смуги, що задається константами:
Scrollbar.HORIZONTAL і *Scrollbar.VERTICAL*;
<поч.зн.> – початкове значення, в яке ставиться движок смуги прокручування;
<видно> – скільки пикселів прокручуваної області видно, і наскільки цю область буде прокручено при щиглику мишею на смузі прокручування;
<мін. зн.> – мінімальна координата смуги прокручування;
<макс. зн.> – максимальна координата смуги прокручування1C6.

Звичайно в якості прокручуваної області виступає об'єкт класу *Canvas* чи породжений від нього об'єкт1C7. При створенні такого класу його конструктору необхідно передати посилання на смуги прокручування1C8.

TextField і TextArea

Два родинних класи, *TextField* і *TextArea,* які успадковують властивості класу *TextComponent*, дозволяють відображати текст із можливістю його виділення і редагування1C9. По своїй суті це маленькі редактори: однорядковий (*TextField*) і багаторядковий (*TextArea*)1CA. 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*:

Конструктори1CB:

```
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 81CC.

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

Спільні методи для обох класів		
getText()	зчитати текст	1CD
setText(String)	відобразити текст;	1CE
select(int, int)	виділити текст між початковою і кінцевою позиціями;	1CF
selectAll()	виділити весь текст	1D0
getSelectedText()	прочитати виділений текст;	1D1
SetEditable(Boolean)	заборонити редагування тексту	1D2
isEditable()	перевірити, чи дозволене редагування тексту	1D3
getSelectionStart()	повернути позицію початку виділення	1D4
getSelectionEnd()	повернути позицію закінчення виділення	1D5
getColumns()	повернути кількість видимих символів у рядку редагування (не довжина рядка!)	1D6
Додаткові методи класу *TextField*		
setEchoChar(char)	встановити символ маски; застосовується при введенні паролів	1D7
getEchoChar()	довідатися символ маски	1D8
echoCharIsSet()	довідатися, чи встановлений символ маски	1D9
Додаткові методи класу *TextArea*		
int getRows()	зчитати кількість рядків у вікні	1DA
insertText(String, int)	вставити текст у визначеній позиції	1DB

appendText(String)	додати текст в кінці	1DC
replaceText(String, int, int)	замінити текст між заданими початковою і кінцевою позиціями	1DD

Обробка повідомлень текстового рядка та текстового поля. Як і клас *List*, клас *TextArea* не використовує метод *action()***1DE**. Оскільки події цього класу – це події клавіатури і миші, тому краще створити додаткову кнопку, яку користувач міг би натиснути, вказуючи, що введення завершено**1DF**. Після цього можна застосувати метод *getText()* для отримання результату введення і редагування**1E0**.

Для класу *TextField* також краще застосувати додаткову кнопку**1E1**. Хоча може бути використаний і метод *action()*, але тільки тоді, коли користувач натискає клавішу <Enter>**1E2**.

Розкладки

Для того щоб керувати розташуванням елементів всередині віконконтейнерів, у Java існує менеджер розкладок (layout manager)1E3. Від нього успадковуються п'ять класів, що визначають той чи інший тип розташування компонентів користувацького інтерфейсу у вікні1E4. 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 розкладку1E5:

setLayout(new BorderLayout());

FlowLayout (послідовне розташування)

Це найпростіший спосіб розташування елементів один за одним, застосовуваний за замовчуванням1E6. Коли в одному рядку вже не вміщуються нові елементи, заповнення продовжується з нового рядка1E7.

Конструктори1E8:

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

Параметр *align* може приймати одне з значень1E9:

FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.CENTER.

GridLayout (табличне розташування)

GridLayout розташовує елементи один за іншим усередині деякої умовної таблиці1EA. Всі елементи будуть однакового розміру1EB. Розмір комірок можна програмно змінювати1EC.

Конструктори1ED:

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

Якщо задано кількість рядків, то кількість стовпчиків буде розраховано, і навпаки1EE. Якщо треба створити розкладчик з заданим числом рядків, то кількість стовпців треба вказати 0 1EF. Якщо ж треба задати кількість стовпців, то замість кількості рядків слід задати 0 1F0. Таким чином, виклик *GridLayout(3, 4)* еквівалентний виклику *GridLayout(3, 0)* 1F1.

BorderLayout (полярне розташування)

Дана розкладка розділяє контейнер на 5 областей і розміщає елементи або поруч з обраним краєм вікна, або в центрі1F2. Для цього після установки *BorderLayout* додавання елементів у вікно-контейнер виконується методом *add()* з додатковим параметром, що задається рядками *“North”, “South”, “East”, “West” і “Center”*1F3. Таким чином, даний розкладчик розділяє контейнер на п’ять областей1F4.

Конструктори 1F5:

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

Даний розкладчик не дозволяє додавати в одну область більше одного компонента1F6. Якщо додано більше одного компонента, то буде видно лише останній1F7.

Метод *add()* має для даного розкладчика матиме такий вигляд:

add(int poz, Component comp);

де *poz* означає той край вікна, до якого необхідно пригорнути елемент, що вставляється *(“North”, “South”, “East”, “West” і “Center”)*1F8.

CardLayout (блокнотне розташування)

При розкладці цього типу елементи розміщаються один за іншим, як карти в колоді (в кожний момент часу видно тільки один елемент)1F9. Звичайно такий розклад зручний, якщо нам необхідно динамічно змінювати інтерфейс вікна1FA. Крім того, ми можемо робити елементи, що знаходяться один над іншим по черзі1FB. Створюються вкладки, вміст яких відображається при натисканні кнопки миші на заголовку1FC.

GridBagLayout (коміркове розташування)

Це сама мудрована, але в той же час і сама потужна розкладка1FD. Вона розташовує елемент в умовній таблиці, як це робиться у випадку з *GridLayout*1FE. Але на відміну від останньої, можна варіювати розмір кожного елемента окремо1FF. Правда, прийдеться набрати додатково не один рядок вихідного тексту200. Для кожного елемента задають власні "побажання"201. Ці побажання вміщуються в об'єкт *GridBagConstraints*, який містить такі змінні:

gridx, gridy – координати комірки, куди буде розміщений наступний компонент202. 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 обіймає компонент по горизонталі і вертикалі203. За замовчуванням – 1 204. 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*205;

206 *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;

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

208 *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**209**. Може приймати значення**20A**:

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**20B**.

Контейнери

Будь-який з компонентів, що вимагає показу на екрані, повинний бути доданий у клас-контейнер20C. Контейнери служать сховищем для візуальних компонентів інтерфейсу й інших контейнерів20D. В *awt* визначено такі контейнери20E:

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

Навіть якщо в аплеті явно не створюється контейнер, він все рівно буде використовуватися, оскільки клас *Applet* є похідним від класу *Panel*20F.

Найпростіший приклад контейнера – клас *Frame*, об'єкти якого відображаються на екрані як стандартні вікна з рамкою210.

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 екрані211. Незважаючи на настільки довгий список дій, у вихідному тексті цей процес займає усього кілька рядків212. Наприклад,

213 *Label text = new Label("Рядок"); // Створюється текстовий об'єкт*
// з надписом "Рядок"
214 *SomeContainer.add (text); // Об'єкт додається в деякий контейнер*
215 *SomeContainer.Show(); // Відображається контейнер*

Методи контейнерів наведено в таблиці 9.

Таблиця 9 – Основні методи класу *Container*

add()	додавання елемента інтерфейсу у вікно контейнера	216
add(Component, int)	передаються порядковий номер, куди буде вставлено елемент, і посилання на об'єкт	217
add(String,Component)	посилання на об'єкт, що вставляється. Рядків, припустимих як перший аргумент, всього п'ять: North, South, East, West і Center	218
getComponent(int)	повертає посилання на компонент (повертає тип Component) за заданим індексом	219
getComponents()	повертає масив Component[] всіх елементів даного контейнера	21A
countComponent()	повертає кількість компонентів у	21B

	контейнері	
remove()	видалення конкретного елемента	21C
removeAll().	метод видалення усіх візуальних компонентів	21D

Приклад.

```
add(someControl);        // Вставити елемент у вікно контейнера21E
add(-1, someControl);    // Вставити елемент після інших
                         // елементів у  контейнері21F
add("North", someControl);    // Вставити елемент у вікно
                              // контейнера у його верхньої
границі220
```

Панель

Клас *Panel* (панель) - це простий контейнер, у який можуть бути додані інші контейнери чи елементи інтерфейсу221. Звичайно він використовується в тих випадках, коли необхідно виконати складне розміщення елементів у вікні Java-програми й аплета222. При цьому панель може бути включена в склад інших контейнерів223.

Конструктор:

```
Panel();
```

Панель може містити в собі декілька інших панелей, тобто їх можна вкладати одна в одну224.

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 користувацького інтерфейсу можна вважати клас *Frame*. За його допомогою реалізуються вікна для Java-програм і аплетів225. На відміну від інших класів користувацького інтерфейсу, екземпляри класу *Frame* створюються рідко. Їх використовують для створення окремих додатків226. Звичайно від нього успадковується новий клас, а вже потім створюється екземпляр нового класу 227:

```
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	228
getTitle()	повернути заголовок вікна	229
setTitle(String)	встановити заголовок вікна	22A
getIconImage()	повернути піктограму вікна	22B
setIconImage(Image)	встановити піктограму вікна	22C
getMenuBar()	повернути об'єкт меню вікна	22D
setMenuBar(MenuBar)	встановити меню вікна	22E
remove(MenuComponent)	забрати визначений компонент із меню вікна	22F
isResizable()	повернути true, якщо розмір вікна можна змінювати, інакше – false	230
setResizable(boolean)	дозволити зміна розмірів вікна	231
getCursorType()	повернути поточний тип курсору миші для вікна	232
setCursor(int)	встановити тип курсору миші для вікна: Frame.DEFAULT_CURSOR, Frame.CROSSHAIR_CURSOR, Frame.TEXT_CURSOR, Frame.WAIT_CURSOR, Frame.HAND_CURSOR,	233

Dialog

Для підтримки зв'язку з користувачем застосовується клас *Dialog*, на основі якого можна створювати діалогові панелі234. 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,* що володіє цією діалоговою панеллю, тобто для створення діалогового вікна необхідно мати фрейм235. 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 236:

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

Конструктори цього класу 237:

```
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 модальність238. Якщо він дорівнює *true*, то діалогове вікно створюється модальним, у протилежному випадку воно дозволяє переключитися на інше вікно додатка239.

Крім загальних для усіх вікон методів *getTitle(), setTitle(), isResizable()* і *setResizable()* у класі *Dialog* є метод *isModal()*, що повертає *true*, якщо діалогова панель модальна23A.

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*23B. Перший з них, *MenuBar* - це основний клас усієї системи меню23C. Він слугує контейнером для інших класів. Коли ви створюєте вікно, то як посилання на меню потрібно передати посилання на клас *MenuBar*23D. Таким чином, для створення меню використовується конструктор23E:

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

Додати меню у фрейм можна так 23F:

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

Наступний клас *Menu* на смузі меню відображується як пункт вибору, що, якщо по ньому клацнути, розкривається у виді сторінки з пунктами вибору (pop-up menu)240. Самі ж елементи вибору меню звичайно реалізуються як екземпляри класів *MenuItem* (простий елемент вибору) і *CheckboxMenuItem* (відмічуваний елемент вибору)241.

Приклад створення повнофункціональної смуги меню 242:

```
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);
            // Створити і  додати 1-й пункт
        MenuItem item1_1 = new MenuItem("Item #1");
        menu1.add(item1_1);
        CheckboxMenuItem item1_2 = CheckboxMenuItem("Item #2");
            // відмічуваний пункт
        menu1.add(item1_2);
            // Створити і додати  друге меню
        Menu menu2 = new Menu("Menu 2");
        menuBar.add(menu2);
            // Створити і додати меню/ наступного рівня
        Menu nextLevel = New Menu("Next Level Menu");
        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 *nextLevel Menu* одержали чергове меню. На такий спосіб у Java реалізовано каскадне меню.

Обробка пунктів меню виконується за допомогою метода *action()*, де перший параметр – об’єкт класу *Event* 243, при цьому треба робити перевірку на приналежність до меню, а саме 244:

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

а другий параметр – назва вибраного пункту меню.

Контрольні питання

1. *Екземпляри яких класів пакету 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*245.

3.2 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 та непротирічні механізми для генерації та обробки подій246. Ця концепція доволі проста: джерело генерує подію та посилає її одному або кільком блокам прослуховування (listeners) подій247. За цією схемою блок прослуховування просто очікує на подію248. Отримавши подію, блок прослуховування обробляє її та повертає управління249. Перевага такого способу полягає в тому, що логіку компонента, який обробляє подію, чітко відокремлено від логіки інтерфейса користувача, що генерує ці події24A. Елемент інтерфейса користувача здатний “делегувати” обробку події окремій частині коду24B.

В моделі делегування подій блоки прослуховування мають зареєструватися у джерелі, щоб приймати повідомлення про події24C. Це забезпечує важливу перевагу: повідомлення відсилаються тільки тим блокам прослуховування, які хочуть його прийняти24D. 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 подію24E. Даний метод вимагає від компонентів прийняття подій, які вони не обробляють, на що витрачається певний час24F. Модель делегування подій усуває такі накладні витрати250.

Події

В моделі делегування подія – це об’єкт, який описує зміни стану джерела251. Він може бути згенерований як послідовність взаємодій оператора з елементами графічного інтерфейса користувача252. Генерацію подій можуть викликати такі дії оператора, як натиснення кнопки, введення символу з клавіатури, вибір елемента в списку, клік мишею та інші дії253. Крім того, програміст має змогу сам визначати події, які буде знаходити та обробляти його додаток254.

Джерела повідомлень

Джерело – це об’єкт, який генерує подію255. Генерація події відбувається тоді, коли змінюється внутрішній стан цього об’єкту256. Джерела можуть генерувати події кількох типів257.

Щоб блоки прослуховування мали змогу приймати повідомлення про певний тип подій, джерело має реєструвати ці блоки258. Кожен тип подій має власний метод реєстрації259. Загальна форма таких методів 25A:

public void addTypeListener(TypeListener el)

де *Type* – це ім’я події, *el* – посилання на блок прослуховування події25B. Наприклад, метод, який реєструє блок прослуховування події клавіатури, називається *addKeyListener()*25C. Метод, що реєструє блок прослуховування руху миші, називається *addMouseMotionListener()*25D. Коли подія відбувається, всі зареєстровані блоки прослуховування повідомляються про це та приймають копію об’єкта події25E. За будь-яких умов повідомлення відсилаються тільки тим блокам прослуховування, які зареєструвалися для їх приймання25F.

Деякі джерела можуть дозволяти реєструватися лише одному блоку прослуховування260. Загальна форма такого методу 261:

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

Коли подія відбувається, про це повідомляється тільки цей зареєстрований блок прослуховування262.

Джерело також має забезпечити метод, який дозволить блоку прослуховування не реєструвати зацікавленість у певному типі повідомлень263. Загальна форма такого методу 264:

public void removeTypeListener(TypeListener el)

Наприклад, щоб видалити блок прослуховування клавіатури, слід викликати метод *removeKeyListener()*265.

Методи, які додають або вилучають блоки прослуховування, забезпечуються джерелом, що генерує подію266. Наприклад, клас *Component* забезпечує методи для додавання та вилучення блоків прослуховування подій клавіатури та миші267.

Блок прослуховування подій

Блок прослуховування – це об’єкт, який отримує повідомлення, коли відбувається подія268. До нього висувається дві головних вимоги269. По-перше, щоб приймати повідомлення відносно певних типів подій, він має бути зареєстрованим одним або кількома джерелами цих подій26A. По-друге, він має реалізувати методи для приймання та обробки цих повідомлень26B.

Методи, що приймають і обробляють події, визначені в наборі інтерфейсів, що знаходяться в пакеті *java.awt.event*26C. Наприклад, інтерфейс *MouseMotionListener* визначає два методи для приймання повідомлень про події перетягування та пересування миші26D. Будь-який об'єкт може приймати та обробляти одну або обидві ці події, якщо він забезпечує реалізацію цього інтерфейса26E.

Класи подій

В основі механізму обробки подій знаходяться класи подій, які забезпечують непротирічиві та зручні для використання засоби інкапсуляції подій26F.

В корені ієрархії класів подій Java знаходиться клас *EventObject*, який розташовано в пакеті *java.util*270. Це – суперклас для всіх подій271. Один з його конструкторів 272:

EventObject(Object src)

де *src* – об'єкт, який генерує цю подію273.

Клас *EventObject* містить два методи: *getSource()* і *toString()*274. Метод *getSource()* повертає джерело події, а метод *toString()* повертає рядок – еквівалент події275.

Клас *AWTEvent*, визначений в пакеті *java.awt*, є підкласом класу *EventObject*276. Це суперклас (прямо або опосередковано) всіх AWT-подій, що використовуються моделлю делегування подій277. Для визначення типу події можна використовувати його метод *getID()*278. Його сигнатура 279:

int getID()

Пакет *java.awt.event* визначає декілька типів подій, які генеруються різноманітними елементами інтерфейса користувача27A. В таблиці 11 перераховуються найбільш важливі з цих класів подій та описується, коли вони генеруються27B. Відзначимо, що всі перераховані класи є нащадками класу *AWTEvent*27C.

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

27D	ActionEvent	генерується, коли натиснуто кнопку, відбувся подвійний клік на елементові списку або обрано пункт меню
27E	AdjustmentEvent	генерується при маніпуляціях із смугою прокручування
27F	ComponentEvent	генерується, коли компонент сховано, пересунуто, змінено в розмірі або зроблено видимим
280	ContainerEvent	генерується, коли компонент додається або вилучається з контейнера

281	FocusEvent	генерується, коли компонент отримує або втрачає фокус
282	InputEvent	абстрактний суперклас для всіх класів подій введення компонентів
283	ItemEvent	генерується, коли помічено прапорець або елемент списку, зроблено вибір елемента в списку вибору, обрано (відмінено) елемент меню з міткою
284	KeyEvent	генерується, коли отримано введення з клавіатури
285	MouseEvent	генерується, коли об'єкт перетягується (dragged) або пересувається (moved), відбувся щиглик (clicked), натиснуто (pressed) або відпущено (released) кнопку миші; також генерується коли покажчик миші входить або виходить в (поза) межі компонента
286	TextEvent	генерується, коли змінено значення текстової області або текстового поля
287	WindowEvent	генерується, коли вікно активізовано, закрито, дизактивовано, розгорнуто або згорнуто, відкрито або відбувся вихід (exit) з нього

Інтерфейси прослуховування подій

Модель делегування подій містить дві частини: джерела подій та блоки прослуховування подій288. Блоки прослуховування подій створюються шляхом реалізації одного або кількох інтерфейсів прослуховування подій, визначених в пакеті *java*289.*awt*28A.*event*28B. Коли подія відбувається, джерело події викликає відповідний метод, визначений блоком прослуховування, та передає йому об'єкт події як параметр28C. В таблиці 12 перераховано найбільш часто використовувані інтерфейси прослуховування та наведено короткий опис методів, визначених в цих блоках прослуховування28D.

Таблиця 12 – Інтерфейси прослуховування подій

28E	ActionListener	визначає один метод для приймання action-події
28F	AdjustmentListener	визначає один метод для приймання adjustment-події
290	ComponentListener	визначає чотири методи, що обробляють події, пов'язані з приховуванням, пересуванням, зміною та показом компонента
291	ContainerListener	визначає два методи, що обробляють події додавання або вилучення елемента з контейнера
292	FocusListener	визначає два методи, що обробляють події, пов'язані з

		отриманням або втратою компонентом фокуса клавіатури
293	ItemListener	визначає один метод, що обробляє подію зміни стану елемента
294	KeyListener	визначає три методи, що обробляють події натиснення, відпускання клавіші та введення символу
295	MouseListener	визначає п’ять методів для обробки подій входу в межі компонента, виходу за межі компонента, кліка, натиснення та відпускання кнопки миші
296	MouseMotionListener	визначає два методи, що обробляють події перетягування або пересування миші
297	TextListener	визначає один метод для обробки події зміни текстового значення
298	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*-подія299. Його загальна форма:

*void actionPerformed(ActionEvent evt)*29A.

Класи-адаптери

В Java існує спеціальний засіб, що носить назву класа адаптера (adapter class), яке за певних умов може спростити створення обробників повідомлень29B. Клас адаптера, або просто адаптер, забезпечує порожню реалізацію всіх методів в інтерфейсі прослуховування подій29C. Клас корисний, якщо ви хочете приймати та обробляти тільки частину подій, що надаються конкретним інтерфейсом прослуховування29D. 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 обробляти29E.

Іншими словами, якщо ви реалізуєте деякий інтерфейс, ви зобов’язані описати всі методи, визначені в цьому інтерфейсі29F. Якщо ви використовуєте замість інтерфейса адаптер, вам достатньо реалізувати тільки ті методи, які є дійсно необхідними2A0. Решту методів (у вигляді порожніх заготовок) вже містить сам адаптер2A1. В таблиці 13 представлено класи-адаптери, що їх визначено в пакеті *java.awt.event* з відповідними інтерфейсами, які кожен з них реалізує2A2.

Таблиця 13 – Інтерфейси прослуховування подій

	Клас-адаптер	Інтерфейс прослуховування подій
2A3	ComponentAdapter	ComponentListener
2A4	ContainerAdapter	ContainerListener

2A5	FocusAdapter	FocusListener
2A6	KeyAdapter	KeyListener
2A7	MouseAdapter	MouseListener
2A8	MouseMotionAdapter	MouseMotionListener
2A9	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?*

Приклад 2AA

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>.*

4 ПОТОКИ. АПЛЕТИ. РАСТРОВІ ЗОБРАЖЕННЯ

4.1 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 оброблювати повідомлення WM_TIMER.

Якщо повернутися ще назад, то, при створенні програм під DOS, коли треба було, щоб програма одночасно щось виводила на екран і аналізувала клавіші, ви 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 з заданим інтервалом часу буде, наприклад, перемальовувати зображення або зчитувати інформацію з буфера при обміні з іншими комп'ютерами.

Створення потоків

Для реалізації багатопоточності ми маємо скористатися класом *java.lang.Thread* 2AB. В ньому визначено всі методи для створення потоків, управління їх станом та синхронізації2AC.

Є дві можливості для того, щоб дати можливість вашим класам працювати в різних потоках.

По-перше, можна створити свій клас-нащадок від суперкласа *Thread*2AD. При цьому ви отримуєте безпосередній доступ до всіх методів потоків:

*public class MyClass extends Thread*2AE.

По-друге, ваш клас може реалізувати інтерфейс *Runnable*2AF. Це – ліпший варіант, якщо ви бажаєте розширити властивості якогось іншого класу, наприклад, *Frame*, як ми це робили раніше, або *Applet*2B0. Оскільки в Java немає множинної спадковості, реалізація інтерфейса *Runnable* – єдина можливість рішення цієї проблеми2B1.

*public class MyClass extends Frame implements Runnable*2B2.

І в тому, і в іншому випадку вам доведеться реалізовувати метод *run()*2B3.

Є сім конструкторів в класі *Thread*2B4. 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 конструктору передається значення посилання на цей клас (*this*)2B5.

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!)2B6. 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 виду 2B7:

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

При цьому можна вважати, що код додатка та код метода *run()* працюють одночасно як різні потоки2B8. Коли цикл закінчується та метод *run()* повертає управління, потік завершує роботу нормальним чином2B9.

А що знаходиться в середині циклу *while*? Як правило, він містить виклик методу *repaint()* для перерисовки, а також виклик метода *sleep()* класу *Thread*, який ми зараз розглянемо2BA.

Керування потоками

Після того як потік створений, над ним можна виконувати різні керуючі операції: запуск, зупинку, тимчасову зупинку і т.д**2BB**. Для цього необхідно використовувати методи, визначені в класі *Thread***2BC**.

Запуск потоку. 2BD 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 класі2BE.

Якщо метод *run()* повертає керування, запущений потік завершує свою роботу2BF. Однак, звичайно метод *run()* запускає нескінченний цикл, тому потік не завершить своє виконання доти, поки він не буде зупинений (чи завершений) примусово2C0. Щоб визначити, запущений даний чи потік ні, можна скористатися таким методом2C1:

public final boolean isAlive();

Зупинка потоку. Якщо додаток бажає зупинити потік нормальним неаварійним чином, то він викликає для відповідного об'єкта метод *stop()*2C2:

public final void stop();

Тимчасова зупинка і поновлення роботи потоку. За допомогою метода *sleep()* ви можете затримати виконання потоку на заданий час (в мілісекундах) 2C3:

public static void sleep(long ms);

При цьому управління передається іншим потокам. Роботу ж нашого потоку буде поновлено через заданий інтервал часу2C4. Метод *suspend()* тимчасово припиняє роботу потоку2C5:

public final void suspend();

Для продовження роботи потоку необхідно викликати метод *resume()*2C6:

public final void resume();

Пріоритети потоків. Клас *Thread* містить методи що дозволяють управляти пріоритетами потоків2C7. Є також засоби для синхронізації та блокування потоків2C8.

Таким чином, якщо необхідно запустити **один** потік для анімації, слід виконати такі дії2C9:

1) В об’яві класу вказати, що він реалізує інтерфейс *Runnable*2CA.
2) Описати змінну класу *Thread* як поле класу2CB.
3) Описати метод *start()* в середині вашого класу. Ця вимога не є обов’язковою, але якщо ви потім захочете перетворити ваш додаток на

аплет, ліпше зробити саме так2CC. В методі *start()* створити об’єкт класу *Thread* (за допомогою *new*) і викликати метод *start()* для цього об’єкта2CD:

myThread.start()

Сам метод *start()*

MyClass.start()

для додатка можна викликати з методу *main()*2CE. Для аплета він викликається автоматично2CF.

4) Описати метод run() в середині вашого класу2D0. В середині метода run() створити “нескінчений” цикл2D1. В циклі має знаходитися перерисовка зображення (метод repaint()) і призупинка роботи потоку на деякий час (метод sleep()), щоб інші потоки також могли виконати свої функції2D2.
5) Завершити роботу потоку можна за допомогою метода stop()2D3.

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*2D4. Кожен такий клас відповідає одному потоку і має свій власний метод *run()*2D5. У класі аплета нам потрібно визначити необхідну кількість об'єктів класу, що реалізує потік, при цьому інтерфейс *Runnable* в самому аплеті реалізовувати не потрібно2D6.

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?*

Приклад 2D7

4.2 Виняткові ситуації та їх обробка

Виняток або виключення (exception) – це спеціальний тип помилки, який виникає у випадку неправильної роботи програми2D8. Виняткова (виключна) ситуація може виникнути при роботі Java-програми в результаті, наприклад, ділення на нуль або може бути ініційована програмно в середині метода деякого класу2D9. Прикладом такого винятку, що генерується програмно, може служити *FileNotFoundException*, який викидається (*throw*) методами класів введення-виведення при спробі відкрити неіснуючий файл2DA. Замість терміну "викидається" часто вживають синоніми: збуджується, генерується, ініціюється2DB.

Після того як Java-машина створить об'єкт-виняток, цей об'єкт пересилається додатку2DC. Об'єкт, що утворюється при збудженні винятку, несе інформацію про виняткову ситуацію (точка виникнення, опис тощо)2DD. Використовуючи методи цього об'єкта можна, наприклад, вивести на екран або в файл інформацію про цей виняток2DE.

Додаток має перехопити виняток. Для перехоплення використовується так званий *try-catch* блок2DF. Наприклад, при спробі читання даних з потоку стандартного пристрою введення-виведення може виникнути виняток *IOException*2E0:

```
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 виняток2E1. Якщо все відбувається нормально, весь код в середині блока *try* виконується, а блок *catch* пропускається2E2.

Зверніть увагу. Блок *catch* нагадує метод, адже йому передається як параметр об'єкт-виняток2E3. Тип параметра – *IOException*, ім'я параметра – *e*2E4. 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 виняток2E5. 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 консоль2E6. 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 консоль2E7. 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 роботу2E8. Якщо в методі, з якого викликано даний метод, також немає блока перехоплення, то і він припиняє свою роботу2E9. І т. д., поки не буде знайдено блок перехоплення або не закінчиться ланцюжок викликаних методів2EA.

Звідси **висновок:** обробляти виняток необов'язково в тому ж самому методі, в якому він генерується2EB.

Приклад 2EC

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

При написанні власних методів ви також маєте враховувати, що вони можуть збуджувати винятки. В цьому випадку метод має виглядати так2ED:

```
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()*2EE:

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

Можна створювати власні класи винятків, але це зовсім інша тема [2].

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*2EF. 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?*

4.3 Аплети

Аплети – це невеличкі програми, що працюють в середині браузера2F0.

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 Internet. Відповідно великий аплет потребує багато часу на завантаження.

Проблема безпеки

Суть проблеми в тому, що аплет – це програма, яку користувач отримує з зовнішнього джерела. Відповідно вона є потенційно небезпечною2F1. Тому аплети сильно обмежені в своїх правах2F2. Наприклад, аплети не можуть читати локальні файли (тобто файли на клієнтській машині), а тим більше в них писати2F3. Є також обмеження на передачу даних через мережу: аплет може обмінюватись даними тільки з тим сервером Web, з якого його завантажено2F4.

Створення аплетів

Для побудови аплета треба створити клас-нащадок класа *Applet*, який входить до складу пакета *java.applet*, та перевизначити в ньому низку методів класа *Applet*2F5. 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 втрачаючи2F6. Тож розглянемо їх.

public void init()

Викликається браузером **один раз** одразу після завантаження аплета перед першим викликом метода *start()*2F7. Цей метод треба перевизначати практично завжди, якщо в аплеті потрібна будь-яка ініціалізація2F8. Мабуть, ми не помилимося, якщо весь код, який знаходився у програмі-додатку в конструкторі перенесемо в метод *init()* аплета2F9.

Як і конструктор, *init()* має свою контрпару2FA:

public void destroy()

Викликається браузером один раз перед вивантаженням даної сторінки2FB. Якщо аплет використовував ресурси, які перед знищенням аплета треба звільнити, це необхідно зробити, перевизначивши цей метод2FC.

public void start()

Викликається браузером при кожному "відвідуванні" даної сторінки2FD. 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)2FE. 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 даної2FF. В останньому випадку *stop()* викликається перед *destroy()*300.

Методи *start()* і *stop()* використовуються в парі для заощадження ресурсів, наприклад, при створенні анімації301. Тоді *stop()* може її зупинити, а *start()* запустити знову302.

Зверніть увагу. Жоден з цих методів не викликається програмістом напряму, всі вони викликаються браузером303.

Перетворення додатка на аплет (на прикладі програми NervousText.java) можна такими діями304:

1) Додати рядок

import java.applet.Applet;

2) В заголовку класу замінити extends Frame на extends Applet.
3) Конструктор public NervousText() замінити на public void init().
4) Знищити (закоментувати) виклик конструктора суперкласа *super("Nervo")*.
5) Знищити (закоментувати) метод *main()* повністю.
6) Якщо його не було визначено, створити метод *stop()*, в якому зупинити 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 в 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

Internet Explorer. Але попередньо необхідно створити клас аплета NervousText.class, як і завжди, за допомогою компілятора javac.exe305.

Контрольні питання

1. *Чим відрізняється використання методів init() і start()?*
2. *По аналогії 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?*

4.4 Створення растрових зображень

При створенні додатка, наприклад, на C++ можна обирати 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 JPEG306.

Завантаження растрового зображення

Виконується за допомогою метода *getImage()*307. Існує декілька варіантів цього метода. Насамперед, варіант цього метода визначено в класі *Applet* 308:

public Image getImage(URL url, String name);

Клас *URL* надає URL (Uniform Resource Locator, уніфікований покажчик ресурсів), який є форматом адрес ресурсів в WWW309. Другий параметр задає розташування файла зображення відносно адреси URL30A. Наприклад30B,

```
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 так30C:

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

Метод *getCodeBase()*, який також належить класу *Applet*, повертає URL-адресу аплета30D. Замість нього можна використовувати метод *getDocumentBase()*, який повертає URL-адресу HTML-файла, що містить аплет30E.

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

Якщо ви створюєте не аплет, а додаток, ліпше використовувати інший варіант *getImage()*, який визначено в класі *Toolkit* 30F

public abstract Image getImage(String filename)

Як звернутися до цього метода (зверніть увагу, що він має один параметр)? Наведемо приклад використання *getImage()* для завантаження файла *duke1.gif*, що знаходиться в підкаталозі *images* поточного каталога310:

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

За будь-яких умов метод *getImage()* повертає об’єкт класу *Image*311.

Виведення зображення

Насправді метод *getImage()* **не завантажує** зображення через мережу, як це може здаватися312. Він тільки створює об’єкт *Image*313. Реальне завантаження файла растрового зображення буде виконуватися методом рисування *drawImage()*, який належить класу *Graphics*314. Варіанти цього методу (не всі) 315:

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()*316. Далі *x* та *y* – координати лівого верхнього кута прямокутного регіону, в якому буде виводитись зображення317. Якщо для рисування обрано метод *drawImage()* з параметрами *width* (ширина) та *height* (висота), зображення буде виведено з масштабуванням318. Помноживши ці параметри на коефіцієнти, можна розтягнути (стиснути) зображення по горизонталі та вертикалі319. Параметр *observer* – це посилання на об’єкт класу *ImageObserver*, який отримає звістку при завантаженні зображення31A. Звичайно таким об’єктом є сам клас, тому цей параметр вказується як *this*.

Коли викликається метод *drawImage()* зображення ще може бути не завантажено31B. Оскільки процес завантаження по мережі – досить тривалий та не передбачуваний 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 по мірі готовності, можна дочекатися повного завантаження, а вже потім виводити на екран31C.

Клас Image

Розглянемо детальніше методи класу Image.

Методи *getHeight()* та *getWidth()*, визначені в класі *Image*, дозволяють визначити відповідно висоту та ширину зображення31D:

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*31E.

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 екрані31F.

Техніка анімації

Є три можливості оживити Web-сторінку320:

- створення AVI-файл;
- створення багатосекційного GIF-файла;
- використання кількох файлів в форматі GIF або JPEG як окремих кадрів відеофільма321.

Звичайно, нас цікавить саме остання можливість. Отже, ідея проста. Завантажуємо за допомогою *getImage()*, декілька файлів в масив, елементи якого мають тип *Image*322. Потім, користуючись методами управління потоками, створюємо зміну кадрів323. При цьому в методі *paint()* необхідно передбачити зміну індексу в масиві, щоб на екран виводився кожен раз новий кадр324.

Усунення мерехтіння

Головним чинником цього неприємного явища є те, що зображення малюється безпосередньо перед очима користувача325. Ця перерисовка помітна оку та викликає ефект мерехтіння326. Стандартний вихід з цієї ситуації – подвійна буферизація327.

Основна ідея полягає в тому, що поза екраном (в оперативній пам'яті) створюється зображення, і все рисування відбувається саме на цьому зображенні**328**. Коли рисування завершується, можна скопіювати зображення на екран за допомогою лише одного метода, таким чином поновлення екрану відбудеться миттєво**329**.

Інше джерело мерехтіння – метод *update()*, який викликається методом *repaint()***32A**. Стандартний метод *update()* спочатку очищує область рисування, а потім викликає метод *paint()***32B**. Щоб позбавитися від цього, достатньо просто перевизначити метод *update()*, щоб він просто викликав метод *paint()***32C**:

```
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 малюнку, якщо нове зображення його повністю не покрило32D.

Подвійна буферизація, хоча й більш кропітка, дозволяє повністю позбавитись від мерехтіння32E. Спочатку треба визначити поле нашого класу типу *Image*, яке буде служити позаекранним зображенням32F:

```
private Image offScreenImage;
```

Далі в конструкторі класу додати ініціалізацію (створення) цього поля330:

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

І, на сам кінець, треба перевизначити метод *update()*, щоб він не очищував екран, а дозволяв методу *paint()* сформувати зображення, яке потім копіюється на екран331:

```
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?*

4.5 Можливості Java 2D

У систему пакетів і класів Java 2D, основа якої – клас *Graphics2D* пакету *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)*: екрану монітора, принтера332. Методи класу *Graphics2D* автоматично переводять систему координат користувача в систему координат пристрою при виведення графіки333.

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 стиснення334. Воно визначається як об'єкт класу *AffineTransform*335. Його можна встановити як перетворення за замовчуванням методом *setTransform()*336. Можливо виконувати перетворення "на льоту" методами *transform()* і *translate()* і робити композицію перетворень методом *concatenate()*337. Причому для аффінного перетворення координати задаються дійсними, а не цілими числами338.

3. Графічні примітиви (прямокутник, овал, дуга і ін.) реалізують тепер новий інтерфейс *Shape* пакету *java.awt*339. Для їх викреслювання можна використовувати новий єдиний для всіх фігур метод *draw()*, аргументом якого здатний служити будь-який об'єкт, що реалізував інтерфейс *Shape*33A. Введений метод *fill()*, що заповнює фігури – об'єкти класу, який реалізує інтерфейс *Shape*33B.

4. Для викреслювання ліній введене поняття пера (pen)33C. Властивості пера описує інтерфейс Stroke33D. Клас BasicStroke реалізує цей інтерфейс33E. Перо володіє чотирма характеристиками: воно має товщину; може закінчити лінію якимось способом; сполучати лінії різними способами та креслити лінію різними пунктирами і штрих-пунктирами33F.

5. Методи заповнення фігур описані в інтерфейсі Paint340. Три класи реалізують цей інтерфейс341. Клас Color реалізує його суцільною (solid) заливкою, клас GradientPaint – градієнтним (gradient) заповненням, при якому колір плавно міняється від однієї заданої точки до іншої, клас TexturePaint – заповненням за заздалегідь заданим зразком (pattern fill)342.

6. Літери тексту розуміються як фігури, тобто об'єкти, що реалізовують інтерфейс Shape, і можуть викреслюватися методом draw() з використанням всіх можливостей цього методу343. При їх викреслюванні застосовується перо, всі методи заповнення і перетворення344.

7. Окрім імені, стилю і розміру, шрифт одержав багато додаткових атрибутів, наприклад, перетворення координат, підкреслення або

перекреслювання тексту, виведення тексту справа наліво345. Колір тексту і його фону є тепер атрибутами самого тексту, а не графічного контексту346. Можна задати різну ширину символів шрифту, надрядкові і підрядкові індекси347. Атрибути встановлюються константами класу TextAttribute348.

8. Процес візуалізації (rendering) регулюється правилами (hints), визначеними константами класу RenderingHints349.

З такими можливостями Java 2D стала повноцінною системою малювання, виведення тексту і зображень. Подивимося, 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 шрифт34A. Надалі його можна змінити методом *setTransform()* так само, як змінюється колір або шрифт34B. Аргументом цього методу служить об'єкт класу *AffineTransform* з пакету *java.awt.geom*34C.

Перетворення координат задається двома основними конструкторами34D:

AffineTransform{double a, double b, double c, double d, double e, double f);

*AffineTransform(float a, float b, float c, float d, float e, float f)*34E.

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 пристрою34F.

Таке перетворення не скривлює площину – прямі лінії переходять в прямі, кути між лініями зберігаються350. Прикладами таких перетворень служать повороти навколо будь-якої точки на будь-який кут, паралельні зрушення, віддзеркалення від осей, стиснення і розтягування по осях351.

Наступні конструктори використовують як аргумент масив *{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)*352.

П'ятий конструктор створює новий об'єкт по готовому об'єкту:

*AffineTransform (AffineTransform at)*353.

Шостий конструктор (за замовчуванням) створює тотожне перетворення:

*AffineTransform ()*354.

Всі ці конструктори математично точні, але не завжди зручні при конкретних перетвореннях355. Тому у багатьох випадках зручніше створити перетворення статичними методами356:

357*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 мінялися перетворенням віддзеркалення, то додатне значення angle задає поворот за годинниковою стрілкою;

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

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

35A*getShareInstance (double shx, double shy)* – перетворить кожну точку *(x,y)* у точку *(x+shx*y, shy*x+y)*;

35B*getTranslateInstance (double tx, double ty)* – зсовує кожну точку (x,y) у точку *(x+tx, y+ty)*;

35C*createInverse()* – повертає перетворення, зворотне поточному35D.

Після створення перетворень його можна змінити методами35E:

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 перетворень35F:

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 поточного перетворення360.

Інші методи класу *AffineTfansform* здійснюють перетворення різних фігур в просторі користувача361.

Приклад перетворення системи координат наведено на рисунку 6.

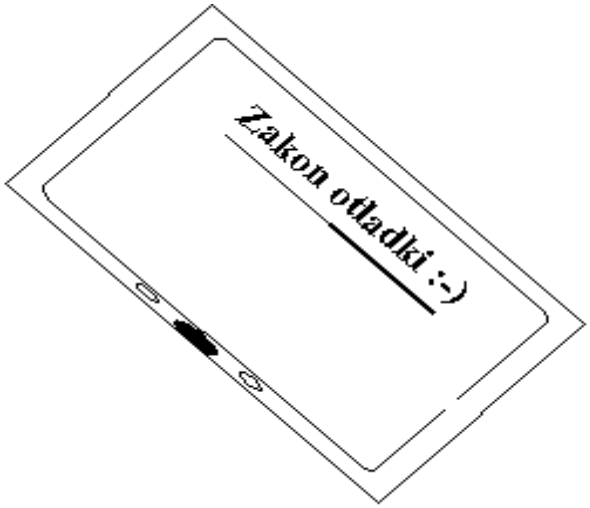

Рисунок 6 – Приклад змінення системи координат засобами Java2D

Малювання фігур засобами Java2D. Клас *BasicStroke*

Характеристики пера для малювання фігур описані в інтерфейсі *Stroke*362. У Java2D є поки тільки один клас, що реалізовує цей інтерфейс – клас *BasicStroke*, основний конструктор якого такий 363:

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

де *width* – товщина пера в пікселах;
cap – оформлення кінця лінії; це одна з констант364:
CAP_ROUND – закруглений кінець лінії;
CAP_SQUARE – квадратний кінець лінії;
CAP_BUTT – оформлення відсутнє;
join – спосіб сполучення ліній; це одна з констант365:
JOIN_ROUND – лінії сполучаються дугою кола;
JOIN_BEVEL – лінії сполучаються відрізком прямої, перпендикуляр-ним бісектрисі кута між лініями;
JOIN_MITER – лінії просто стикуються;
miter – відстань між лініями, починаючи з якого застосовується сполучення JOIN_MITER366;
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 циклічно367;
dashBegin – індекс, починаючи з якого перебираються елементи масиву dash368.
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 сполученням JOIN_ ROUND або JOIN_BEVEL; для сполучення JOIN_MITER задається значення miter=10.0f;

BasicStroke (float width) – прямий обріз CAP_SQUARE і сполучення JOIN_MITER із значенням miter=10.0f,

BasicStroke ( ) – ширина 1.0f.

Після створення пера одним з конструкторів і установки пера методом *setStroke()* можна малювати різні фігури методами *draw()* і *fill()*369. Загальні властивості фігур, які можна намалювати методом *draw()* класу *Graphics2D*, описані в інтерфейсі *Shape*36A. Цей інтерфейс реалізований для створення звичного набору фігур – прямокутників, прямих, еліпсів, дуг, точок – класами *Rectangle2D, RoundRectangle2D, Line2D, Ellipse2D, Arc2D, Point2D* пакету *java.awt.geom* (рис.7)36B.

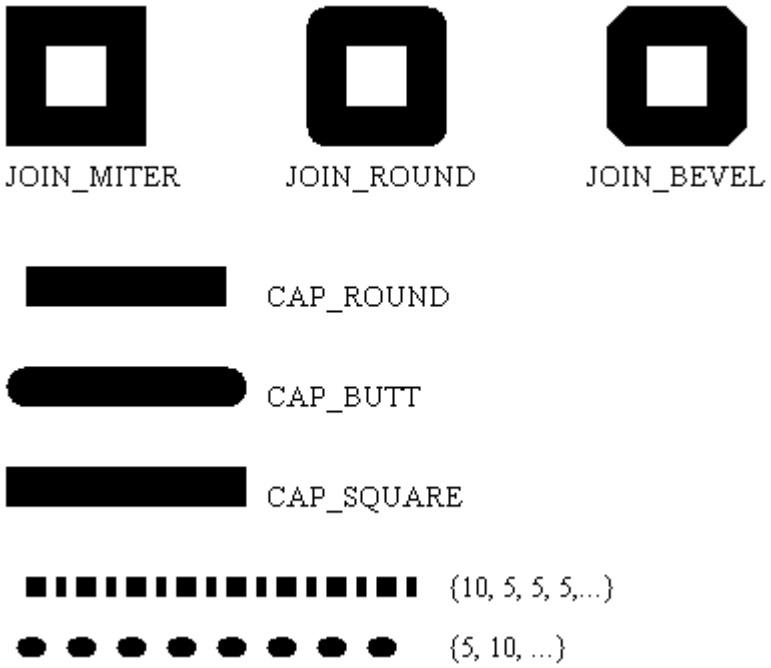

Рисунок 7 – Приклад виведення фігур засобами Java2D

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, сполучених або не сполучених між собою36C. Більш того, оскільки цей клас реалізує інтерфейс *Shape*, його екземпляри самі є фігурами і можуть бути елементами інших об'єктів класу *GeneralPath*36D.

Клас GeneralPath

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

GeneraiPath()

або об'єкт, що містить одну фігуру, конструктором

*GeneraiPath (Shape sh)*36E.

Потім до цього об'єкту додаються фігури методом

*append (Shape sh, boolean connect)*36F.

Якщо параметр *connect* дорівнює *true*, то нова фігура з'єднується з попередніми фігурами за допомогою поточного пера370.

У об'єкті є поточна точка, спочатку її координати (0,0)371. Далі можна застосувати методи372:

moveTo(float x, float y) – переміститися в точку (x, y);

lineTo(float x, float, y) – провести від поточної точки до точки (x,y) відрізок прямої;

quadTo(float x1, float y1, float x, float, y) – провести відрізок квадратичної кривої;

curveTo(float x1, float y1, float x2, float y2, float x1, float y1) – провести криву Без'є.

Поточною після цього стає точка *(x,y)*. Початкову і кінцеву точки можна з'єднати методом *closePath()*373. Наприклад, так можна створити трикутник із заданими вершинами374:

```
GeneralPath p = new GeneralPath();
p.moveTo(x1,y1);  // переносимо поточну точку в першу вершину
p.lineTo(x2,y2);  // проводимо сторону трикутника до другої вершини
p.lineTo(x3,y3);  // проводимо другу сторону
p.closePath();    // проводимо третю сторону до першої вершини
```

Способи заповнення фігур визначені в інтерфейсі *Paint*375. В даний час *Java2D* містить три реалізації цього інтефейса – класи *Color*, *GradientPaint* і *TexturePaint*376.

Класи GradientPaint і TexturePaint

Клас *GradientPaint* пропонує зробити 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 міняється від *c1* до *c2* уздовж прямої, що сполучає точки *M* і *N*, залишаючись постійним уздовж кожної прямої, перпендикулярної прямої *MN*. Таку заливку створює конструктор 377

*GradientPaint(float xl, float yl, Color cl, float x2, float y2, Color c2)*378.

При цьому поза відрізком *MN* колір залишається постійним: за точкою *M* – колір *c1*, за точкою *N* – колір *c2*.

Другий конструктор 379:

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

При *cyclic=true* повторюється заливка смуги *MN* у всій фігурі37A.

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

Клас *TexturePaint* поступає складніше. Спочатку створюється буфер – об'єкт класу *BufferedImage* з пакету 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*37B.

Графічний контекст буфера заповнюється фігурою, яка служитиме зразком заповнення37C.

Потім за буфером створюється об'єкт класу *TexturePaint*37D. При цьому ще задається прямокутник, розміри якого будуть розмірами зразка заповнення. Конструктор виглядає так37E:

TexturePaint(Bufferedimage buffer, Rectangle2D anchor).

Після створення заливки – об'єкту класу *Color, GradientPaint* або *TexturePaint* – вона встановлюється в графічному контексті методом 37F

setPaint (Paint p)

і використовується надалі методом *fill (Shape sh)*380.

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, аффінне перетворення, розташування зліва направо або справа наліво381.

Атрибути шрифту задаються як статичні константи класу *TextAttribute*382. Найбільш використовувані атрибути перераховані в таблиці 14.

На жаль, не всі шрифти дозволяють задати всі атрибути. Подивитися список допустимих атрибутів для даного шрифта можна методом *getAvailableAttHbutes()* класу *Font*383.

Таблиця 14 – Атрибути шрифтів

	Атрибут	Значення
384	BACKGROUND	Колір фону. Об'єкт, що реалізовує інтерфейс Paint
385	FOREGROUND	Колір тексту. Об'єкт, що реалізовує інтерфейс Paint
386	BIDI_EMBEDDED	Рівень вкладеності проглядання тексту, ціле від 1 до 15
387	CHAR_ REPLACEMENT	Фігура, замінююча символ. Об'єкт GraiphicAttribute
388	FAMILY	Сімейство шрифта. Рядок типа String
389	FONT	Шрифт. Об'єкт класу Font

38A	JUSTIFICATION	Допуск при вирівнюванні абзаца. Об'єкт класу Float (від 0,0 до 1,0). Є дві константи: JUSTIFICATION__FULL і JUSTIFICATION_NONE
38B	POSTURE	Нахил шрифта. Об'єкт класу Float. Є дві константи: POSTUREJDBLIQUE і POSTURE_REGULAR
38C	RUN_DIRECTION	Проглядання тексту: run_direction_ltr – зліва направо, run_DIRECTION_rtl – справа наліво
38D	SIZE	Розмір шрифту в пунктах. Об'єкт класу Float
38E	STRIKETHROUGH	Перекреслювання шрифта. Константа STRIKE-THROUGH_ON (за замовчуванням перекреслювання немає)
38F	SUPERSCRIPT	Підрядкові або надрядкові індекси. Константи: SUPERSCRIPT_NO, SUPERSCRIPT_SUB, SUPERSCRT_SUPER
390	SWAP_COLORS	Заміна місцями кольору текста і кольору фона. Константа Swap_colorsjdn, за замовчуванням заміни немає
391	TRANSFORM	Перетворення шрифта. Об'єкт класу AffineTransform
392	UNDERLINE	Підкреслення шрифта. Константи: underline_on, UNDERLINE_LOW_DASHED, UNDERLINE_LOW_DOTTED, UNDERLINE_LOW_GRAY, UNDERLINE LOW_ONE_PIXEL, UNDERLINE LOW TWO PIXEL
393	WEIGHT	Товщина шрифта. Константи: WEIGHT_ULTRA_LIGHT, WEIGHT_EXTRA_LIGHT, WEIGHT_LIGHT і ін.
394	WIDTH	Ширина шрифта. Константи: WIDTH_CONDENSED, WIDTH_SEMI_CONDENSED, WIDTH_REGULAR, WIDTH_SEMI_EXTENDED, WIDTH_EXTENDED

У класі Font є конструктор *Font(Map attrib)*, яким можна відразу задати потрібні атрибути створюваному шрифту395. Це вимагає попереднього запису атрибутів в спеціально створений для цієї мети об'єкт класу, що реалізовує інтерфейс *Map*: класу *HashMap*, *WeakHashMap* або *Hashtable*396. Наприклад 397,

```
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*398.

Текст в Java 2D має власний контекст – об'єкт класу *FontRenderContext*, що зберігає всю інформацію, необхідну для виведення тексту399. Одержати його можна методом *getFontRendexContext()* класу *Graphics2D*39A.

Вся інформація про текст, у тому числі і про його контекст, збирається в об'єкті класу *TextLayout*39B. Цей клас в Java 2D замінює клас *FontMetrics*39C.

У конструкторі класу *TextLayout* задається текст, шрифт і контекст39D. Початок методу *paint()* зі всіма цими визначеннями може виглядати так:

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

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

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

що викреслює вміст об'єкту класу *TextLayout* в графічний області *g*, починаючи з точки *(x,y)*39E.

Ще один метод

getOutline(AffineTransform at)

повертає контур шрифту у вигляді об'єкту *Shape*39F. 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 пакету *java.text*. Конструктор цього класу

AttributedString(String text, Map attributes)

задає відразу і текст, і його атрибути3A0. Потім можна додати або змінити характеристики тексту одним їх трьох методів *addAttibute()*3A1.

Якщо текст займає декілька рядків, то постає питання його форматування. Для цього замість класу *TextLayout* використовується клас LineBreakMeasurer, методи якого дозволяють відформатувати абзац3A2. Для кожного сегменту тексту можна одержати екземпляр класу *TextLayout* і вивести текст, використовуючи його атрибути3A3.

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 його3A4.

Нарешті, можна задати окремі правила для виведення кожного символу тексту3A5. Для цього треба одержати екземпляр класу *GlyphVector* методом

createGlyphVector()

класу *Font*, змінити позицію символу методом

setGlyphPosition(),

задати перетворення символу, якщо це допустимо для даного шрифту, методом

setGlyphTransform()

і вивести змінений текст методом

drawGlyphVector()

класу *Graphics2D*3A6.

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)*3A7.

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

Таблиця 15 – Методи візуалізації і їх значення

	Методи (ключі)	Значення
3A8	KEY_ANTIALIASING	Розмивання крайніх пікселів ліній для гладкості зображення; задаються константами: VALUE_ANTIALIAS_DEFAULT, VALUE_ANTIALIAS_ON, VALUE_ANTIALIAS_OFF
3A9	KEY_TEXT_ANTIALIASING	То же для тексту. Константи: VALUE_TEXT_ANTIALIASING_DEFAULT VALUE_TEXT_ANTIALIASING_ON, VALUE_TEXT_ANTIALIASING_OFF
3AA	KEY_RENDERING	Три типи візуалізації. Константи: VALUE_RENDER_SPEED, VALUE_RENDER_QUALITY, VALUE_RENDER_DEFAULT
3AB	KEY_COLOR_RENDERING	То же для кольору. Константи: VALUE_COLOR_RENDER_SPEED, VALUE_COLOR_RENDER_QUALITY, VALUE_COLOR_RENDER_DEFAULT
3AC	KEY_ALPHA_ INTERPOLATION	Плавне сполучення ліній. Константи: VALUE_ALPHA_INTERPOLATION_SPEED, VALUE_ALPHA_INTERPOLATION_QUALITY, VALUE_ALPHA_INTERPOLATION_DEFAULT
3AD	KEY_INTERPOLATION	Способи сполучення. Константи: VALUE_NTERPOLATIO_BILINEAR, VALUE_INTERPOLATION_BICUBIC, VALUE_INTERPOLATION_NEAREST_NEIGHBOR
3AE	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 насправді3AF. 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 Java2D?*
2. *Як можна здійснити перетворення системи координат у Java?*
3. *Назвіть основні методи класу AffineTfansform. Яке їх функціональне призначення?*
4. *Які особливості малювання фігур засобами Java2D?*
5. *Чи використовує Java2D градієнтні методи заливки? Якщо так, то яким чином і які класи для цього використовуються?*
6. *Які особливості виведення текчту засобами Java 2D?*
7. *Які методи поліпщення візуалізації ви знаєте? В чому їх сутність?*

5 ОБРОБКА ФАЙЛІВ І СТВОРЕННЯ МЕРЕЖНИХ ПРОГРАМ

5.1 Потоки та файли

Насамперед зазначимо, що два англомовних терміна “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”.

Отже, які типи потоків пропонує нам Java.

Потоки, пов’язані з локальними файлами;

Потоки, пов’язані з даними в ОП;

Канальні потоки, тобто потоки для передачі даних між різними обчислювальними потоками;

Стандартні потоки введення-виведення.

Основну увагу нами буде приділено файловим потокам.

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

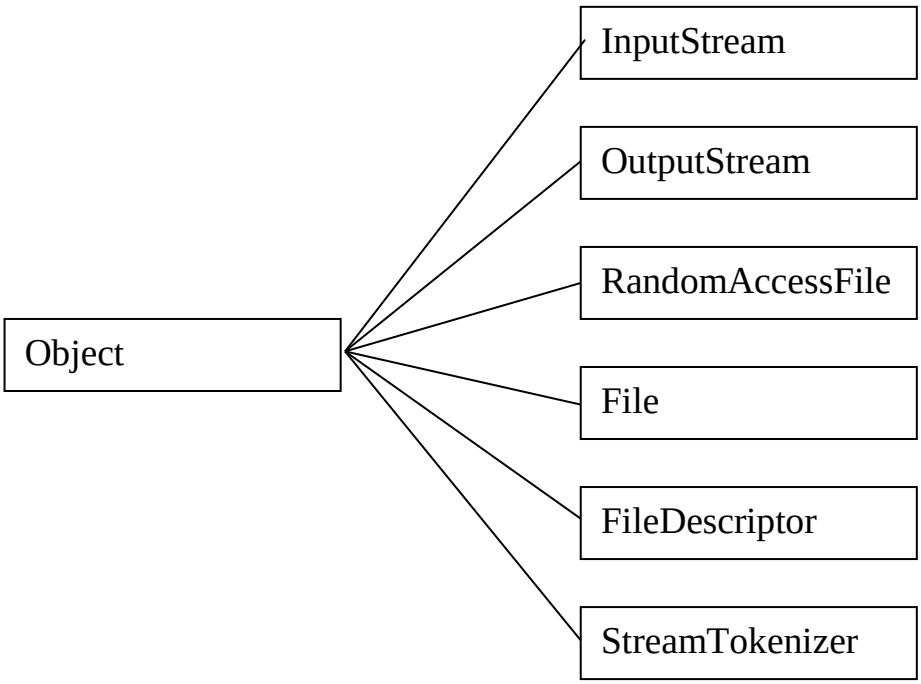

Рисунок 9 – Основні потокові класи

Клас *InputStream* є базовим для більшості класів, на основі яких створюються потоки введення. Саме ці похідні класи і

використовується на практиці. Щодо *InputStream* – це абстрактний клас, який містить декілька корисних методів, а саме3B6:

int read() – зчитує з вхідного потоку окремі байти як цілі числа та повертає –1, коли більше нема чого читати; 3B7

int read(byte b[]) – зчитує множину байтів в байтовий масив, повертаючи кількість реально введених байтів; 3B8

int read(byte b[], int off, int len) – також читає дані в байтовий масив, але дозволяє крім того задати ще зсув в масиві та максимальну кількість зчитаних байтів; 3B9

long skip(long n) – пропускає n байтів в потоці; 3BA

int available() – повертає кількість байтів, які є в потоці в даний момент; 3BB

void mark(int readlimit) – помічає поточну позицію в потоці, *readlimit* – кількість байтів, які можна прочитати з потоку до моента, коли помічена позиція втратить свою силу; 3BC

void reset() – повертається до поміченої позиції в потоці; 3BD

markSupported() – повертає бульове значення, яке вказує, чи можна в даному потоці відмічати позиції та повертатися до них; 3BE

void close() – закриває потік3BF.

Цей клас має аж шість прямих та чотири непрямих нащадки, 3C0

Клас *OutputStream* утворює пару до класу *InputStream*3C1. Основні методи 3C2:

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-виведення3C3.

Клас *File* призначений для роботи з заголовками каталогів та файлів3C4. 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 **класа *FileDescriptor*** можна перевірити ідентифікатор відкритого файла3C53C6.

Клас *StreamTokenizer* дозволяє організувати виділення з вхідного потоку даних елементів, що відділяються один від іншого заданими розділювачами3C7.

Класи, похідні від InputStream

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

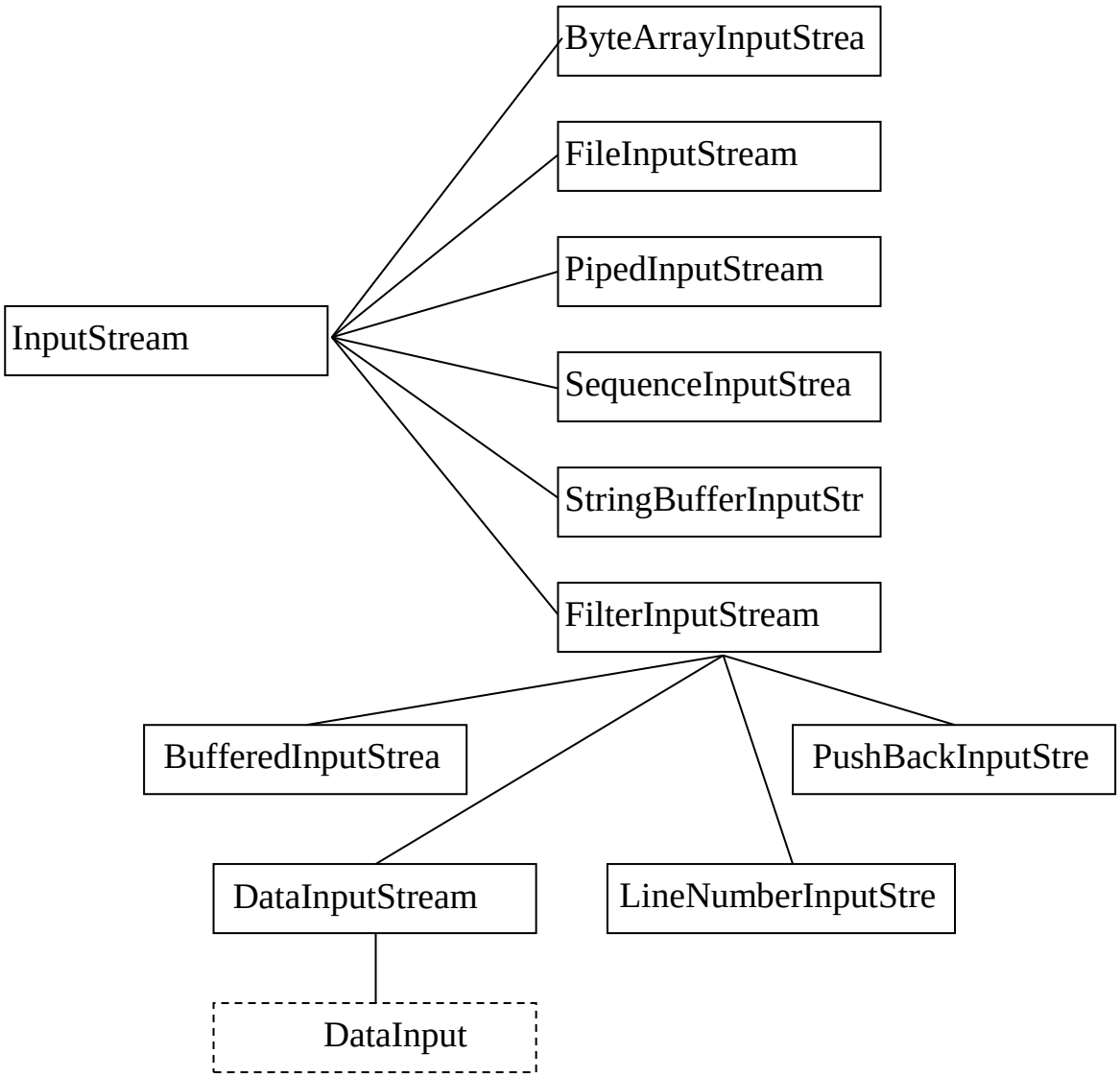

Рисунок 10 – Класи, похідні від *InputStream* ЗС8

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

Клас FileInputStream майже не має ніяких методів на додаток до тих, що визначено у його батька InputStreamЗС9. Але у нього є три потужних конструкториЗСА:

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

Приклад використання класа *FileInputStream* – програма, що виводить на екран свій вихідний текстЗСВ.

```
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);
                }
        }
}
```

Здавалося б, щоб вивести на екран програму, яка складається з багатьох рядків, необхідно використання циклів. Але в Java, як і в C, рядок обмежується символом ‘\0’ і не збігається з його представленням на екрані3CC. Тому весь текст програми – це один рядок, який можна вивести на екран за допомогою лише одного метода println()3CD.

Клас *FilterInputStream* надає можливість з’єднання потоків3CE. Для чого це потрібно? Як ми бачили, базовий вхідний потік *InputStream* надає лише метод *read*() для читання байтів3CF. 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. І знову – ніяких нових методів, тільки конструктор. Він має вигляд 3D0

```
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*3D1. 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 класів3D2. І останнє. Клас *FilterInputStream* є абстрактним, отже екземплярів цього класу створювати не будемо, а будемо використовувати його класи-нащадки3D3.

Зокрема **клас *DataInputStream*** є похідним від класу *FilterInputStream*3D4. Він надає можливості для читання всіх вбудованих типів даних Java, а також рядків3D5. Як створити екземпляр цього класу, щоб отримати доступ до його методів? Наприклад 3D6,

```
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 зустрічалис. Коли ми створювали першу програму, записуючи

System.out.println("Hello World!");

ми використали метод *println()*, визначений в класі *PrintStream*3D7.

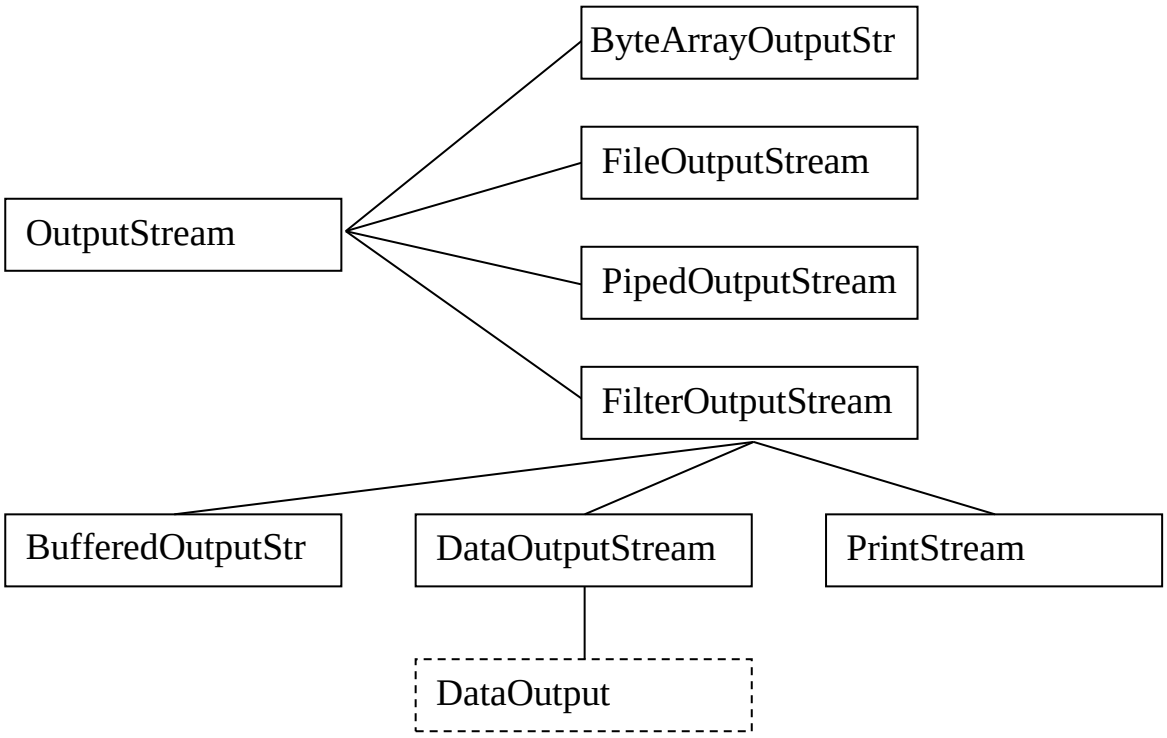

Рисунок 11 – Класи, похідні від *OutputStream*3D8

Тепер можна прояснити деякі моменти, які тоді залишилися нез'ясованими. Справа в тому, що *out* – це екземпляр класу *PrintStream*3D9. І цей екземпляр є одним з полів класу *System*, в якому є ще два поля – *in* і *err*3DA. 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*3DB. Сам же клас *System* має описувач *final* і містить ще декілька 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 представлення у вигляді текстового рядка3DC.

Приклад. Фрагмент програми, яка при натисненні на кнопку *Save* записує в файл зміст текстового поля3DD.

```
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 фрагменту. 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.*

5.2 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 рівень3DE. 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 дані3DF. IP характеризується як ненадійний протокол, тому що він не знаходить помилок і не виконує повторне пересилання даних3E0. Що таке IP- та DNS-адреса можна прочитати в [2].

Протоколи TCP та UDP відносяться до транспортного рівня3E1. Протокол TCP забезпечує надійний зв'язок на основі логічного з'єднання з неперервним потоком даних3E2. Доставка TCP-пакета схожа з доставкою листа за замовленням.

З неперервним потоком даних. TCP забезпечує механізм передачі, що дозволяє пересилати довільну кількість байтів3E3. Дані нарізаються на сегменти певної довжини, завдяки чому для рівня додатків емулюється неперервний потік даних3E4.

На основі логічного з'єднання. Перед початком передачі даних TCP встановлює з віддаленою машиною з'єднання, обмінюючись службовою інформацією3E5. Цей процес носить назву handshaking – рукостистискання3E6. 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 сегмент3E7.

Доставка UDP-пакета схожа з доставкою поштової картки, яку кидають у скриньку сподіваючись, що хтось її звідти рано чи пізно дістане3E8.

Протокол UDP має такі характеристики:

Оснований на повідомленнях3E9. UDP дозволяє додаткам пересилати інформацію у вигляді повідомлень (дейтаграм), які є одиницями передачі даних в UDP3EA. Розподіл даних по окремим дейтаграмам має виконувати сам додаток3EB.

Без встановлення логічного з'єднання3EC. Ніякого обміну службовою інформацією на початку з'єднання3ED. Інформація передається в припущенні, що приймальна сторона її очікує3EE.

Ненадійний3EF. UDP не має ані вбудованого механізму знаходження помилок, ані повторного пересилання спотворених або втрачених даних3F0.

Як бачите, майже все говорить на користь TCP. Тоді для чого ж нам UDP, запитаєте Ви. 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 чудовий TCP потребує великих додаткових витрат, а це не завжди виправдано3F1. Приклад використання UDP – служба часу3F2.

Універсальна адреса ресурсів URL

Адреса IP дозволяє ідентифікувати вузол, але його недостатньо для ідентифікації ресурсів, що є на цьому вузлі (працюючі додатки або файли)3F3. Причина очевидна – на вузлі, що має одну адресу IP, може існувати багато різних ресурсів3F4.

Для посилання на ресурси мережі Internet застосовується так звана універсальна адреса ресурсів *URL (Universal Resource Locator)*3F5. 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 ресурсу3F6. Документи HTML, наприклад, передаються з серверу Web видаленим користувачам за допомогою протоколу HTTP3F7. Файлові сервери в мережі Internet працюють з протоколом FTP3F8.

Для посилання на мережеві ресурси через протокол HTTP використовується наступна форма універсальної адреси ресурсів URL3F9:

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

Параметр host обов'язковий. Він повинен бути вказаний як доменна адреса або як адреса IP (у вигляді чотирьох десяткових чисел)3FA. Наприклад:

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

Необов'язковий параметр *port* задає номер порту для роботи з сервером3FB. 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 мережі3FC. 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 даних3FD. Наприклад, так потрібно указувати в адресі URL номер порту:

http://www.myspecial.srv:82

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

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

Якщо як адреса URL вказати навігатору тільки доменне ім'я серверу, сервер перешле навігатору свою головну сторінку. Ім'я файлу цієї сторінки залежить від серверу. 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 яке-небудь ще, визначене при установці серверу, наприклад, home.html або home.htm.

Для посилання на конкретний документ HTML або на файл будь-якого іншого об'єкту необхідно вказати в адресі URL його шлях, що включає ім'я файлу, наприклад3FE:

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

Кореневий каталог серверу Web позначається символом /. Якщо для протоколу HTTP шлях не заданий, то використовується кореневий каталог3FF.

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*400. Простота створення мережевих додатків з використанням цього класу в значній мірі 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 методів401.

Конструктори класу URL**402**:

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 мережевого ресурсу, адреса URL якого передається конструктору у вигляді текстового рядка через єдиний параметр *url*, в процесі створення об'єкту перевіряється задана адреса URL, а також наявність вказаного ресурсу403. Якщо адреса вказана невірно або заданий ресурс відсутній, виникає виключення *MalformedURLException*404.

Другий варіант конструктора класу URL допускає роздільне вказання протоколу, адреси вузла, номери порту і імені файлу[405].

Третій варіант припускає використання номера порту за замовчуванням[406]. Наприклад, для протоколу HTTP це порт з номером 80[407].

І, нарешті, четвертий варіант конструктора допускає вказання контексту адреси URL і рядка адреси URL[408]. Рядок контексту дозволяє вказати компоненти адреси URL, що відсутні в рядку *url*, такі як протокол, ім'я вузла, файлу або номер порту[409].

Методи класу URL:

openStream() – дозволяє створити вхідний потік для читання файлу ресурсу, пов'язаного із створеним об'єктом класу URL[40A]. Для виконання операції читання із створеного таким чином потоку ми можемо використовувати будь-який метод *read()*, визначений в класі *InputStream*[40B]. Дану пару методів (*openStream()* з класу *URL* і *read()* з класу *InputStream*) можна застосувати для вирішення задачі отримання вмісту двійкового або текстового файлу, що зберігається в одному з каталогів серверу Web[40C]. Зробивши це, звичний додаток Java або аплет може виконати локальну обробку одержаного файлу на комп'ютері видаленого користувача;

getContent() – визначає і одержує вміст мережевого ресурсу, для якого створений об'єкт URL[40D]. Практично можна використовувати цей метод для отримання текстових файлів, розташованих в мережевих каталогах[40E]. На жаль, даний метод непридатний для отримання документів HTML, оскільки для даного ресурсу не визначений обробник вмісту, призначений для створення об'єкту[40F]. Метод *getContent()* не здатний створити об'єкт ні з чого іншого, окрім текстового файлу[410]. 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*[411];

getHost() – визначає ім'я вузла, що відповідає даному об'єкту URL[412];

getFile() – дозволяє одержати інформацію про файл, пов'язаний з даним об'єктом URL[413];

getPort() – визначає номер порту, на якому виконується зв'язок[414];

getProtocol() – визначає протокол, з використанням якого встановлено з'єднання з ресурсом, заданим об'єктом URL[415];

getRef() – повертає текстовий рядок посилання на ресурс, що відповідає даному об'єкту URL[416];

hashCode() – повертає хеш-код об'єкту URL[417]:

sameFile() – визначіє, чи посилаються два об'єкти класу URL на один і той самий ресурс[418]. Якщо об'єкти посилаються на один ресурс, метод повертає *true*, якщо ні – *false*[419].

equals() – визначіє ідентичність адрес URL, заданих двома об'єктами[41A]. Якщо адреси URL ідентичні, метод повертає true, якщо ні - значення false[41B].

toExternalForm() – повертає текстовий рядок зовнішнього представлення адреси URL, визначеної даним об'єктом класу URL[41C];

toString() – повертає текстовий рядок, що представляє даний об'єкт[41D];

openConnection() – призначений для створення каналу між додатком і мережевим ресурсом, представленим об'єктом класу URL[41E]. Якщо ми створюємо додаток, який дозволяє читати з каталогів серверу Web текстові або двійкові файли, можна створити потік методом *openStream()* або одержати вміст текстового ресурсу методом *getContent()*[41F]. 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*[420]. Така методика дозволяє визначити чи встановити перед створенням потоку деякі характеристики каналу, наприклад, задати кешування[421]. Проте найцікавіша можливість, яку надає цей метод, полягає в організації взаємодії додатку Java і серверу Web[422].

Приклад програми використання адреси URL для отримання Web-сторінки[423]:

```
import java.net.*;
import java.io.*;
public class ReadURL
{   public static void main( String agr[])
   { try {                    //Об'являем адрес URL
                URL myURL = new URL("http://www.yahoo.com/");
                           //Створюємо потів введення
                InputStream in = myURL.openStream();
                int ch;
                           //Читаем з потоку и 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*424.

Але після створення об'єкту з'єднання ще не встановлено, і можна задати параметри зв'язку425. Наведемо деякі з методів, призначених для цього.

Методи для встановлення параметрів з'єднання426:

setDoInput(boolean doinput) – встановлює можливість використання потоку для введення; 427

setDoOutput(boolean dooutput) – встановлює можливість використання потоку для виведення; 428

setUseCaches(boolean usecaches) – включення або відключення кешування;

setDefaultUseCaches(boolean defaultusecaches) – включення або відключення кешування за замовчуванням; 429

setIfModifiedSince(long ifmodifiedsince) – установлення дати модифікації документа42A.

connect() – установлення з'єднання з об'єктом, на який посилається об'єкт класу URL42B.

Методи для отримання параметрів з'єднання42C:

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

Методи для витягання інформації із заголовка протоколу HTTP42D:

getContentEncoding() – повертає вміст *content-encoding* (кодування ресурсу, на який посилається URL), або *null*, якщо сервер його не вказав; 42E

getContentLength() – повертає довжину отриманої інформації (розмір документа в байтах), або –1, якщо сервер її не вказав; 42F

getContentType() – повертає тип інформації *content-type* (тип вмісту), тобто рядок типу *"text/html"* або *null* (якщо сервер не вказав); 430

getDate() – повертає дату посилки ресурсу в секундах з 01.01.1970; 431

getLastModified() – повертає дату зміни ресурсу в секундах з 01.01.1970; 432

getExpiration() – повертає дату застарівання ресурсу в секундах з 01.01.1970. [433]

Інші методи в класі *URLConnection* дозволяють одержати всі заголовки або заголовки із заданим номером, а також іншу інформацію про з'єднання[434].

Приклад програми, що пересилає рядок текста за адресою URL[435].

```
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 відповіді

              BufferedReader 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/IP436. 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, скажімо, що сокети – це програмний інтерфейс, призначений для передачі даних між додатками437.

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 сокета438.

З адресою вузла IP ми вже стикалися. Номер порту служить для ідентифікації додатку439. 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 HTTP43A.

Що ж до типів сокетів, то їх два – потокові і датаграмні43B.

За допомогою ***потокових сокетів*** ми можемо створювати канали передачі даних між двома додатками Java у вигляді потоків, які ми вже розглядали раніше43C. Потоки можуть бути43D:

- 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 додатками43E.

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 додатку, або скористатися датаграмними сокетами43F. ***Датаграмні сокети*** дозволяють передавати дані відразу всім вузлам мережі, хоча така можливість рідко використовується і часто блокується адміністраторами мережі440.

Для передачі даних через датаграмні сокети не потрібно створювати канал – дані посилаються безпосередньо тому додатку, для якого вони призначені з використанням адреси цього додатку у вигляді сокета і номера порту441. При цьому один клієнтський додаток може обмінюватися даними з декількома серверними додатками або навпаки, один серверний додаток – з декількома клієнтськими442.

На жаль, датаграмні сокети не гарантують доставку переданих пакетів даних443. Навіть якщо пакети даних, передані через такі сокети, дійшли до адресата, не гарантується, що вони будуть одержані в тій самій послідовності, в якій були передані444. Потокові сокети, навпаки, гарантують доставку пакетів даних, причому в правильній послідовності445.

Причина відсутності гарантії доставки даних при використанні датаграмних сокетів полягає у використанні такими сокетами протоколу UDP, який, у свою чергу, заснований на протоколі з негарантованою доставкою IP446. Потокові сокети працюють через протокол гарантованої доставки TCP447.

Сокет (socket) **–** це описувач мережного з'єднання448. Сокет TCP використовує протокол TCP, успадковуючи всі характеристики цього протоколу449. Для створення сокета TCP необхідно мати таку інформацію44A:

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

Сервер – це комп'ютер, який очікує звертань від різних машин з запитом конкретних ресурсів44B. Відповідно ***клієнти*** – це комп'ютери які звертаються з цими запитами до сервера44C. 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 дані44D. 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 порта44E. Порт – це абстракція (до речі, як і сокет), яка дозволяє розділити різні додатки, що можуть входити до мережі з одного і того самого комп'ютера44F. 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.

5.3 Робота з потоковими сокетами

Всю роботу в мережі інкапсульовано в пакеті *java.net*450. В Java є два класи, які дозволяють створювати мережні програми на основі сокетів: *Socket* та *ServerSocket*451.

Клас Socket

Клас *Socket* використовується для звичайного двобічного обміну даними452. Він має чотири конструктори453:

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

де *host* – це адреса комп'ютера, з яким треба установити зв'язок; *port* – номер порта на комп'ютері, з яким треба установити зв'язок454. Наприклад455,

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

Другий конструктор першим параметром отримує змінну класу *InetAddress*456:

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

Об'єкт класу *InetAddress* містить IP-адресу комп'ютера в мережі457. Цей клас не має конструкторів, але має цілу низку так званих фабричних (статичних) методів, які повертають екземпляри класу *InetAddress*458. Наприклад 459,

```
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-адресу45A.

Третій та четвертий конструктори аналогічні першим двом, але мають ще додатковий третій параметр, який дозволяє задати бульове

значення45B. 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)45C.

Клас *Socket* також має методи, що дозволяють читати та писати в нього: *getInputStream()* і *getOutputStream()*, які повертають потоки введення та виведення45D. Для зручності використання ці потоки звичайно надбудовуються добре відомими вам класами *DataInputStream* і *PrintStream* відповідно45E. І метод *getInputStream()*, і метод *getOutputStream()* викидає виняток *IOException*, який треба перехопити та обробити45F.

```
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 потоки460

```
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) 461:

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

Метод getLocalPort() повертає для даного сокета номер локального порту: 462

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

Завершуючи обмін даними треба закрити потоки, а потім і сам сокет463:

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

Клас ServerSocket

Щоб створити сокет TCP, необхідно скористатися класом, який дозволяє прив'язатися до порта та очікувати підключення клієнтів464. При кожному підключенні буде створено екземпляр класу *Socket*465. *ServerSocket* має два конструктори466:

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

Перший з них створює сокет, підключений до вказаного порта467. За замовчуванням в черзі очікування підключення може знаходитися до 50 клієнтів468. Другий конструктор дозволяє задавати довжину черги469.

Після створення об'єкта *ServerSocket* можна використовувати метод *accept()* для очікування підключення клієнтів46A. Цей метод блокується до момента підключення клієнта, а потім повертає об'єкт *Socket* для зв'язку з клієнтом46B. Блокування означає, що програма входить у внутрішній нескінчений цикл, який завершується за певних умов46C. Іншими словами, поки клієнт не підключиться, наступний за *accept()* оператор виконуватися не буде46D.

Наступний текст створює об'єкт *ServerSocket* з портом 4444, очікує підключення і далі створює потоки, через які буде виконуватись обмін даними з клієнтом, що підключився46E:

```
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);
}
```

Приклад46F

Приклад 470

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?*

5.4 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/передавання471.

Канал, а також вхідні та вихідні потоки створювати не треба472. Дані передаються та приймаються методами *send()* і *receive()*, визначеними в класі *DatagramSocket*473.

Клас DatagramSocket

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

В класі *DatagramSocket* визначено два конструктора474:

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

Перший з цих конструкторів дозволяє визначити порт для сокета, інший припускає використання будь-якого вільного порта475.

Звичайно серверні додатки працюють з використанням заздалегідь визначеного порта, номер якого є відомим для додатків-клієнтів476. Тому для серверних додатків ліпше використовувати перший з наведених вище конструкторів477.

Клієнтські додатки, навпаки, часто-густо використовують будь-які вільні на локальному вузлі порти, тому для них ліпшим є конструктор без параметрів478.

Звичайно перший з цих конструкторів використовують для серверів, які, як правило, знають самі і мають повідомити клієнтам номер свого порта, другий – для клієнтів479.

До речі, за допомогою метода *getLocalPort()* додаток завжди може довідатися номер порта, що його закріплено за даним сокетом47A:

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

Прийом і передавання даних на дейтаграмному сокеті виконується за допомогою методів *receive()* і *send()* відповідно47B:

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

Як параметр цим методам передається посилання на пакет даних, визначений як об'єкт класа *DatagramPacket*47C.

Ще один метод в класі *DatagramSocket*, яким ви маєте скористатися, це метод *close()*, призначений для закриття сокета47D:

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*47E. Метод *receive()* запише в такий об'єкт прийняті дані, а метод *send()* – перешле дані з об'єкта класу *DatagramPacket* вузлу, адреса якого указана в пакеті47F.

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 масива *length*480.

Якщо вам треба підготувати пакет для передавання, скористайтеся конструктором, який додатково дозволяє задати адресу IP *addr* и номер порта *port* вузла призначення481:

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*482.

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 пакет483.

Метод *getData()* повертає посилання на масив даних пакета484:

public byte[] getData();

Розмір пакета, дані з якого зберігаються в цьому масиві, легко визначити за допомогою метода *getLength()*485:

public int getLength();

Методи getAddress() і getPort() дозволяють визначити адресу та номер порта вузла, звідки прийшов пакет, або вузла, для якого призначено пакет486:

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

Якщо ви створюєте клієнт-серверну систему, в якій сервер має заздалегідь відому адресу та номер порта, 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'язку487.

Приклад 488

Приклад 489

Широкомовні пакети

Якщо адреса сервера невідома, клієнт може посилати широкомовні (рос. – широковещательные, англ. – broadcast) пакети, вказавши в об'єкті класа *DatagramPacket* адресу мережі48A. Така методика звичайно використовується в локальних мережах48B. Більш детально про широкомовні пакети можна прочитати в [2].

Наведемо фрагмент програми-сервера, який, отримавши запит від клієнта, витягує з нього всю необхідну інформацію та посилає відповідь48C

```
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.*