Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

І. Є. Бернакевич, П. П. Вагін

ПРОГРАМУВАННЯ МОВОЮ JAVA: ВИКОРИСТАННЯ ФУНДАМЕНТАЛЬНИХ КЛАСІВ

Тексти лекцій

Львів
Видавничий центр ЛНУ імені Івана Франка
2002

Бернакевич І. Є., Вагін П. П. Програмування мовою Java: використання фундаментальних класів: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002. – 48 с.

Розглянуто фундаментальні класи мови Java та їх використання, а саме: класи для роботи із рядками символів, ієрархію класів для обробки винятків, класи для розробки багатопотокових програм. Значну увагу приділено потоковому введенню-виведенню, серіалізації об'єктів та стисненню даних. Детально описано методи кожного із класів та наведено приклади їхнього використання.

Ці тексти лекцій започатковують серію публікацій, присвячену вивченню різних аспектів мови Java.

Для студентів та аспірантів факультету прикладної математики та інформатики.

Рецензент Львівський національний університет імені Івана Франка
канд.фіз.-мат.наук, доц. Р. В. Гудзь

© Бернакевич І. Є., Вагін П. П., 2002

Вступ

Мова Java виникла в 1991 році. Її розробили Джеймс Гослінг, Патрік Ноутон, Крис Варт, Ед Франк і Майкл Шерідан із компаніїї Sun Microsystem. Спочатку мова називалась Oak і лише в 1995 році перейменована в Java. Стимулом виникнення мови був зовсім не Internet, а потреба в незалежній від платформи мові, яка б могла використовуватись з метою створення програмного забезпечення для електронних пристроїв різноманітних користувачів. І лише під час розробки деталей мови виявилось, що незалежної від платформи мови потребує Web. Власне Internet сприяв висуненню мови Java на передову лінію програмування та її успіху.

Java запозичила синтаксис у мов С та С++. Проте Java не є Internet-версією С++, оскільки несумісна з нею. Розробники мови характеризують її так:

- проста
- безпечна
- переносна
- об'єктно-орієнтована
- стійка
- багатопотокова
- архітектрно незалежна
- інтерпретована
- високоефективна
- розподілена

Розглянемо коротко наведені характеристики.

Проста. Вважається, що вивчення Java спрощується, якщо програміст знайомий з принципами об'єктно-орієнтованого програмування. Оскільки Java виникла на основі С++, то програмістам С++ не знадобиться багато зусиль для переходу на Java. Крім того, деякі заплутані концепції С++ були вилучені із Java або спрощені.

Безпечна. Мова забезпечує захист, обмежуючи Java-програму середовищем її виконання, і не дозволяє їй отримати доступ до інших частин машини. Це стосується Java-аплетів, які динамічно завантажуються із мережі і виконуються під управлінням Web-браузера.

Переносна. Java-програма може виконуватися на комп'ютерах, які працюють під управлінням різних операційних систем. Безпечність і переносність програм досягається за допомогою використання байт-коду. Внаслідок компіляції Java-програми отримуємо високо-оптимізований набір команд, призначених для виконання віртуальною машиною Java, JVM (*Java Virtual Machine*). Це спрощує її виконання різноманітними середовищами. Єдиною вимогою є реалізація JVM для кожної платформи.

Об'єктно-орієнтована. Java – повністю об'єктно-орієнтована мова програмування. Неможливо створити програму поза межами класу.

Стійка. Мережне середовище ставить підвищені вимоги щодо надійності виконуваних програм. Java-програма повинна виконуватися надійно і передбачувано у різних середовищах. Java – мова зі строгою типізацією даних. Перевірка програми здійснюється як на етапі компіляції, так і у процесі її виконання. Найтиповіші помилки, зокрема управління пам'яттю та помилки часу виконання, усунені в Java. Зокрема, Java сама управляє розподілом і звільненням пам'яті, а виявлення помилок забезпечується об'єктно-орієнтованим обробленням виняткових ситуацій.

Багатопотокова. Java підтримує багатопотокове програмування, що дає змогу створювати програми, які паралельно виконують різні потоки команд. Це, відповідно, дає змогу створювати інтерактивні мережні програми й ефективно використовувати час процесора.

Архітектурно незалежна. Написане одного разу виконується скрізь, у будь-який час і завжди. Для досягнення цієї цілі розробники зробили деякі жорсткі обмеження у мові та віртуальній Java-машині.

Інтерпретована та високоефективна. Java дає змогу створювати міжплатформові програми завдяки їх компіляції у проміжковий байт-код, який може інтерпретуватися в будь-якій системі, забезпеченій віртуальною Java-машиною.

Розподілена. Java розроблена для розподіленого Internet-середовища. Java-програма може обробляти протоколи TCP/IP та здійснювати доступ до ресурсів із використанням URL. Більше того, технологія RMI (*Remote Method Invocation*) вносить новий рівень абстракції в програмування на основі клієнт-сервер, забезпечуючи віддалений виклик методів об'єкта, який виконується на іншому комп'ютері.

1. Рядки та споріднені класи

Рядок в Java – послідовність символів. На відміну від інших мов програмування, які реалізують рядки у вигляді масивів символів, рядки в Java реалізуються як об'єкти.

Для підтримки роботи зі рядками Java API містить три класи:

- **`java.lang.String`** – об'єднує текстові рядки Java. Об'єкти **`String`** є постійними (не змінними); одного разу створені, вони не можуть бути змінені (тобто не можна змінити послідовність символів рядка або його довжину);
- **`java.lang.StringBuffer`** – об'єднує рядки змінної довжини, в яких послідовність символів може змінюватися. Використання

об'єкта **StringBuffer** дає змогу вставляти символи скрізь у послідовності символів та додавати символи в кінець послідовності;

- **java.util.StringTokenizer** – забезпечує підтримку синтаксичного розбору рядка на послідовності слів або лексем.

1.1. Клас java.lang.String

Створення рядків

Є багато способів створення рядків у Java. Наприклад, рядок можна створити простим присвоєнням рядкового літералу змінній типу **String**:

```
String quote = "To be or not to be";
```

Усі рядкові літерали автоматично конвертуються в об'єкти **String**. На відміну від C++, об'єкти типу **String** не завершуються нульовим символом. Клас **String** використовує символьний масив для внутрішнього подання об'єкта. Оскільки масиви в Java є об'єктами, які знають свою довжину, об'єкт **String** також знає свою довжину і не потребує спеціального закінчення. Для визначення довжини рядка використовується метод **length()**:

```
int length = quote.length();
```

Крім цього, рядок можна створити шляхом конкатенації рядків. Наступні стрічки коду створюють однакові рядки:

```
String name = "John " + "Smith";
String name = "John ".concat("Smith");
```

Звичайно, є багато інших шляхів створення об'єктів **String.** Клас **String** має декілька конструкторів, які дають змогу створювати **String** з масиву символів або його частини:

```
char [] data={'L','e','m','m','i','n','g'};
String lemming = new String(data);
String lemming1 = new String(data, 3, 3);
```

із масиву байтів або його частини

```
byte [] data = { 97, 98, 99 ,100, 101};
String abcde = new String(data, "8859_5");
String abc = new String(data, 0, 3, "8859_5");
```

Останній параметр у конструкторі **String** задає схему кодування. Він використовується для конвертування байтів до символів Unicode. Якщо він не вказаний, то застосовується кодування за замовчуванням для заданої платформи.

Можна також створити рядок з іншого об'єкта **String**, або об'єкта **StringBuffer**.

Взаємне перетворення рядків та інших об’єктів

Хоча об’єкти **String** незмінні, клас **String** надає можливість застосування декількох корисних методів для роботи із рядками. Будь-які операції, які намагаються змінити символи чи довжину рядка, повертають новий рядок, який містить необхідну частину попереднього рядка.

Клас **String** визначає статичні методи **valueOf()**, які повертають рядкове подання простих типів даних та об’єктів Java:

```
String one = String.valueOf(1);
String two = String.valueOf(2.0f);
String notTrue = String.valueOf(false);
```

Більше цього, всі об’єкти Java мають метод **toString()**, успадкований від класу **Object**. Для об’єктів метод **String.valueOf()** викликає метод об’єкту **toString()** для отримання рядкового подання. Якщо посилання на об’єкт є нуль, то результатом є літерал **"null"**:

```
String date = String.valueOf(new Date());
System.out.println(date);
          // Sun Dec 19 05:45:34 CST 1999
date = null;
System.out.println(date); // null
```

Отримання простих типів із рядка не є функціональністю класу **String**. Для цього необхідно скористатись класами оболонками простих типів. Ці класи надають метод **valueOf()**, який генерує об’єкт із рядка, а також методи для відтворення відповідних простих типів. Наприклад:

```
int i=Integer.valueOf("123").intValue();
double d=Double.valueOf("123.0").doubleValue();
```

У вищенаведеному коді, **Integer.valueOf()** створює об’єкт **Integer**, який подає значення 123. Об’єкт **Integer** дає змогу отримати прості значення в формі **int** за допомогою методу **intValue()**. У цьому разі вимагається правильний рядок. В нижченаведеному прикладі генерується виняткова ситуація **NumberFormatException**:

```
double d =                      //помилка!
      Double.valueOf("1.234,56").doubleValue();
```

Добування символів

Клас **String** надає декілька методів для добування символів із рядка. Індексація символів рядка починається з нуля. Зокрема, метод **charAt()** класу **String** дає змогу отримати символи рядка подібним до масиву способом:

```
String s="Newton";
```

```
for(int i=0; i<s.length(); i++)
            System.out.println(s.charAt(i));
```

Цей код видруковує по одному символи рядка. Крім цього, є можливість перетворити рядок у символьний масив за допомогою методу **toCharArray()**:

```
char [] abcs="abcdef".toCharArray();
```

Методи **getChars()** і **getBytes()** повертають послідовність символів рядка у вигляді символьного чи байтового масивів відповідно. Наприклад, рядки коду

```
String s1 = "This is a first string of text";
int start = 10;
int end = 15;
char buf[] = new char [end - start];
s1.getChars(10, 15, buf, 0);
```

заповнять символьний масив **buf** підрядком **first**.

Порівняння рядків

Як і в С, не можна порівнювати рядки з використанням оператора "==", оскільки рядки є об'єктами. Для порівняння вмісту рядків застосовується метод **equals()**:

```
String one = "Foo";
char [] c = { 'F', 'o', 'o' };
String two = new String(c);
if (one.equals(two))     //правильно
```

Інший метод, **equalsIgnoreCase()**, використовується для перевірки еквівалентності рядків без врахування регістра:

```
String one = "FOO";
String two = "foo";
if (one.equalsIgnoreCase(two)) // правильно
```

Однак у процесі сортування рядків недостатньо їх порівняння щодо збігу. Метод **compareTo()** дає змогу визначити, який із рядків більший за лексичним значенням. Він повертає ціле значення, яке може бути менше від нуля, рівне нулеві або більше від нуля.

```
String abc = "abc";
String def = "def";
String num = "123";
if ( abc.compareTo( def ) < 0 )   // правильно
if ( abc.compareTo( abc ) == 0 )  // правильно
if ( abc.compareTo( num ) > 0 )   // правильно
```

Порівняння символів відбувається строго за їхнім розміщенням в кодовій таблиці Unicode.

Метод **regionMatches()** визначає, чи два рядки містять однакові підпослідовності символів. Прототип даного методу наступний:

```
boolean regionMatches([boolean ignoreCase,]
                      int start1, String s2,
                      int starts2,int nums2)
```

де **start1** визначає індекс початкової області для порівняння, **starts2** – індекс початкової області для порівняння в стрічці **s2**, **nums2** – кількість символів для порівняння. Необов'язковий параметр **ignoreCase** із значенням **true** дає змогу виконувати порівняння без врахування регістра. Наприклад:

```
String s1 = "First String";
String s2 = "Second sTring";
boolean b = s1.regionMatches(true,6,s2,7,6);
                                          //true
```

Методи **startsWith**() та **endsWith**() визначають, чи стрічковий об'єкт починається або закінчується деякою послідовністю символів. Окрім цього, можна встановити початкову точку області порівняння. Наприклад:

```
String s1 = "First string";
boolean b = s1.endsWith("string", 6);
```

Пошук рядків

Для пошуку окремого символу чи послідовності символів у стрічці використовуються наступні методи:

- **indexOf()** шукає перше входження символу чи рядка;
- **lastIndexOf()** шукає останнє входження символу чи рядка.

Ці методи мають декілька перевантажених версій, які дають змогу здійснювати пошук із початку рядка, або із вказаної позиції у рядку. У будь-якому випадку ці методи повертають індекс рядка, за яким було знайдено символ чи підрядок. Якщо ж пошук був невдалим, то повертається –1.

Редагування рядків

Багато методів, що оперують із рядками, повертають як результат новий рядок. Якщо потрібно модифікувати оригінал рядка, то ліпше використовувати об'єкт класу **StringBuffer**, який розглядається нижче.

Метод **substring()** дає змогу отримати підрядок рядка-оригіналу починаючи із вказаного індекса і до кінця рядка:

```
String s1 = "First string";
String s2 = s1.substring(6);        //string
```

або із зазначенням початкового та кінцевого індексів рядка-оригіналу. У цьому разі символ з початковим індексом вводиться в новий рядок, а кінцевий – ні:

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
String cde = abcs.substring(2, 5);// "cde"
```

Метод **replace()** дає змогу замінити всі входження одного символу іншим. Наприклад:

```
String s="Hello".replace("l","w");//Hewwo
```

Метод **trim()** застосовується для вилучення початкових і кінцевих пробільних символів (таких як пропуск, повернення каретки, новий рядок, символ табуляції) у рядку:

```
String abc = "   abc   ";
abc = abc.trim();                  // "abc"
```

У вищенаведеному прикладі ми втратили рядок-оригінал.

Методи **toUpperCase()** і **toLowerCase()** повертають новий рядок відповідного регістра:

```
String foo = "FOO".toLowerCase();
String FOO = foo.toUpperCase();
```

1.2. Клас java.lang.StringBuffer

Клас **java.lang.StringBuffer** репрезентує самозростаючий буфер символів. Він забезпечує багато функціональних можливостей для рядків. Клас визначає три конструктори:

StringBuffer() – резервує ділянку пам'яті для 16 символів

StringBuffer(int size) – **size** вказує на ромір буфера

StringBuffer(String s) – резервує пам'ять для **s.length()+16** символів та ініціалізує об'єкт стрічкою **s**.

Для визначення довжини рядки, яка зберігається в об'єкті **StringBuffer**, та загального обсягу пам'яті символьного буфера використовуються методи **length()** та **capacity()** відповідно. Задати розмір буфера вже створеного об'єкта **StringBuffer** можна за допомогою методу **ensureCapacity()**, передавши йому як параметр потрібну ємність буфера. Крім цього, метод **setLength()** дає змогу задати довжину рядка, який зберігається в об'єкті **StringBuffer**. Якщо методу передається число, менше за довжину рядка, то символи, які не поміщаються в нову довжину, будуть втрачені.

Використання **StringBuffer** є ефективним у коді, подібному до такого:

```
String ball = "Hello";
ball = ball + " there.";
```

```
ball = ball + " How are you?";
```

У вищенаведеному прикладі багаторазово створюється новий стрічковий об'єкт. Це означає, що масив символів мусить копіюватися знову і знову, знижуючи тим самим ефективність. Раціональнішим у цьому випадку є використання об'єкта **StringBuffer** і його методу **append()**:

```
StringBuffer ball = new StringBuffer("Hello");
ball.append(" there.");
ball.append(" How are you?");
```

Клас **StringBuffer** надає велику кількість перевантажених методів **append()** для приєднання різного типу даних до буфера. Наприклад, до об'єкта **StringBuffer** можна приєднати інший рядок, число простого типу чи деякий об'єкт. Для отримання рядкового подання параметра викликається метод **String.valueOf()**.

Клас **StringBuffer** також надає велику кількість перевантажених методів **insert()** для вставлення різного типу даних у визначену позицію символьного буфера. Зокрема, можна вставити рядок, число простого типу або деякий об'єкт. У цьому разі, аналогічно до методу **append()**, для отримання рядкового подання значення застосовується один із методів **String.valueOf()**.

Методи **charAt()** та **setCharAt()** дають змогу, відповідно, отримати чи змінити один символ об'єкта **StringBuffer**, використовуючи як параметр його індекс. Окрім цього, для одержання масиву символів, використовується метод **getChars()**. Його сигнатура аналогічна до відповідного методу класу **String**.

Методи **deleteCharAt()** та **delete()** дають змогу вилучати символ за його індексом або послідовність символів, розташованих між початковим **start** та кінцевим **end-1** індексами відповідно. Наприклад:

```
StringBuffer ball =
        new StringBuffer("Hello my world");
int start = 6;
int end = 9;
ball.delete(start, end);  //Hello world
```

Метод **reverse()** дає змогу змінити на обернений порядок розміщення символів об'єкта **StringBuffer**. Інший метод, **replace()**, замінює набір символів об'єкта **StringBuffer**, розташованих між початковим **start** та кінцевим **end-1** індексами. Наприклад:

```
StringBuffer ball =
        new StringBuffer("Hello my world");
```

```
String s1 = "friend";
int start = 9;
int end = 14;
ball.replace(start, end, s1); //Hello my friend
```

Крім цього, метод **substring()** дає змогу отримати підрядок рядка-оригіналу. Його синтаксис аналогічний до відповідного методу класу **String**. Можна отримати об’єкт **String** із **StringBuffer** за допомогою його методу **toString()**:

```
StringBuffer ball =
       new StringBuffer("Hello my world");
String message = ball.toString();
```

Багато обчислень використовують об’єкт **StringBuffer**. Наприклад, напишемо метод, який приймає об’єкт **String** і повертає новий **String**, що містить символи в оберненому порядку, використавши **StringBuffer** так:

```
public static String reverse(String s) {
    StringBuffer buf = new
        StringBuffer(s.length());
    for (int i = s.length()-1; i >= 0; i--) {
        buf.append(s.charAt(i));
    }
    return buf.toString();
}
```

Зазначимо, що для об’єднання рядків можна використовувати єдиний перевантажений оператор мови Java +. Яким чином компілятор використовує **StringBuffer** для реалізації операції конкатенації розглянемо на прикладі:

```
String foo = "To " + "be " + "or";
```

Це еквівалентне:

```
String foo = new
StringBuffer().append("To ").↵
       append("be ").append("or").toString();
```

Такого типу ланцюжки виразів є однією з особливостей прихованого перевантаженого оператора в інших мовах.

1.3. Клас java.util.StringTokenizer

Оброблення текстів складається із синтаксичного аналізу відформатованого рядка. Клас **java.util.StringTokenizer** забезпечує синтаксичний розбір рядка на послідовність слів (або лексем),

які відокремлені деякою множиною розділових символів. Нижче наведено приклад застосування класу **StringTokenizer**:

```
StringTokenizer s=new StringTokenizer("This is it");
      while (s.hasMoreTokens())
            System.out.println(s.nextToken());
```

Цей приклад починається зі створення об'єкта **StringTokenizer** для пошуку лексем заданого рядка. Використовується конструктор, який не визначає рядка роздільників слів, а тому за замовчуванням новий об'єкт **StringTokenizer** використовує такі роздільники: пропуск, символ табуляції ('\t'), символ переведення каретки ('\r') та символ нового рядка ('\n').

У циклі **while** відбувається фактичне добування лексем з об'єкта **StringTokenizer**. Метод **hasMoreTokens()** повертає **true**, допоки ще є лексеми для вибору з об'єкта **StringTokenizer** і поки метод **nextToken()** повертає наступну лексему. Нижче наведено результат синтаксичного аналізу рядка:

```
This
is
it
```

Також можна використати об'єкт **StringTokenizer** для добування лексем із рядка, в якій як роздільники використовуються інші символи. Наприклад, припустимо, що потрібно добути лексеми, розділені комами, як у стрічці, наведеній нижче:

```
String commaString = "abc,def,123,789";
```

У цьому випадку використаємо конструктор із додатковим параметром, який специфікує рядок роздільників. Наприклад:

```
StringTokenizer s =
       new StringTokenizer(commaString, ",");
```

Другий аргумент у цьому конструкторі специфікує символи-роздільники, що в цьому випадку є одним символом – комою.

2. Виняткові ситуації

Виняткова ситуація (ВС) – це подія, яка порушує нормальне виконання програми. Java підтримує універсальну систему оброблення помилок – механізм винятків, який передбачає генерацію винятку, і його обробку. ВС можуть генеруватися як системою, так і програмістом.

У процесі роботи із винятками використовуються такі ключові слова: **try**, **catch**, **throw**, **throws**, **finally**. Потенційно-небезпечні оператори поміщають у блок **try**. У випадку виникнення виняткової

ситуації її перехоплюють в операторі **catch**. Виняткові ситуації виконавчої системи Java генеруються автоматично. Щоб самому згенерувати виняткову ситуацію, використовується ключове слово **throw**. Оператор **throws** описує виняткові ситуації, які можуть генеруватись методом, але ним не обробляються. Оператори, які необхідно у будь-якому випадку виконати, поміщають у блок **finally**. В загальному випадку шаблон перехоплення та обробки виняткових ситуацій має такий вигляд:

```
try{    …    }
    catch(ExcepionType1 e){    …    }
    [catch(ExceptionType2 e){    …    }]
[finally{    …    }]
```

Зазначимо, що блок **try** повинен мати хоча б один, відповідний йому, **catch**-блок, або блок **finally**.

Якщо не обробляти винятки в програмі, то виконавча система у процесі виникнення помилки сама створить новий об'єкт винятку та згенерує його. У цьому разі виконання програми буде зупинено, оскільки будь-який згенерований виняток повинен бути негайно перехоплений. Оскільки в програмі відсутній обробник виняткової ситуації, то її перехопить обробник винятків виконавчої системи Java, заданий за замовчуванням. Розглянемо наступний приклад:

```
class Example0{
public static void main(String args[])
{int s=0;
int a=77/s;}
}
```

У випадку ділення на нуль буде згенерована ВС **ArithmeticException**. Оскільки в програмі відсутній обробник ВС, то обробку помилки здійснить обробник виконавчої системи, який відобразить на екрані опис ВС, трасу стеку від точки виникнення помилки та завершить виконання програми. Вивід цього прикладу буде таким:

```
java.lang.ArithmeticException: /by zero//опис помилки
at Example0.main(Example0.java: 4)      //траса стека.
```

Траса стека містить ім'я класу (**Example0**), ім'я методу (**main**), ім'я файла (**Example0.java**) та номер рядка (**4**).

Стек викликів корисний у випадку відлагодження програми. Він вказує послідовність викликів методів, які спричинили ВС. Розглянемо попередній приклад, модифікований таким чином, що ВС виникає не в методі **main**, а в методі **sub**

```
class Example1{
static void sub()
```

```
{int s=0;
int a=77/s;}
public static void main(String args[]){
Example1.sub();}
}
```

У цьому випадку траса стеку буде дещо іншою:

```
java.lang.ArithmeticException: /by zero//опис помилки
      at Example1.sub(Example1.java:4)
      at Example1.main(Example1.java:6)//траса стека.
```

На дні стеку рядок 6 методу **main**, яка звернулась до методу **sub**, який згенерував ВС. Якщо виняток не обробляється методом **sub**, він опускається вниз по стеку викликів у метод **main**. Тут теж не обробляється виняток, а тому він передається обробнику ВС виконавчої системи.

Розглянемо кожен з етапів механізму обробки виняткових ситуацій більш детально.

2.1. Виявлення й обробка виняткових ситуацій

Хоча виконавча система Java надає обробник винятків за замовчуванням, обробка винятків в програмі має деякі переваги: по-перше, дає змогу зафіксувати помилку, по-друге – попереджує автоматичне завершення програми.

Для виявлення й обробки помилок код поміщають у блок **try**, після якого розміщують один чи декілька блоків **catch** із вказаним типом винятку. Розглянемо попередній приклад, в якому використовується обробка винятків:

```
class Example2{
public static void main(String args[])
{int s,a;
 try{
 s=0;
 a=77/s;
 System.out.println
       ("Цей рядок не буде надрукований");
}catch(ArithmeticException e){
   System.out.println("Ділення на нуль");
}
System.out.println("Після оператора catch");
  }
}
```

Результатом виконання програми буде таке повідомлення:

```
Ділення на нуль
Після оператора catch
```

Зазначимо, що після виникнення виняткової ситуації управління передається на відповідний блок **catch**, після чого виконання програми продовжується з оператора, який розміщений за блоком **try-catch**, а тому повідомлення “**Цей рядок не буде надрукована**” ніколи не буде виведене на екран.

Блоки **try-catch** взаємопов'язані. Область видимості **catch** обмежена найближчим блоком **try**, що йому передує. Оператор **catch** не може обробити виняток, згенерований в іншому блоці **try**, за винятком вкладених операторів **try**.

Якщо в будь-якому випадку необхідно виконати якість дії, то їх поміщають у блоці **finally.** Код цього блоку виконується незалежно від того, чи був згенерований виняток, чи ні. Крім того, якщо **catch**-блок відсутній, перед поверненням із **try**-блоку безумовно будуть виконані оператори блоку **finally**. Наприклад:

```
try {
    out.write(b);
} catch (IOException e) {
    System.out.println("Output Error");
} finally {
    out.close();
}
```

Якщо **out.write()** згенерує помилку **IOException**, виняток буде перехоплений у **catch**-блоці. Незважаючи на те, чи **out.write()** завершиться нормально чи згенерує помилку, блок **finally** буде виконаний, що гарантує безумовне виконання **out.close()**.

В одній ділянці програми може бути згенеровано декілька винятків. Тоді можна задати декілька блоків **catch** для обробки різних типів помилок. У випадку виникнення помилки **catch**-блоки послідовно переглядаються на відповідність типу згенерованого винятку. У цьому разі виконується тільки один із блоків **catch**, решта – ігноруються.

Зауваження. При використанні декількох блоків **catch** важливо, щоб підкласи ВС оброблялися швидше від їхніх суперкласів. Це зумовлено тим, що в операторі **catch** здійснюється обробка ВС зазначеного класу та всіх його підкласів. Тому блок **catch** підкласу не буде перевірятися, якщо він розміщений після аналогічного блоку суперкласу. Крім

того, виконавчою системою Java в цій ситуації буде згенерована помилка.

Розглянемо такий код:

```
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println
            ("Generic Exception catch.");
    }
    /* Цей catch – недосяжний,
       оскільки ArithmeticException
       є підкласом Exception. */
    catch(ArithmeticException e) { // ПОМИЛКА
    System.out.println("Недосяжний оператор.");
    }
  }
}
```

У процесі компіляції програми буде згенерована помилка, оскільки **ArithmeticException** є підкласом **Exception**. Перший оператор **catch** обробляє помилки свого класу та всіх своїх підкласів, зокрема і **ArithmeticException**. А отже, другий оператор **catch** ніколи не виконається.

2.2. Виведення опису виняткових ситуацій і роздрук стека викликів

У класі **Throwable** метод **toString()** перекрито таким чином, що він повертає рядок з описом ВС. Цей опис можна вивести з допомогою **println()**, передавши ВС як параметр. Наприклад, унаслідок виконання рядків коду

```
catch (ArithmeticException e){
    System.out.println("Exception"+e);
    a=0;}
```

буде надруковано рядок

Exception: java.lang.ArithmeticException: /by zero.

У процесі перехоплення винятку може знадобитися роздрукувати стек викликів для локалізації пошуку винятку. Траса стеку має такий вигляд:

```
java.lang.ArithmeticException: / by zero
        at t.cap(t.java:16)
        at t.doit(t.java:8)
        at t.main(t.java:3)
```

Можна роздрукувати трасу стека шляхом виклику методу **printStackTrace()**, який успадковується всіма класами похідними від **Throwable**. Наприклад:

```
int cap (x) {return 100/x}
try {
    cap(0);
} catch(ArithmeticException e) {
    e.printStackTrace();
}
```

Роздрукувати трасу стека можна також у процесі генерації винятку:

```
new Throwable().printStackTrace();
```

2.3. Оголошення винятків

Якщо в методі очікується генерація деяких винятків, у сигнатурі методу необхідно констатувати цей факт за допомогою оператора **throws**. Якщо реалізація методу містить оператор **throw**, то ймовірно, що виняток буде згенерований зсередини методу. Крім цього, якщо метод викликає інші методи, оголошені з використанням оператора **throws**, існує можливість генерації винятку із середини цього методу. Якщо виняток не перехоплюється всередині методу за допомогою оператора **try**, він буде згенерований зовні цього методу. Будь-який виняток, який може бути згенерований у методі, повинен бути зареєстрований в сигнатурі цього методу за допомогою оператора **throws**. Класи, зареєстровані в операторі **throws**, повинні бути підкласами **Throwable** або будь-яких його підкласів. **Throwable** є суперкласом для всіх типів винятків, які можна згенерувати в Java.

Однак існують типи винятків, які немає необхідності реєструвати за допомогою оператора **throws**, а саме якщо виняток є екземпляром класів **Error**, **RuntimeException** або їхніх підкласів.

Розглянемо приклад:

```
import java.io.IOException;
class throwsExample {
char[] a;
int position;
// Метод явно генерує виняток
int read() throws IOException {
```

```
   if (position >= a.length)
      throw new IOException();
return a[position++];
}
// Метод явно генерує виняток
String readUpTo(char terminator)
                 throws IOException {
StringBuffer s = new StringBuffer();
while (true) {
int c = read();// Може генерувати IOException
   if (c == -1 || c == terminator) {
      return s.toString();
      }
   s.append((char)c);
   }
return s.toString();
}
// Метод перехоплює виняток усередині
int getLength() {
String s;
try { s = readUpTo(':');
   } catch (IOException e) {
   return 0;
}
return s.length();
}
// Метод може генерувати RuntimeException
int getAvgLength() {
int count = 0;
int total = 0;
int len;
while (true){
   len = getLength();
   if (len == 0)
      break;
   count++;
   total += len;
} // Може генерувати ArithmeticException
return total/count;
}
}
```

Метод **read()** може генерувати **IOException**, тому оголошує цей факт у своєму операторі **throws**. Якщо цього не зробити, компілятор видасть повідомлення, що метод повинен або оголосити виняток **IOException** у сигнатурі методу, або перехопити його. Хоча метод **readUpTo()** явно не генерує жодного винятку, він викликає метод **read()**, який генерує виняткову ситуацію, а тому оголошує це в своїй сигнатурі за допомогою оператора **throws**. **getLength()** перехоплює виняток **IOException**, згенерований методом **readUpTo()**, так що він не повинен оголошувати цей виняток. Метод **getAvgLength()** може генерувати виняткову ситуацію **ArithmeticException**, якщо значення змінної **count** – нуль. Оскільки **ArithmeticException** – підклас **RuntimeException**, то немає потреби оголошувати його за допомогою оператора **throws**.

Зауваження. Java вимагає, щоб методи перехоплювали або оголошували всі винятки, які можуть виникнути до виконання програми і які можуть бути згенеровані в області видимості методу. Тобто якщо у процесі виконання методу може виникнути виняткова ситуація, яка в цьому методі не обробляється, то в сигнатурі методу необхідно оголосити, що метод може генерувати виняткову ситуацію (оператор **throws**). В цьому не має необхідності для виняткових ситуацій, таких як **Error**, **RuntimeException** та їхніх підкласів.

2.4. Генерація винятків

Досі розглядалися винятки, які генерувалися виконавчою системою Java. Однак програма сама може генерувати винятки. Для цього використовується оператор **throw**. Наприклад:

```
throw new AnyException;
```

де **AnyException** – генерований об’єкт.

Генеровані об’єкти – це екземпляри підкласів **Throwable** (використання інших типів заборонене). Є два способи оголошення об’єкта **Throwable** – використати параметр **catch**-блоку або створити новий об’єкт з допомогою оператора **new**.

```
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
     System.out.println("Перехоплення "+
```

```
                                  "всередині demoproc.");
        throw e; // повторна генерація винятку
      }
    }
    public static void main(String args[]) {
      try {
        demoproc();
      } catch(NullPointerException e) {
        System.out.println("Повторне "+
                                  "перехоплення: " + e);
      }
    }
```

У всіх вбудованих ВС Java є два конструктори: один – без параметрів, інший – з параметром типу **String**. Цей рядок містить опис виняткової ситуації і використовується, коли об'єкт ВС виступає параметром у методі **println()**. Цю рядок можна отримати і з допомогою методу **getMessage()** класу **Throwable**.

Після перехоплення винятку він може бути повторно згенерований. Найкраще, що можна зробити у випадку повторної генерації винятку – потурбуватися про виявлення місцезнаходження згенерованого винятку.

Для повторної генерацію винятку й отримання траси стека необхідно просто повторно згенерувати виняток:

```
try {
    cap(0);
} catch(ArithmeticException e) {
    throw e;
}
```

Для підготовки траси стека, яка відображає фактичне місце, з якого виняток був повторно згенерований, необхідно викликати метод **fillInStackTrace()**. Цей метод з'ясовує інформацію траси стека, базуючись на поточному контексті винятку. Нижче наведено приклад із використанням методу **fillInStackTrace()**:

```
try {
    cap(0);
} catch(ArithmeticException e) {
    throw
(ArithmeticException)e.fillInStackTrace();
}
```

Важливо викликати метод **fillInStackTrace()** у тому ж рядку, що оператор **throw**, оскільки номер рядка, вказаний у стеку, зіставляється з рядком, в якому виявлено оператор **throw**. Метод

fillInStackTrace() повертає посилання на об'єкт **Throwable**, тому необхідно звести посилання до фактичного типу винятку.

2.5. Створення власних класів виняткових ситуацій

Незважаючи на те, що вбудовані винятки Java забезпечують обробку найбільш загальних помилок, у деяких випадках доцільно створити власний тип ВС, специфічний для певного застосування. Для цього необхідно визначити підклас класу **Exception**. У цьому разі немає потреби що-небудь реалізовувати. Саме існування класу дає змогу використовувати його як виняток. Розглянемо приклад:

```
class WrongDayException extends Exception {
   public WrongDayException () {}
   public WrongDayException(String msg) {
   super(msg);
   }
}
public class ThrowExample {
   void doIt() throws WrongDayException{
   int dayOfWeek =
          (new java.util.Date()).getDay();
   if (dayOfWeek != 2 && dayOfWeek != 4)
     throw new WrongDayException(
                 "Вівторок або Четвер.");
   System.out.println("Все в порядку.");
   }
public static void main (String [] argv) {
   try {
        (new ThrowExample()).doIt();
        } catch (WrongDayException e) {
   System.out.println("Вибачте, це можна " +
          "зробити тільки у "+ e.getMessage());
      }
   }
}
```

Визначається клас **WrongDayException** для представлення специфічного типу винятку, який генерується цим прикладом. Клас визначає два конструктори. Якщо поточний день тижня не збігається ні з вівторком, ні з четвергом, метод **doIt()** генерує **WrongDayException**. У цьому випадку об'єкт **WrongDayException** створюється у процесі генерації винятку. Це загальноприйнята практика. Оголошення методу

doIt() містить оператор **throws** для вказівки, що цей метод може генерувати **WrongDayException**.

У методі **main()** виклик **doIt()** поміщено в блок **try**, так що він може обробити **WrongDayException**, згенерований у **doIt()**. В **catch**-блоці здійснюється роздрук повідомлення про помилку із застосуванням методу **getMessage()**. Даний метод відтворює рядок, переданий конструктору при створенні об’єкту винятку.

2.6. Типи вбудованих виняткових ситуацій та їхня ієрархія

Всі виняткові ситуації – підкласи класу **Throwable**. **Exception** і **Error** – похідні від нього. До класу **Exeption** належать виняткові ситуації, які повинні перехоплюватися програмою користувача. Цей клас також застосовується для створення користувацьких типів ВС. Він має підклас **RuntimeException** (ділення на нуль, вихід за межі масиву тощо). ВС цього типу визначаються автоматично для всіх програм.

У класі **Error** описуються ВС, які, зазвичай, не перехоплюються програмою. Ці ВС відносять до помилок виконавчої системи Java (наприклад, переповнення стека). Вони являють собою перебої в системі, і їх неможливо обробити в програмі.

Всеможливі винятки в Java-програмі організовані в ієрархію класів винятків. Клас **Throwable** – безпосередній підклас **Object** – вершина ієрархії класів винятків. На рис. 2.1 наведено стандартні класи винятків.

Найбільш загальні ВС є підкласами стандартного класу **RuntimeException**. Немає необхідності поміщати їх у список **throws**. Такі ВС називаються неконтрольованими (*unchecked exception*), оскільки компілятор не перевіряє, чи ці ВС генеруються й обробляються в методі. В табл. 2.1 наведено список неконтрольованих вийнятків та їхній опис.

Виняткові ситуації, які необхідно поміщати у список **throws**, називаються контрольованими (*checked exception*). В табл. 2.2 наведено список контрольованих винятків Java та їхній опис.

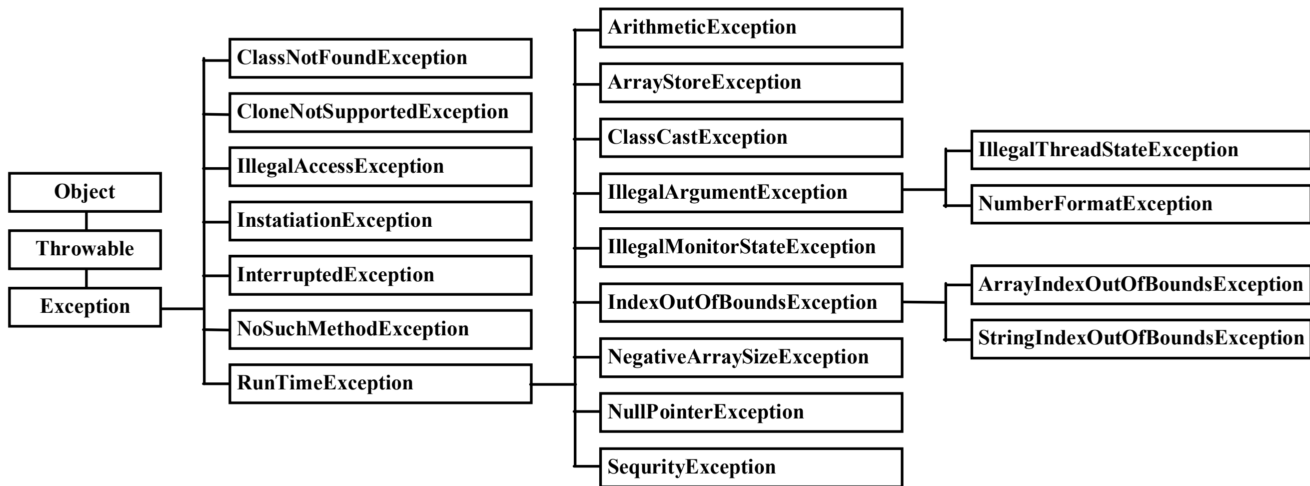

Рис. 2.1. Стандартні класи винятків Java

Таблиця 2.1. Неконтрольовані виняткові ситуації Java

Виняток	Опис
ArithmeticException	Арифметична помилка, наприклад, ділення на нуль
ArrayIndexOutOfBoundsException	Вихід індексу за межі масиву
ArrayStoreException	Спроба збереження в масиві елемента несумісного типу
ClassCastException	Недопустиме зведення типів
IllegalArgumentException	Методу переданий невідповідний аргумент
IllegalMonitorStateException	Методи **wait()**, **notify()** або **notifyAll()** об'єкта викликаються з потоку, який не є власником монітора цього об'єкта
IllegalStateException	Середовище або застосування перебувають у некоректному стані
IllegalThreadStateException	Операція несумісна з поточним станом потоку
IndexOutOfBoundsException	Підкласи цього класу генеруються, якщо індекс масиву чи рядка виходить за межі розміру
NegativeArraySizeException	Створення масиву від'ємного розміру
NullPointerException	Недопустиме використання посилання null
NumberFormatException	Недопустиме перетворення рядка в числовий формат
SecurityException	Спроба порушення захисту
StringIndexOutOfBoundsException	Вихід індексу за межі рядка
UnsupportedOperationException	Непідтримувана операція

Таблиця 2.2. Контрольовані виняткові ситуації Java

ClassNotFoundException	Клас не знайдено
CloneNotSupportedException	Спроба клонувати об'єкт, який не реалізує інтерфейс **Cloneable**
IllegalAccessException	Неправильний доступ
InstantiationException	Спроба створення екземпляра абстрактного класу або інтерфейсу
InterruptedException	Один потік був перерваний іншим потоком
NoSuchFieldException	Вказане поле не існує
NoSuchMethodException	Вказаний метод не існує

3. Багатопотокове програмування

В Java передбачена вбудована підтримка багатопотокового програмування (*multithreaded programming*). Багатопотокова програма містить декілька частин, які виконуються одночасно і називаються потоками. Потоки працюють незалежно один від одного. Застосування потоків дозволяє писати програми, які максимально використовують ресурси процесора. Це особливо важливо для інтерактивних і мережних середовищ, в яких використовується Java.

Підтримка багатопотокового програмування забезпечується класами **Thread**, **ThreadGroup** та інтерфейсом **Runnable** із пакету java.lang. В табл.3.1 наведено методи класу **Thread** та їх опис.

Під час запуску Java-програми створюється ***головний потік*** (*main thread*). Він створюється автоматично, але його роботою можна керувати через об'єкт **Thread**. Для цього необхідно отримати посилання на цей об'єкт за допомогою методу **currentThread()**:

```
Thread t=Thread.currentThread();
t.setName("MyThread");
System.println("Current thread is: " + t);
```

Група потоків – структура даних, яка контролює стан усіх потоків.

Головний потік відіграє важливу роль, оскільки породжує дочірні потоки і повинен завершуватися останнім. У випадку зупинки головного потоку програма завершує свою роботу.

3.1. Створення потоків

Є два способи створення потоків: реалізація так званого *виконавчого інтерфейсу* **Runnable** та розширення класу **Thread**.

У процесі реалізації інтерфейсу **Runnable** слід:

- створити клас, який реалізує інтерфейс **Runnable**. Цей інтерфейс містить єдиний метод **run()**, який необхідно перевизначити. У цьому разі створюється точка входу іншого, конкуруючого потоку, який завершується при завершенні методу **run()**;
- оголосити змінну, яка буде містити об'єкт класу **Thread**, за допомогою одного з конструкторів;
- викликати метод **run ()** для запуску потоку.

Таблиця 3.1. Методи класу **Thread**

Оголошення	Опис
Конструктори	
Thread()	Створює новий потік
Thread(Runnable target)	Створює новий потік, який використовує метод **run()** вказаного адресата **target**
Thread(ThreadGroup group, Runnable target)	Створює новий потік у групі потоків **group**, який використовує метод **run()** адресата **target**
Thread(String s)	Створює новий потік з іменем **s**
Thread(Runnable target, String s)	Створює новий потік з іменем **s** , який використовує метод **run()** вказаного адресата **target**
Thread(ThreadGroup group, String name)	Створює новий потік з іменем **s** в групі потоків **group**
Thread(ThreadGroup group, Runnable target, String name)	Створює потік з іменем **s** в групі потоків **group**, який використовує метод **run()** адресата **target**
Константи	
static final int MAX_PRIORITY=10	Найвищий пріоритет
static final int MIN_PRIORITY=1	Найнижчий пріоритет
static final int NORM_PRIORITY=5	Середній пріоритет
Методи класу	
int activeCount()	Повертає поточну кількість потоків у групі
Thread currentThread()	Повертає поточний об'єкт **Thread**
boolean interrupted()	Повертає **true**, якщо потік може бути перерваний
void sleep(long millis) throws InterruptedException	Переводить потік в стан очікування на **millis** мілісекунд
void yield()	Примушує потік поступитися процесором для іншого потоку

Методи екземпляру	
void checkAccess() throws SecurityException	Визначає, чи виконуваний потік має право модифікувати об'єкт **Thread**
final String getName()	Повертає ім'я потоку
final int getPriority()	Повертає пріоритет потоку
final ThreadGroup getThreadGroup()	Повертає групу потоків, до якої належить потік
void interrupt()	Перериває виконання потоку
final boolean isAlive()	Повертає **true**, якщо потік діючий
final boolean isDaemon()	Повертає **true**, якщо потік – демон
boolean isInterrupted()	Повертає **true**, якщо потік був перерваний
final synchronized void join(long millis) throws InterruptedException	Примушує потік, який його викликав, чекати завершення пов'язаного з цим методом об'єкта **Thread** протягом заданих **millis** мілісекунд
final void join() throws InterruptedException	Примушує потік, який його викликав, чекати завершення пов'язаного з цим методом об'єкта **Thread**
final void resume()	Відновлює роботу призупиненого потоку
void run()	Метод інтерфейсу **Runnable**. Запускається на виконання у процесі виклику методу **start()** потоку
final void setDaemon(boolean on)	Встановлює атрибути для потоку демону.
final void setName(String name)	Встановлює ім'я потоку
final void setPriority (int newPriority)	Встановлює пріоритет потоку
synchronized native void start()	Запускає потік на виконання
final void stop()	Зупиняє виконання потоку
final void suspend()	Призупиняє виконання потоку
String toString()	Повертає рядкове подання об'єкта **Thread**

Приклад використання інтерфейсу **Runnable**:

```
class TNewRunnable implements Runnable
{
  static final short kLimit = 1000;
  private short fLimit = 0;
  public void run()
  {
    while (fLimit++ < kLimit)
    {
      System.out.println("Виконується потік " +
      Thread.currentThread().toString());
      try
      {
         Thread.currentThread().sleep(10);
      } catch (InterruptedException e) {
        System.err.println("Очікування "+
              "дочірнього процесу перервано.");
        System.exit(1);
        }
     }
  }
}
public class TNewThreadDemo
{
  public static void main(String argv[])
  {
    TNewRunnable aNewRunnable =
                    new TNewRunnable();
    aNewRunnable.run();
  }
}
```

У процесі розширення класу **Thread** слід:

- створити клас, який розширює клас **Thread**. У розширеному класі необхідно перекрити метод **run**();
- оголосити змінну, яка буде містити об'єкт нового класу;
- викликати метод **start()** для запуску потоку.

Приклад використання класу **Thread**:

```
class TNewThread extends Thread{
  static final short kLimit = 1000;
  private short fLimit = 0;
public TNewThread(String aName) {
```

```
      super(aName);
   }
   public void run(){
      while (fLimit++ < kLimit)
      {
       System.out.println("Виконується потік " +
       Thread.currentThread().toString());
       try{
           Thread.currentThread().sleep(10);
           } catch (InterruptedException e) {
             System.err.println("Очікування "+
              "дочірнього процесу перервано");
              System.exit(1);}
      }
     }
   }
   public class TNewThreadDemo{
     public static void main(String argv[])  {
       TNewThread aNewThread =
               new TNewThread("Новий потік");
       aNewThread.start();
     }
   }
```

3.2. Атрибути потоків

Щоб ефективно використовувати потоки, необхідно розуміти їхні різноманітні аспекти й особливості роботи виконавчої системи Java, а саме:

- тіло потоку – як створити тіло потоку;
- стан потоку – життєвий цикл потоку;
- пріоритет потоку – як виконавча система планує виконання потоків;
- потоки-демони – як вони створюються;
- групи потоків – всі потоки повинні міститися в деякій групі потоків.

Тіло потоку

Реалізується в його методі **run()**. Тіло потоку можна створити шляхом породження похідного класу від **Thread** і перевизначення методу **run()**. Або ж створити клас, який реалізує інтерфейс **Runnable**. В цьому випадку необхідно створити об’єкт класу **Thread** і передати йому об’єкт виконавчого інтерфейсу адресата:

```
updateThread = new Thread(this);
```

Стан потоку

Потік може перебувати в одному із чотирьох нижче наведених станів.

Новий потік – стан, у який переходить потік під час створення екземпляру потоку

```
Thread myThread=new Thread(this);
```

Під час цього відбувається розподіл системних ресурсів. Наразі – це порожній об'єкт. Його можна запустити на виконання чи зупинити. Інші методи генерують виняткову ситуацію **`IllegalThreadStateException`**.

Виконуваний потік – стан, у який переходить потік під час виклику методу **`start()`**. Це означає, що процес може бути виконаний, якщо планувальник надасть йому час процесора. На цей момент він може і не виконуватися, але ніщо не заважає йому виконатися, тобто він не заблокований і не завершений.

Заблокований потік – стан, коли процес може бути запущеним, але не виконується. Допоки процес заблокований, планувальник просто пропускає його і не виділяє для нього час процесора. Цей стан можливий у таких випадках:

- потік був призупинений за допомогою методу **`suspend()`**. В цьому випадку він переходить у забуття. Повернути його у виконуваний стан можна за допомогою методу **`resume()`**. В Java 2 використання цих методів не рекомендується, оскільки метод **`suspend()`** захоплює блокування об'єкта і можлива ситуація взаємного блокування. Отже, можлива ситуація, коли декілька об'єктів очікують один одного, що викликає зависання програми;
- потік очікує деякий заданий проміжок часу (метод **`sleep()`**). Метод **`sleep()`** не звільняє блокування;
- потік призупинений за допомогою методу **`wait()`**. Вихід із цього стану здійснюється за допомогою методів **`notify()`**, **`notifyAll()`**, або якщо мине вказаний у методі **`wait()`** час очікування. Метод звільняє блокування, а тому під час очікування можуть викликатися інші синхронізовані методи цього об'єкта. Зазначимо, що метод **`wait()`** може бути викликаний тільки із синхронізованого методу. В іншому випадку буде згенерований виняток **`IllegalMonitorStateException`**;
- потік заблокований іншим потоком (наприклад, потоком, пов'язаним із операцією введення-виведення). У цьому випадку потік вважається невиконуваним, навіть якщо він повністю готовий до виконання.
- потік робить спробу викликати синхронізований метод іншого об'єкта, і блокування цього об'єкта неможливе.

Завершений потік – стан, в який переходить потік під час виклику методу **`stop()`** потоку, або у випадку завершенні методу **`run()`**.

Метод **`isAlive()`** дає змогу визначити стан потоку. Якщо він повертає **`true`**, то потік був створений і запущений. Наразі він може бути у стані виконання або заблокований. Якщо результат виклику методу **`false`** – то потік зупинений або незапущений на виконання.

Зауваження. Методи **`suspend()`**, **`resume()`** та **`stop()`** хоча і підтримуються в Java 2, проте застосовувати їх не рекомендують, оскільки вони можуть призвести до серйозних системних збоїв. Щоб призупинити, відновити чи зупинити виконання потоку, необхідно спроектувати його так, щоб метод **`run()`** періодично перевіряв, у який стан потік повинен перейти. Цього можна досягти залученням змінної прапорця, яка б вказувала на стан потоку.

Пріоритети потоків

Використовує диспетчер потоків для визначення моментів перемикання між потоками. Теоретично потік із вищим пріоритетом отримує більше часу процесора. Однак на практиці все залежить ще й від методу реалізації багатопотоковості в ОС. Крім цього, потоки з вищим пріоритетом можуть призупинити потоки з нижчим пріоритетом. Для потоків з однаковим пріоритетом усе залежить від реалізації багатопотоковості в ОС. Тому потоки з однаковим пріоритетом час від часу повинні віддавати управління головному потоку, щоб дати можливість запуску інших потоків в будь-якій ОС. Для цього застосовується метод **`yield()`**.

Пріоритети потоків встановлюються за допомогою методу **`setPriority()`** у межах **`MIN_PRIORITY`** і **`MAX_PRIORITY`**.

```
MyThread.setRriority(MAX_PRIORUTY);
```

Пріоритети потоків використовуються для визначення моменту перемикання між потоками за такими правилами:

- потік може передати управління із власної ініціативи. Це відбувається у процесі переходу до стану очікування чи блокування. Тоді опитуються інші потоки, готові до виконання, і потік з найвищим пріоритетом отримує процесор у своє розпорядження. Якщо два потоки мають однаковий пріоритет, то планувальник виконує їх за круговою схемою;
- потік може бути призупинений іншим потоком із вищим пріоритетом (пріоритетна схема з витісненням).

Для добровільного звільнення доступу до процесора застосовується метод **`yield()`**.

Потоки-демони

Це потоки, які підтримують інші потоки. В тілі потоку-демона часто міститься нескінченний цикл, в якому очікується запит від об'єкта чи іншого потоку. Коли приходить запит, потік-демон його обробляє. Щоб зробити потік демоном застосовується метод **`setDeamon(true)`**. Для визначення, чи є потік демоном, застосовуємо метод **`isDeamon()`**, який повертає **`true`**, якщо потік – демон.

Групи потоків

Всі потоки в Java повинні входити до деякої групи. Під час створення нового потоку можна вказати групу потоків, до якої ввійде створений потік. Для цього передбачено три конструктори класу **`Thread`**. Якщо під час створенні потоку не вказується група потоків, то створений потік ввійде до так званої групи ***main***. Групи потоків особливо корисні, оскільки дають змогу керувати цілою групою потоків, тобто призупиняти чи запускати всі потоки групи одночасно. Методи класу **`ThreadGroup`** наведено в табл. 3.2.

3.3. Синхронізація потоків

Коли двом чи більше потокам потрібен доступ до ресурсів, необхідно забезпечити доступ не більше ніж одного потоку в кожен момент часу. Процес, за допомогою якого це досягається, називається *синхронізацією*. Ключем до синхронізації є концепція *монітора*.

Монітор – це об'єкт, який використовується як взаємно-виключне блокування. Тільки один потік може володіти монітором у певний час. Коли потік блокується, то говорять, що він увійшов в монітор. Всі інші потоки, які захочуть увійти в заблокований монітор, будуть призупинені.

Java забезпечує синхронізацію на рівні мови. Є два способи синхронізації потоків: застосування методів синхронізації та синхронізуючий блок. В Java синхронізація програмується легко, оскільки всі об'єкти мають свої неявні монітори. Щоб зайти в монітор об'єкта, необхідно викликати метод, доповнений ключовим словом **`synchronized`**. Для виходу з монітора потік – власник монітора – просто повертається із синхронізованого методу.

Таблиця 3.2. Методи класу **ThreadGroup**

Оголошення	Опис
Конструктори	
ThreadGroup(String name)	Створює групу потоків із заданим іменем **name** у тій самій групі потоків, що й поточний потік
ThreadGroup(ThreadGroup parent, String name)	Створює групу потоків із заданим іменем **name** у вказаній батьківській **parent** групі потоків
Методи екземпляру	
int activeCount()	Повертає приблизну кількість потоків, які належать групі потоків та будь-яким її дочірнім групам потоків
int activeGroupCount()	Повертає приблизну кількість дочірніх груп потоків, які належать групі потоків
boolean allowThreadSuspension(boolean b)	Повертає **true**, якщо віртуальна машина Java дає змогу призупиняти потоки завдяки низькорівневій пам'яті
final void checkAccess()	Метод закінчується, якщо виконуваний у цей момент потік має дозвіл модифікувати групу потоків
final void destroy()	Руйнує групу потоків та будь-які дочірні групи потоків. У цьому випадку група потоків не повинна містити жодного потоку. Цей метод також вилучає групу потоків із його батьківської групи потоків. Генерує вийняткову ситуацію **IllegalThreadStateException**, якщо група потоків непорожня або вже знищена
int enumerate(Thread list[])	Зберігає посилання на активні потоки, що належать певній групі потоків або будь-яким дочірнім групам, у масив. Для визначення розміру масиву можна скористатись методом **activeCount()**. Повертає кількість потоків, які зберігаються у масиві потоків

int enumerate(Thread list[], boolean rec)	Аналогічний до попереднього методу. Змінна **rec** вказує, чи поміщати потоки із дочірніх груп у масив.
int enumerate(ThreadGroup list[])	Зберігає посилання на активні групи потоків, що належать певній групі потоків або будь-яким дочірнім групам, у масив. Для визначення розміру масиву використовують метод **activeGroupCount()**. Повертає кількість груп потоків, які зберігаються в масиві.
int enumerate(ThreadGroup list[], boolean rec)	Аналогічний до попереднього. Змінна **rec** вказує, чи поміщати групи потоки із дочірніх груп у масив.
final int getMaxPriority()	Повертає максимальне значення пріоритету, який може бути призначений потоку, що належить до групи потоків
final String getName()	Повертає ім’я групи потоків
final ThreadGroup getParent()	Повертає групу потоків – батьківську – для певної групи потоків. Якщо ця група потоків міститься у вершині групової ієрархії потоків, то метод повертає **null**
final boolean isDaemon()	Повертає **true**, якщо група потоків – демон
synchronized boolean isDestroyed()	Повертає **true**, якщо група потоків уже знищена
void list()	Виводить список потоків групи
final boolean parentOf (ThreadGroup g)	Повертає **true**, якщо група потоків є прямим чи непрямим предком вказаної групи **g** або збігається з нею
final void resume()	Відновлює виконання всіх потоків в певній групі потоків
final void setDaemon(boolean daem)	Змінює стан демона групи потоків.
final void setMaxPriority(int pri)	Встановлює максимальне значення пріоритету групі
final void stop()	Зупиняє всі потоки групи потоків та її дочірніх груп.
final void suspend()	Призупиняє всі потоки групи потоків та всіх її дочірніх груп потоків.
String toString()	Повертає рядкове подання групи потоків

Кожного разу, коли застосовується метод чи група методів, які обробляють внутрішній вміст об'єкта в багатопоточній ситуації, необхідно використовувати ключове слово **synchronized**. Як тільки потік входить у синхронізований метод деякого екземпляра, жоден інший потік не може ввійти в будь-який інший синхронізований метод цього екземпляра. Однак будь-який інший потік може отримати доступ до несинхронізованих методів цього екземпляра.

Розглянемо приклад. Програма здійснює правильний вивід завдяки тому, що метод **call** синхронізований. В іншому випадку вивід програми непрогнозований.

```
class Callme {
  synchronized void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    target.call(msg);
  }
}
public class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target,
                                "Synchronized");
    Caller ob3 = new Caller(target, "World");
```

```
    //очікування завершення потоків
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Перервано");
    }
  }
}
```

Вивід цієї програми такий:

```
[Hello]
[Synchronized]
[World]
```

Якщо ми хочемо синхронізувати доступ до об'єктів класу, який не розроблений для багатопотоковості, або не маємо доступу до коду класу, необхідно використовувати синхронізуючий блок із викликами методів класу:

synchronized (об'єкт){оператори, які необхідно синхронізувати}

де **об'єкт** – посилання на об'єкт, який синхронізуємо.

Синхронізуючий блок призначений для того, щоб виклик методів об'єкта класу відбувався тільки після того, як потік успішно увійшов у монітор об'єкта.

Передача інформації між потоками

У процесі синхронізації використовувалось безумовне блокування потоків від асинхронного доступу до деяких методів. Проте є можливість більш тонко керувати процесом з використанням механізму міжпотокових зв'язків.

Для взаємодії між потоками застосовуються методи **wait()**, **notify()** і **notifyAll()** класу **Object**. Ці методи можна викликати тільки із синхронізованих методів.

- **wait()** – наказує потоку, який викликав цей метод, віддати монітор і перейти в стан очікування, поки інший потік не ввійде в монітор і не викличе метод **notify()**.
- **notify()** – активізує перший потік, який викликав метод **wait()** на тому ж об'єкті.
- **notifyAll()** – активізує всі потоки, які викликали **wait()** одного і того ж об'єкта. Першим запускається потік з найвищим пріоритетом.

Існують перевантажені версії методу **`wait()`**, які дають змогу вказати максимальний період часу очікування.

Взаємне блокування

Помилка взаємного блокування виникає у випадку, коли два потоки мають циклічну залежність від пари синхронізованих методів. Нехай один потік входить у синхронізований метод об'єкта **`X`**, а інший – у синхронізований метод об'єкта **`У`**. Якщо потік у методі **`X`** захоче викликати будь-який синхронізований метод об'єкта **`У`**, то він буде заблокований. Якщо ж потік у методі об'єкта **`У`** захоче викликати синхронізований метод об'єкта **`X`**, то він чекатиме безконечно.

4. Потоки введення-виведення

Пакет java.io містить фундаментальні класи для підтримки операцій введення-виведення. Ці класи поділяються на такі базові групи:

- класи для зчитування інформації з потоку;
- класи потокового запису;
- класи маніпуляції із файлами;
- класи для серіалізації об'єктів.

Весь фундаментальний ввід-вивід Java базується на понятті потоків. Потік репрезентує потік даних або канал зв'язку. Java 1.0 підтримувала тільки байтові потоки. Починаючи з Java 1.1, підтримуються як байтові потоки, так і символьні.

Розглянемо всі групи класів.

4.1. Класи потокового введення

`InputStream` – абстрактний клас, який визначає методи для послідовного зчитування з потоку байтів. Java надає ряд підкласів класу **`InputStream`** для зчитування інформації з файлів, об'єктів **`StringBuffer`**, масиву байтів, та ін. Додатково підкласи **`InputStream`** можуть бути об'єднані для забезпечення додаткової логіки, наприклад зберігання поточного номера рядка або комбінації декількох джерел вхідної інформації в один логічний вхідний потік. Легко створити підклас класу **`InputStream`**, який визначає методи для читання з будь-якого виду джерела даних. На рис. 4.1 зображено ієрархію класів байтового введення. Зірочкою позначені класи, які підтримуються в Java 2, але використовувати їх не рекомендується.

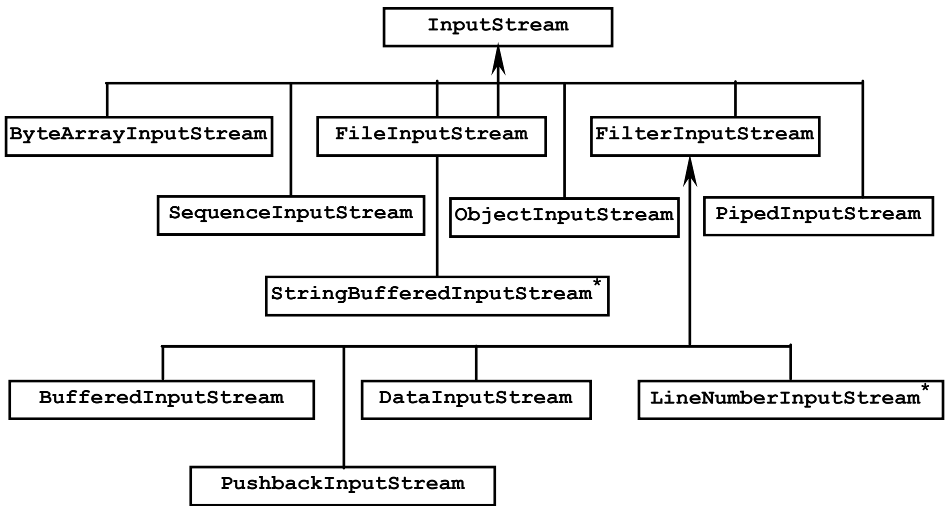

Рис. 4.1. Ієрархія класів байтового введення

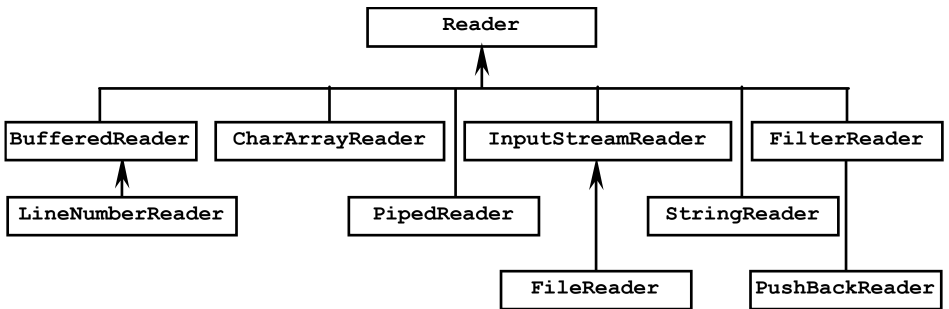

Рис. 4.2. Ієрархія класів символьного введення

Reader – абстрактний клас, який визначає методи для послідовного читання з потоку символів. Багато байт-орієнтованих підкласів класу **InputStream** мають символьно-орієнтовані аналоги – підкласи класу **Reader**. Зокрема, існують підкласи класу **Reader** для зчитування інформації з файлів, символьних масивів і рядкових об'єктів. На рис.4.2 зображено ієрархію класів символьного введення.

Розглянемо кожен із класів детальніше.

InputStream

Клас **InputStream** є абстрактним суперкласом для всіх інших класів байтового введення. Він визначає три методи **read()** для зчитування байтів з потоку:

read() – повертає цілочисельне подання введеного байта;

read(byte[] buff) – читає до **buff.length** байтів з потоку в буфер **buff** і повертає кількість успішно зчитаних байтів;

read(byte[] buff, int start, int num) – читає **num** байтів з потоку в буфер **buff**, починаючи з позиції **start**, і повертає кількість успішно зчитаних байтів.

Якщо немає даних, доступних для зчитування, ці методи блокуються допоки ввід не стане доступний. Клас також визначає метод **available()**, який повертає кількість байтів, доступних для читання, і метод **skip(long num)**, який пропускає зазначену кількість байтів. Клас **InputStream** визначає механізм для відмітки позиції у потоці та повернення до неї згодом, за допомогою методів **mark(int label)** і **reset()**. Метод **markSupported()** повертає **true**, якщо підклас підтримує ці методи. Метод **close()** закриває потік, після чого спроба читання з нього генерує **IOException**.

Оскільки клас **InputStream** – абстрактний, не можна створити його екземпляр. Для зчитування інформації використовуються різні підкласи класу **InputStream**.

Клас **InputStream** розроблений таким чином, що методи **read(byte[])** та **read(byte[],int,int)** викликають метод **read()**. Тому у випадку створення підкласу **InputStream**, необхідно визначити лише метод **read()**. Однак для більшої ефективності можна також перевизначити метод **read(byte[],int,int)**, який дає змогу прочитати блок даних ефективніше, ніж читання кожного байта окремо.

Reader

Клас **Reader** є абстрактним суперкласом для всіх інших класів символьного введення. Він визначає такі ж методи, як **InputStream**, за винятком того, що методи **read()** оперують символами, а не байтами:

read() – повертає цілочисельне подання введеного символу;

read(char[] buff) – читає до **buff.length** символів з потоку в буфер **buff** і повертає кількість успішно зчитаних символів;

read(char[] buff, int start, int num) – читає **num** символів з потоку в буфер **buff**. починаючи з позиції **start**, і повертає кількість успішно зчитаних символів.

Метод **available()**, який зустрічається в класі **InputStream**, замінений у **Reader** методом **ready()**. Цей метод повертає прапорець, який вказує, чи потік повинен блокуватися під час зчитування наступного символу.

Reader розроблений таким чином, що методи **read()** і **read(char[])** викликають метод **read(char[],int,int)**. Тому у випадку породження похідного класу від **Reader** необхідно визначити лише метод **read(char[],int,int)**. Зазначимо, що реалізація зчитування в класі **Reader** відрізняється від **InputStream** і, крім цього, ефективніша.

InputStreamReader

Клас **InputStreamReader** служить мостом між об'єктами **InputStream** і **Reader**. Хоча **InputStreamReader** діє подібно до символьних потоків, він отримує вхідні дані від байтового потоку, який лежить в його основі, і використовує схему кодування символів для переведення байтів у символи. У процесі створення **InputStreamReader**, вказують **InputStream**, що лежить в його основі і, за бажанням, ім'я схеми кодування. Наприклад, наступний фрагмент коду створює об'єкт **InputStreamReader**, який читає символи з файла, закодованого з використанням ISO 8859-5 кодування.

```
String fileName = "encodedfile.txt";
String encodingName = "8859_5";
InputStreamReader in;
try {
    FileInputStream fileIn =
            new FileInputStream(fileName);
in=new InputStreamReader(fileIn, encodingName);
} catch (UnsupportedEncodingException e1) {
  System.out.println(encodingName +
```

```
            " – непідтримувана схема кодування.");
    } catch (IOException e2) {
        System.out.println("Файл " + fileName +
            " не може бути відкритий.");
    }
```

FileInputStream і FileReader

Клас **FileInputStream** є похідним від **InputStream** і дає змогу читати потік байтів із файла. Цей клас не додає нових методів. Замість того, вказаний файл вважається відкритим, якщо створено відповідний об'єкт **FileInputStream**. Є три способи створення **FileInputStream**, а саме шляхом передавання конструктору:

- імені файла

```
FileInputStream f1 =
          new FileInputStream("foo.txt");
```

- об'єкта **File**:

```
File f = new File("foo.txt");
FileInputStream f2 = new FileInputStream(f);
```

- дескриптора файла (об'єкт **FileDescriptor**). **FileDescriptor** інкапсулює подання відкритого файла відповідної операційної системи. Об'єкт **FileDescriptor** можна отримати шляхом виклику методу **getFD()** класу **RandomAccessFile**. Отже, можна створити **FileInputStream**, який читає з відкритого файла, зв'язаного з об'єктом **RandomAccessFile**. Це, зокрема, ілюструють наступні рядки коду:

```
RandomAccessFile raf;
raf = new RandomAccessFile("z.txt","r");
FileInputStream f3 =
          new FileInputStream(raf.getFD());
```

Клас **FileInputStream** не підтримує методи **mark()** і **reset()**. Спроба їх застосування генерує виняткову ситуацію **IOException.**

Клас **FileReader** – підклас **Reader**, який читає потік символів із файла. Байти з файлу конвертуються до символів із застосуванням схеми кодування символів, встановленої за замовчуванням. Якщо ж ми хочемо використати іншу схему кодування, необхідно **FileInputStream** помістити в **InputStreamReader**, як зазначено вище. **FileReader** створюється шляхом передачі конструктору імені файла, об'єкта **File** або об'єкта **FileDescriptor**, аналогічно до випадку із **FileInputStream**.

StringReader і StringBufferInputStream

Клас **StringReader** – підклас класу **Reader**, який як джерело інформації використовує об'єкт **String**. Клас **StringReader** підтримує методи **mark()** та **reset()**. Наступний приклад ілюструє використання класу **StringReader**:

```
StringReader sr = new StringReader("abcdefg");
try {
    char[] buffer = new char[3];
    sr.read(buffer);
    System.out.println(buffer);
} catch (IOException e) {
    System.out.println("Зустрілася помилка " +
                       "під час читання.");
}
```

Цей код генерує такий вивід:

```
abc
```

Клас **StringBufferInputStream** – байт-орієнтований еквівалент **StringReader**. Зазначений клас вважається застарілим, починаючи з Java 1.1, оскільки відповідним чином не конвертує символи рядка до байтового потоку; він просто відкидає вісім старших бітів кожного символу. Хоча метод **markSupported()** класу **StringBufferInputStream** повертає **false**, метод **reset()** спонукає наступну операцію **read()** розпочати читання з початку рядка.

CharArrayReader і ByteArrayInputStream

Клас **CharArrayReader** – підклас **Reader** – як джерело використовує масив символів. Клас **CharArrayReader** підтримує методи **mark()** та **reset()**. **CharArrayReader** можна створити шляхом передачі конструктору посилання на масив символів, як зазначено нижче:

```
char[] c;
...
CharArrayReader r;
r = new CharArrayReader(c);
```

Також можна створити **CharArrayReader**, який читає тільки частину символьного масиву, шляхом передачі конструктору додаткових параметрів, які вказують початкову позицію, з якої необхідно розпочати читання, та кількість символів. Наприклад, щоб створити **CharArrayReader**, який читає елементи масиву, починаючи від 5 і до 24, необхідно записати:

```
r = new CharArrayReader(c, 5, 20);
```

Клас **ByteArrayInputStream** подібний до **CharArrayReader**, за винятком, що замість символів оперує байтами. В Java 1.0 **ByteArrayInputStream** не повністю підтримував методи **mark()** та **reset()**; починаючи з Java 1.1 ці методи повністю підтримуються.

PipedInputStream і PipedReader

Клас **PipedInputStream** – підклас **InputStream**, який забезпечує зв'язок між потоками. Оскільки він зчитує байти, записані зв'язаним із ним **PipedOutputStream**, **PipedInputStream** повинен бути з'єднаним з **PipedOutputStream**. Є декілька шляхів асоціювати **PipedInputStream** об'єкту **PipedOutputStream**. Спершу створюємо об'єкт **PipedOutputStream** і передаємо його конструктору класу **PipedInputStream**:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream(po);
```

Можна також спочатку створити об'єкт **PipedInputStream** і передати його конструктору класу **PipedOutputStream**:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po =
          new PipedOutputStream(pi);
```

Крім цього, обидва класи **PipedInputStream** та **PipedOutputStream** мають метод **connect()**, який можна застосувати для явного з'єднання **PipedInputStream** та **PipedOutputStream**, як зазначено нижче:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
pi.connect(po);
```

Або можна викликати метод **connect()** так:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
po.connect(pi);
```

Декілька об'єктів **PipedOutputStream** можуть з'єднуватися з єдиним **PipedInputStream** одночасно, але результат непередбачуваний. Якщо ми під'єднуємо **PipedOutputStream** до вже під'єднаного **PipedInputStream**, будь-які незчитані байти попередньо зв'язаного **PipedOutputStream** будуть втрачені. Отже, якщо два об'єкти **PipedOutputStream** з'єднані, **PipedInputStream** читає байти, записані одним із двох **PipedOutputStream**, у тому порядку, в якому вони отримуються ними. Послідовність пов'язаних потоків може

змінюватися від одного запуску програми до іншого, так що порядок, в якому **PipedInputStream** отримує байти від декількох об'єктів **PipedOutputStream**, може бути нестійким.

Клас **PipedReader** – символьний еквівалент **PipedInputStream**. Він працює так само, за винятком, що **PipedReader** під'єднується до **PipedWriter** каналу, з використанням відповідного конструктора або методу **connect()**.

FilterInputStream і FilterReader

FilterInputStream – абстрактний клас-оболонка для об'єктів **InputStream**, який забезпечує додаткові функціональні можливості. Концептуально об'єкт похідного від **FilterInputStream** класу огортає об'єкт класу **InputStream**. Констуктори фільтрованих класів вимагають як параметр об'єкт **InputStream**.

Усі методи **FilterInputStream** працюють шляхом виклику відповідних методів суперкласу **InputStream**. Оскільки метод **close()** **FilterInputStream** викликає метод **close()** класу **InputStream**, немає необхідності явно закривати потік **InputStream**.

Клас **FilterInputStream** не надає додаткової функціональності, а тому його об'єкти самі по собі нечасто використовуються. Однак підкласи **FilterInputStream** надають додаткову функціональність об'єктам, які вони огортають двома способами:

- деякі підкласи змінюють логіку методів класу **InputStream.** Наприклад, клас **InflaterInputStream** з пакету java.util.zip автоматично розпаковує дані у процесі зчитування їх методом **read()**;
- деякі підкласи додають нові методи. Наприклад, клас **DataInputStream** надає методи для зчитування з потоку простих типів даних Java.

FilterReader – символьно-орієнтований еквівалент **FilterInputStream. FilterReader** огортає об'єкт **Reader**, що лежить в його основі. Методи **FilterReader** викликають відповідні методи **Reader**. Однак, подібно до **FilterInputStream**, **FilterReader** – абстрактний клас, тому неможливо безпосередньо створити його екземпляр.

DataInputStream

DataInputStream – підклас класу **FilterInputStream**, який надає методи для читання різних типів даних. **DataInputStream**

реалізує інтерфейс **DataInput**, а отже він визначає методи для читання всіх простих типів Java.

Об'єкт **DataInputStream** створюється шляхом передачі його конструктору посилання на об'єкт **InputStream**. Нижче наведено приклад, в якому створений **DataInputStream** використовується для зчитування цілого, яке містить довжину масиву, а опісля – для зчитування масиву значень типу **long**:

```
long[] readLongArray(InputStream in)
                            throws IOException {
    DataInputStream din =
                    new DataInputStream(in);
    int count = din.readInt();
    long[] a = new long[count];
    for (int i = 0; i < count; i++) {
        a[i] = din.readLong();
    }
    return a;
}
```

BufferedReader і BufferedInputStream

BufferedReader – підклас класу **Reader**, який буферизує ввід із об'єкта **Reader**, що лежить в його основі. **BufferedReader** зчитує достатню кількість символів із об'єкта **Reader**, щоб заповнити відносно великий буфер, і опісля виконує операцію зчитування отриманих символів, які вже перебувають у буфері. Якщо більшість операцій читання зчитує тільки декілька символів, застосування **BufferedReader** може поліпшити ефективність, оскільки зменшує кількість операцій зчитування, які програма вимагає від операційної системи. В загальному випадку існує верхня межа, пов'язана із кожним звертанням до операційної системи. Таким чином зменшення кількості звертань до операційної системи поліпшує ефективність. Клас **BufferedReader** підтримує функціональність методів **mark()** та **reset()**. Нижче наведено приклад, який показує, яким чином створити **BufferedReader** для поліпшення ефективності зчитування із файла:

```
try {
    FileReader fileIn =
            new FileReader("data.dat");
    BufferedReader in =
            new BufferedReader(fileIn);
...    // читання з файла
} catch (IOException e) {
```

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

Клас **BufferedInputStream** – байтово-орієнтований аналог **BufferedReader**. Він працює так само, як **BufferedReader**, за винятком, що він буферизує ввід із об'єкта **InputStream**, що лежить в його основі.

LineNumberReader і LineNumberInputStream

Клас **LineNumberReader** – підклас класу **BufferedReader**. Його метод **read()** містить додаткову можливість лічити символи кінця рядка, а отже, зберігає номер рядка. Оскільки різні платформи використовують різні символи для подання кінця рядка, **LineNumberReader** надає гнучкий підхід і визначає "\n", "\r", або "\r\n" як кінець рядка. Незважаючи на те, який символ кінця рядка читається, **LineNumberReader** повертає тільки "\n" із свого методу **read()**.

LineNumberReader можна створити шляхом передачі його конструктору об'єкта **Reader**. В наступному прикладі видруковано п'ять перших рядків файла, кожен з яких починається своїм номером. Коли виконати цей приклад, то можна побачити, що номери рядків починаються за замовчуванням від нуля:

```
    try {
        FileReader fileIn =
                new FileReader("text.txt");
        LineNumberReader in =
                new LineNumberReader(fileIn);
        for (int i = 0; i < 5; i++)
            System.out.println(in.getLineNumber() +
                 " " + in.readLine());
    }catch (IOException e) {
        System.out.println(e);
    }
```

Клас **LineNumberReader** має два методи для операцій із номерами рядків. Метод **getLineNumber()** повертає поточний номер рядка. Щоб змінити поточний номер рядка, застосовують метод **setLineNumber()**. Цей метод не впливає на позицію потоку; він тільки визначає значення номера рядка.

LineNumberInputStream – байтово-орієнтований еквівалент **LineNumberReader**. Починаючи з Java 1.1, цей клас не рекомендується використовувати, оскільки він належним чином не конвертує байти до символів. Незважаючи на проблеми конвертації, **LineNumberInputStream** працює аналогічно до

LineNumberReader, за винятком, що він отримує свій ввід із об'єкта **InputStream**.

SequenceInputStream

Клас **SequenceInputStream** застосовується для об'єднання разом декількох об'єктів **InputStream**. Розглянемо приклад:

```
FileInputStream f1 =
            new FileInputStream("data1.dat");
FileInputStream f2 =
            new FileInputStream("data2.dat");
SequenceInputStream s =
            new SequenceInputStream(f1, f2);
```

В цьому прикладі створюється **SequenceInputStream**, який зчитує всі байти із **f1**, а опісля – всі байти із **f2**. Можна також вкладати об'єкти **SequenceInputStream** один в одний, щоб дати змогу читати як один більше ніж два вхідних потоки. Це можна записати так:

```
FileInputStream f3 =
            new FileInputStream("data3.dat");
SequenceInputStream s2 =
            new SequenceInputStream(s, f3);
```

Клас **SequenceInputStream** надає ще один конструктор, який може бути використаний для об'єднання разом більше ніж двох об'єктів **InputStream**. Він приймає як аргумент **Enumeration** об'єктів **InputStream**. Наступний приклад ілюструє, яким чином в цьому випадку створити **SequenceInputStream**:

```
Vector v = new Vector();
v.add(new FileInputStream("data1.dat"));
v.add(new FileInputStream("data2.dat"));
v.add(new FileInputStream("data3.dat"));
Enumeration e = v.elements();
SequenceInputStream s =
            new SequenceInputStream(e);
```

PushbackInputStream і PushbackReader

Клас **PushbackInputStream** – фільтрований потік, який дає змогу повернути дані назад у вхідний потік та прочитати їх наступною операцією зчитування. Ця функціональність часто використовується для аналізу даних, коли необхідно зчитати дане й опісля повернути його у вхідний потік. **PushbackInputStream** підтримує як однобайтовий буфер повернення, так і буфер повернення довільного розміру.

Під час створення **PushbackInputStream** його конструктору передають об'єкт **InputStream**:

```
FileInputStream ef =
      new FileInputStream("expr.txt");
PushbackInputStream pb =
      new PushbackInputStream(ef);
```

Цей конструктор створює **PushbackInputStream**, який використовує за замовченням однобайтовий буфер повернення.

```
FileInputStream ef =
      new FileInputStream("expr.txt");
PushbackInputStream pb =
      new PushbackInputStream(ef, 10);
```

У цьому випадку створено потік із буфером повернення 10 байтів.

Якщо є дане, яке ви хочете повернути у вхідний потік для зчитування наступною операцією читання, передайте його одному із методів **unread()**:

unread(int ch) – повертає в потік молодший байт параметра **ch**;

unread(byte buff[]) – повертає в потік групу байтів з буфера **buff**;

unread(byte buff[], int start, int num) – повертає в потік **num** байтів буфера **buff**, починаючи з позиції **start**.

Клас **PushbackReader** – символьно-орієнтований еквівалент **PushbackInputStream**. Нижче ілюструється створення **PushbackReader** з буфером повернення 48 символів:

```
FileReader fileIn = new FileReader("expr.txt");
PushbackReader in =
        new PushbackReader(fileIn, 48);
```

Приклад використання **PushbackReader**:

```
public String readDigits(PushbackReader pb) {
    char c;
    StringBuffer buffer = new StringBuffer();
    try {
        while (true) {
            c = (char)pb.read();
            if (!Character.isDigit(c))
                break;
            buffer.append(c);
        }
        if (c != -1)
            pb.unread(c);
    }catch (IOException e) {}
```

```
        return buffer.toString();
    }
```

Наведений вище приклад ілюструє застосування методів, які зчитують із **PushbackReader** символи, що відповідають цифрам. Якщо зчитується символ, що відрізняється від цифри, викликається метод **unread()**, таким чином нечислове значення може бути прочитане наступною операцією читання. Метод повертає рядок, який містить цифри, зчитані із потоку.

4.2. Класи потокового виведення

OutputStream – абстрактний клас, який визначає методи для послідовного запису байтів у потік. Java надає підкласи класу **OutputStream** для запису у файл, масив байтів та ін. Інші підкласи **OutputStream** можуть бути зв'язані разом для надання додаткової логіки, такої як запис багатобайтових типів даних або конвертування даних у рядкове подання. Також неважко визначити підклас **OutputStream**, який записує в інший вид призначення. На рис. 4.3 зображено ієрархію класів байтового потокового виведення.

Writer – абстрактний клас, який визначає методи для послідовного запису символів у потік. Більшість байтово-орієнтованих підкласів класу **OutputStream** мають символьно-орієнтовані аналоги – підкласи класу **Writer**. Зокрема, існують класи для запису у файл та масив символів. На рис. 4.4 зображено ієрархію класів символьного потокового виведення.

OutputStream

OutputStream – абстрактний суперклас для всіх інших класів байтового потокового виведення. Він визначає три методи **write()** для запису байтів у потік:

write(int b) – записує один байт у вихідний потік;

write(byte[] buff) – записує заповнений масив байтів **buff** у вихідний потік;

write(byte[] buff,int start,int num) – записує **num** байтів масиву **buff** у вихідний потік, починаючи з позиції **start.**

Деякі підкласи **OutputStream** можуть реалізувати буферизацію для підвищення ефективності. **OutputStream** надає метод **flush()**, який спонукає **OutputStream** записати будь-який буферизований вивід на відповідний пристрій, такий як диск чи сокет.

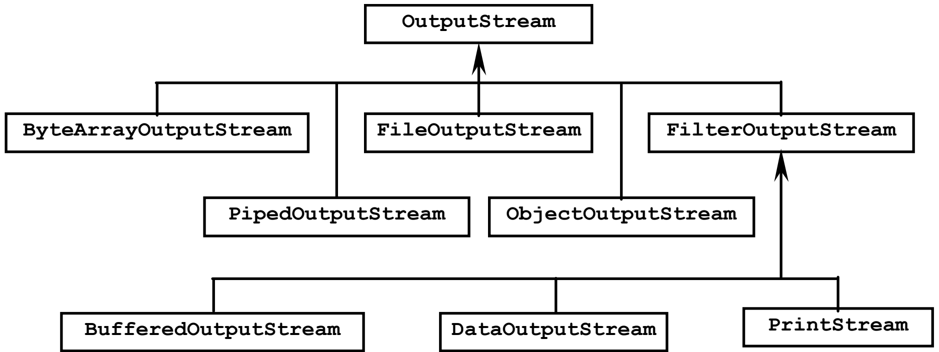

Рис.4.3. Ієрархія класів байтового виведення

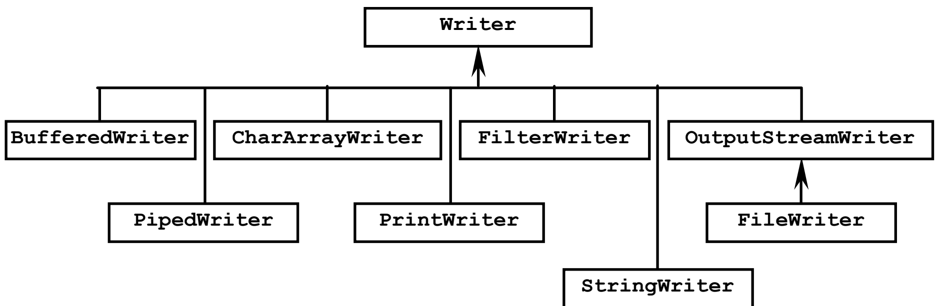

Рис.4.4. Ієрархія класів символьного виведення

Оскільки **OutputStream** – абстрактний, не можна створити об'єкт **OutputStream**. Однак різні підкласи **OutputStream** можуть використовуватися рівнозначно. Наприклад, методи часто приймають параметри типу **OutputStream**. Це означає, що ці методи як аргументи приймають будь-які підкласи **OutputStream**.

OutputStream розроблений таким чином, що методи **write(byte[])** та **write(byte[],int,int)** викликають метод **write(int)**. Тому у процесі створення підкласу класу **OutputStream** достатньо визначити метод **write()**. Однак для більшої ефективності перевизначають **write(byte[],int,int)**, який дає змогу записувати блок даних ефективніше, ніж запис кожного байта окремо.

Writer

Writer – абстрактний суперклас для всіх інших класів символьного потокового виведення. Він визначає методи, аналогічні до класу **OutputStream**, з тією відмінністю, що метод **write()** оперує символами замість байтів.

write(int c) – записує один символ у вихідний потік;
write(char[] buff) – записує заповнений масив символів **buff** у вихідний потік;
write(char[] buff,int start,int num) – записує **num** символів масиву **buff** у вихідний потік, починаючи з позиції **start**;
write(String str) – записує рядок **str** у вихідний потік;
write(String str,int start,int num) – записує підрядок **str** із **num** символів у вихідний потік, починаючи з позиції **start**.

Writer також містить метод **flush()**, який примушує будь-які буферизовані дані записатися в потік.

Writer розроблений так, що методи **write(int)** і **write(char[])** викликають метод **write(char[],int,int)**. Тому під час створення підкласу класу **Writer** достатньо перевизначити метод **write(char[],int,int)**. Зауважимо, що реалізація операції запису в класі **Writer** відрізняється від **OutputStream** і, крім цього, ефективніша.

OutputStreamWriter

OutputStreamWriter служить для зв'язку між об'єктами класів **Writer** та **OutputStream**. Незважаючи на те, що **OutputStreamWriter** працює подібно до символьних потоків, він конвертує символи у байти, застосовуючи схему кодування символів, та записує їх до об'єкта **OutputStream**. Цей клас – вихідний аналог

InputStreamReader. У процесі створення **OutputStreamWriter** як параметр вказують об'єкт **OutputStream** та, за бажанням, схему кодування. Приклад ілюструє процес створення об'єкта **OutputStreamWriter**, який записує символи у файл, застосовуючи схему кодування ISO 8859-5:

```
String fileName = "encodedfile.txt";
String encodingName = "8859_5";
OutputStreamWriter out;
try {
FileOutputStream fileOut =
          new FileOutputStream (fileName);
out =
new OutputStreamWriter (fileOut, encodingName);
} catch (UnsupportedEncodingException e1) {
System.out.println(encodingName +
  " - непідтримувана схема кодування.");
} catch (IOException e2) {
    System.out.println("Файл " + fileName +
                  "не може бути відкритий.");
}
```

FileWriter і FileOutputStream

FileOutputStream – підклас класу **OutputStream**, який застосовується у процесі запису байтів до файла. **FileOutputStream** не додає нових методів. Вважається, що файл безумовно відкритий, якщо створений об'єкт **FileOutputStream**. Є декілька способів створення **FileOutputStream**, а саме шляхом передачі:

- імені файлу для запису:

```
FileOutputStream f1 =
          new FileOutputStream("foo.txt");
```

- імені файлу для запису, додатково вказавши режим відкриття файлу (додавання або заміни). Приклад створення об'єкта **FileOutputStream**, який дає змогу дописувати дані до вказаного файла:

```
FileOutputStream f1 =
  new FileOutputStream("foo.txt", true);
```

- об'єкта **File**:

```
File f = new File("foo.txt");
FileOutputStream f2 = new FileOutputStream(f);
```

- дескриптора файла – об'єкта **FileDescriptor**. Об'єкт **FileDescriptor** інкапсулює подання відкритого файла

відповідної операційної системи. Отримати **FileDescriptor** можна за допомогою методу **getFD()** об'єкта **RandomAccessFile**. Нижче наведено процес створення **FileOutputStream**, який записує до відкритого файла, пов'язаного з об'єктом **RandomAccessFile**:

```
RandomAccessFile raf;
raf = new RandomAccessFile("z.txt","rw");
FileInputStream f3 =
            new FileOutputStream(raf.getFD());
```

Клас **FileWriter** – підклас **Writer**, який записує потік символів до файла. Символи, які записуються у файл, конвертуються до байтів із застосуванням схеми кодування символів, встановленої за замовчуванням. Якщо потрібно застосовувати схему кодування, що відрізняється від заданої за замовчуванням, необхідно огорнути **FileWriter** об'єктом **OutputStreamWriter**. Створити об'єкт **FileWriter** можна із використанням імені файла, об'єкта **File** або **FileDescriptor**, як описано вище для **FileOutputStream**.

StringWriter

StringWriter – підклас **Writer**, який записує дані у рядкой об'єкт. Для внутрішнього подання він використовує **StringBuffer**, який можна дослідити із застосуванням методу **getBuffer()**. Рядок, який містить записані дані, може бути отриманий за допомогою методу **toString()**. Приклад створення об'єкта **StringWriter**, у який записуються дані:

```
StringWriter out = new StringWriter();
char[] buffer = {'b', 'o', 'o', '!', 'h', 'a'};
out.write('B');
out.write("uga");
out.write(buffer, 0, 4);
System.out.println(out.toString());
```

Вивід такий:

```
Bugaboo!
```

CharArrayWriter і ByteArrayOutputStream

CharArrayWriter – підклас **Writer**, який записує символи у символьний масив. Є три способи відтворення даних, записаних до **CharArrayWriter**:

- метод **toCharArray()** повертає посилання на екземпляр внутрішнього масиву;

- метод **`toString()`** повертає **`String`**, створений із внутрішнього масиву;
- метод **`writeTo()`** записує внутрішній масив до іншого об'єкта **`Writer`**.

Приклад, який демонструє створення **`CharArrayWriter`**, запис у нього даних та їх відтворення:

```
CharArrayWriter out = new CharArrayWriter();
try {
    out.write("Daphne");
}catch (IOException e) {}
char[] buffer = out.toCharArray();
System.out.println(buffer);
String result = out.toString();
System.out.println(result);
```

Вивід такий:

```
Daphne
Daphne
```

Внутрішній буфер **`CharArrayWriter`** у процесі запису даних за необхідності розширюється. Якщо кількість символів заздалегідь відома, то можна під час створення **`CharArrayWriter`** вказати початковий розмір буфера.

`ByteArrayOutputStream` – байтово-орієнтований еквівалент **`CharArrayWriter`**. Він функціонує аналогічно, за винятком:

- метод **`write()`** оперує байтами, а не символами. Крім цього, **`ByteArrayOutputStream`** не має методу **`write(String)`**, визначеного в **`CharArrayWriter`**;
- замість **`toCharArray()`**, **`ByteArrayOutputStream`** має метод **`toByteArray()`**;
- має три методи **`toString()`**. Один, без аргументів, конвертує байти у внутрішній масив символів, застосовуючи схему кодування за замовчуванням. Метод **`toString(int)`** вважається застарілим, починаючи з Java 1.1, оскільки відповідним чином не конвертує байти до символів. Натомість використовують метод **`toString(String)`**, який коректно конвертує масив байтів у символьний рядок.

PipedOutputStream і PipedWriter

`PipedOuputStream` – підклас **`OutputStream`**, який забезпечує зв'язок між потоками. Для його використання необхідно створити **`PipedInputStream`**, оскільки **`PipedOutputStream`** записує байти,

які будуть зчитані відповідним, пов'язаним із ним, **PipedInputStream**. Є декілька способів об'єднання **PipedOutputStream** і **PipedInputStream**. Спершу слід створити **PipedInputStream** та передати його конструктору **PipedOutputStream**:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po =
          new PipedOutputStream(pi);
```

Можна також створити спершу об'єкт **PipedOutputStream** та передати його конструктору **PipedInputStream**:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream(po);
```

Крім цього **PipedOutputStream** та **PipedInputStream** мають метод **connect()**, який можна застосувати для явного з'єднання **PipedOutputStream** та **PipedInputStream**:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream();
po.connect(pi);
```

Або **PipedInputStream** з **PipedOutputStream** як показано нижче:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream();
pi.connect(po);
```

Тільки один **PipedInputStream** може бути під'єднаний до **PipedOutputStream** одночасно. Якщо використовується **connect()** для з'єднання **PipedOutputStream** з уже зв'язаним **PipedInputStream**, будь-які не прочитані байти попереднього з'єднання будуть втрачені.

PipedWriter – символьний еквівалент **PipedOutputStream**. Він працює аналогічно, за винятком, що **PipedWriter** з'єднується з **PipedReader** в єдиний канал, використовуючи відповідний конструктор або метод **connect()**.

FilterOutputStream і FilterWriter

Клас **FilterOutputStream** – оболонка для об'єктів **OutputStream**. Концептуально об'єкти, які є підкласами **FilterOutputStream**, огортають інші об'єкти **OutputStream**. Конструктор цього класу потребує об'єкта **OutputStream**.

Усі методи **FilterOutputStream** виконуються шляхом виклику відповідних методів **OutputStream**. Оскільки метод **close()** класу **FilterOutputStream** викликає метод **close()** об'єкта

OutputStream, який він огортає, немає необхідності явно закривати потік **OutputStream**.

Клас **FilterOutputStream** не додає жодної функціональності об'єктам, які він огортає, так що сам по собі він не часто застосовується. Однак підкласи класу **FilterOutputStream** додають функціональності об'єктам, які вони огортають, двома способами:

- деякі підкласи додають логіки до методів **OutputStream**. Наприклад, **BufferedOutputStream** дає змогу буферизувати операцію запису;
- інші підкласи додають нові методи. Наприклад, **DataOutputStream** забезпечує методами для запису простих типів Java до потоку.

FilterWriter – символьний еквівалент **FilterOutputStream**. Він огортає об'єкт **Writer**, що лежить в його основі. Методи **FilterWriter** викликають відповідні методи об'єкта **Writer**. Однак, на відміну від **FilterOutputStream**, **FilterWriter** є абстрактним класом, тому неможливо безпосередньо створити його екземпляр.

DataOutputStream

DataOutputStream – підклас **FilterOutputStream**, який надає методи для запису різноманітних типів даних до **OutputStream**. **DataOutputStream** реалізує інтерфейс **DataOutput**, тому він визначає методи для запису всіх простих типів даних.

DataOutputStream створюється шляхом передачі посилання на об'єкт **OutputStream**. Нижче створюється **DataOutputStream**, який спочатку використовується для запису довжини масиву як значення типу **int**, а згодом – запису значень типу **long** у масив:

```
void writeLongArray(OutputStream out, long[] a)
                          throws IOException {
DataOutputStream dout =
          new DataOutputStream(out);
dout.writeInt(a.length);
for (int i = 0; i < a.length; i++) {
     dout.writeLong(a[i]);
   }
}
```

BufferedWriter і BufferedOutputStream

BufferedWriter – підклас класу **Writer**, який зберігає вивід у внутрішньому буфері. Коли буфер заповнений, цілий буфер записується

або скидається до об’єкта **Writer**, що лежить в його основі. Використання **BufferedWriter** збільшує швидкість запису порівнянно з **Writer**, оскільки зменшує кількість звертань до записуючого пристрою, чи це диску, чи мережі. Можна застосовувати метод **flush()** для примусового запису вмісту буфера до **Writer**.

Розглянемо приклад створення **BufferedWriter** навколо мережного сокетного вихідного потоку:

```
public Writer getBufferedWriter(Socket s)
                          throws IOException {
OutputStreamWriter converter =
  new OutputStreamWriter(s.getOutputStream());
    return new BufferedWriter(converter);
}
```

Спершу створюємо **OutputStreamWriter**, який конвертує символи до байтів застосовуючи схему кодування за замовчуванням. Опісля байти записуються до сокета. Потім просто огортаємо **BufferedWriter** навколо **OutputStreamWriter** для буферизації виводу.

BufferedOutputStream – байтово-орієнтований еквівалент класу **BufferedWriter**. Він функціонує аналогічно до **BufferedWriter**, за винятком, що він буферизує вивід для класу **OutputStream**. Приклад створення **BufferedOutputStream**, який використовується із сокетом:

```
public OutputStream
          getBufferedOutputStream(Socket s)
                          throws IOException {
    return
new BufferedOutputStream(s.getOutputStream());
}
```

PrintWriter і PrintStream

Клас **PrintWriter** – підклас **Writer**, який надає ряд методів для друку рядкового подання всіх типів даних Java. **PrintWriter** може огортати об’єкти класів **Writer** або **OutputStream**. У випадку використання **OutputStream**, будь-які записані до **PrintWriter** символи конвертуються до байтів із застосуванням схеми кодування за замовчуванням. Додатково конструктор дає змогу задати, яким чином потік буде скидатися після запису в нього символів нового рядка.

Клас **PrintWriter** надає методи **print()** та **println()** для виводу всіх простих типів Java. Метод **println()** діє аналогічно до

print() і, крім цього, додає символ нового рядка. Нижче демонструється використання **PrintWriter**, який огортає **OutputStream**:

```
boolean b = true;
char c = '%'
double d = 8.31451
int i = 42;
String s = "R = ";
PrintWriter out =
          new PrintWriter(System.out, true);
out.print(s);
out.print(d);
out.println();
out.println(b);
out.println(c);
out.println(i);
```

Вивід такий:

```
R = 8.31451
true
%
42
```

4.3. Управління файлами

Потоки використовуються для обробки операцій введення-виведення в Java. Крім цього, в пакеті java.io міститься декілька непотокових класів, які забезпечують управління файлами. Клас **File** подає файл в локальній файловій системі, клас **RandomAccessFile** забезпечує непослідовний доступ до даних файла. Крім цього, інтерфейс **FilenameFilter** використовується для фільтрації списку файлів.

File

Клас **File** репрезентує файл у локальній файловій системі. Екземпляр класу **File** можна використовувати для ідентифікації файла, отримання інформації про файл, і навіть змінення інформації про файл. Найпростіше створити об’єкт **File** шляхом передачі конструктору імені файла:

```
File myFile = new File("readme.txt");
```

Хоча методи, які надає **File** для маніпулювання інформацією про файл, є незалежні від платформи, ім’я файла повинне відповідати правилам локального системного іменування файлів. Клас **File** надає деяку інформацію, корисну для інтерпретації імені файла та директорії.

Змінна **separatorChar** визначає системно-визначений символ, який використовується для розділення імен директорії. Зокрема, в операційній системі Windows використовується обернений слеш (\), тоді як у UNIX чи Macintosh – прямий слеш (/). Наприклад, створимо об'єкт **File**, який вказує на файл readme.txt у директорії myDir:

```
File myFile = new File("myDir" +
          File.separatorChar + "readme.txt");
```

Клас **File** також надає декілька конструкторів, які полегшують створення об'єктів. Зокрема, існує конструктор **File**, який приймає як параметри два рядки: перший – ім'я директорії, другий – ім'я файла. Наприклад:

```
File myFile = new File("myDir", "readme.txt");
```

Клас **File** має ще один конструктор, який дає змогу визначити директорію файла, використовуючи об'єкт **File** замість **String**:

```
File dir = new File("myDir");
File f = new File(dir, "readme.txt");
```

Іноді програмі необхідно обробити список файлів, які передаються їй у вигляді рядка. Наприклад, такий список файлів, що передаються оточенню Java змінною оточення CLASSPATH, і який може бути отриманий у виразі:

```
System.getProperty("java.class.path");
```

Цей список містить одне чи більше імен файлів, розділених символами-роздільниками. В операційній системі Windows чи Macintosh символом-роздільником є кома з крапкою (;), тоді як в UNIX – двокрапка (:). Системно-визначений символ-роздільник визначається змінною **pathSeparatorChar**. Зокрема, щоб отримати значення CLASSPATH у вигляді колекції об'єктів **File**, слід записати:

```
StringTokenizer s;
Vector v = new Vector();
s = new StringTokenizer(System.getProperty
  ("java.class.path"), File.pathSeparator);
while (s.hasMoreTokens())
    v.addElement(new File(s.nextToken()));
```

Можна відтворити повний шлях файла, поданого об'єктом **File**, за допомогою методу **getPath()**, ім'я файла без решти інформації про шлях – за допомогою методу **getName()**, ім'я директорії – за допомогою методу **getParent()**.

Клас **File** також визначає методи, які повертають інформацію про реальний файл, поданий об'єктом **File**. Метод **exists()** застосовується для перевірки, чи файл існує. **isDirectory()** та

isFile() дають відповідь, чи певний об’єкт є файлом, чи директорією. Якщо об’єкт – директорія, можна скористатись методом **list()** для отримання списку імен файлів, що містяться у цій директорії. Методи **canRead()** та **canWrite()** дають змогу визначити, чи дозволено програмі читати або, відповідно, записувати інформацію до файлу. Можна також знайти розмір файла за допомогою методу **length()** та дату останньої модифікації за допомогою методу **lastModified()**.

Декілька методів **File** дають змогу змінити інформацію про файл. Наприклад, можна перейменувати файл (метод **rename()**) або знищити (метод **delete()**). Методи **mkdir()** та **mkdirs()** надають спосіб створення директорій в межах файлової системи.

Більшість цих методів генерують виняткову ситуацію **SecurityException**, якщо програма не має прав доступу до файлової системи або окремих її файлів. Якщо встановлений **SecurityManager**, то його методи **checkRead()** та **checkWrite()** перевіряють, чи програма має дозвіл на доступ до файлової системи.

FilenameFilter

Призначення інтерфейсу **FilenameFilter** – в наданні способу вирішення, які файли вводити до списку імен файлів, а які – ні. Клас, який реалізує інтерфейс **FilenameFilter** повинен визначати метод **accept()**. Цей метод отримує об’єкт **File**, який визначає директорію, та **String** – ім’я файла. Метод **accept()** повертає **true**, якщо вказаний файл необхідно ввести до списку, і **false** в протилежному випадку. Розглянемо приклад використання інтерфейсу **FilenameFilter** для відбору файлів, що закінчуються певним суфіксом:

```
import java.io.File;
import java.io.FilenameFilter;
public class SuffixFilter
          implements FilenameFilter {
    private String suffix;
    public SuffixFilter(String suffix) {
        this.suffix = "." + suffix;
    }
    public boolean
          accept(File dir, String name) {
        return name.endsWith(suffix);
    }
}
```

Об'єкт **FilenameFilter** передається як параметр методу **list()** об'єкта **File** для фільтрації створюваного списку. Можна також використати **FilenameFilter** для обмеження списку файлів, що відображаються в **FileDialog**.

RandomAccessFile

Клас **RandomAccessFile** надає можливість непослідовного читання та запису файлів. **RandomAccessFile** має два конструктори, кожен з яких приймає два параметри. Перший параметр визначає файл для відкриття за допомогою об'єкта **String** або об'єкта **File**. Другий параметр – рядок, який повинен мати вигляд "r" або "rw". "r" означає що файл відкритий тільки для читання, "rw" – для читання та запису. Метод **close()** закриває файл. Всі методи **RandomAccessFile** можуть генерувати виняткову ситуацію **IOException** у випадку виникнення помилки.

RandomAccessFile визначає три методи **read()** для зчитування байтів із файла. Цей клас також реалізує інтерфейс **DataInput** і таким чином надає додаткові методи для читання із файлу. Крім цього, додаткові методи дають змогу читати прості типи даних машинно-незалежним способом. Всі ці методи у разі спроби читання за межами файла генерують виняткову ситуацію **EOFException**.

RandomAccessFile визначає три методи **write()** для запису інформації у файл. Він також реалізує інтерфейс **DataOutput** і таким чином надає додаткові методи для запису у файл. Крім цього, додаткові методи дають змогу записувати прості типи машинно-незалежним способом.

RandomAccessFile не відповідав би своєму імені, якщо б не надавав способу для непослідовного доступу до файла. Метод **getFilePointer()** повертає поточну позицію в файлі, тоді як метод **seek()** дозволяє перейти у вказану позицію, відповідно. Метод **length()** дає змогу визначити розмір файла в байтах.

5. Серіалізація об'єктів

Механізм серіалізації об'єктів в Java надає спосіб для запису об'єктів у потік байтів із можливістю відтворення їх згодом із потоку байтів. Ця можливість має низку цікавих застосувань. Наприклад, серіалізація об'єктів дозволяє стійке зберігання об'єктів, унаслідок чого об'єкти зберігаються у файлі для подальшого використання. Також копія об'єкта може бути відправлена через сокет іншій Java програмі. Об'єктна

серіалізація формує базис для механізму віддаленого виклику методів (*RMI, Remote Method Invocation*), який використовується у розподіленому програмуванні, та технології візуальних компонент *JavaBeans*. Об'єктна серіалізація підтримується рядом нових класів пакета java.io.

У найпростішій формі серіалізація об'єктів – спосіб автоматичного запису та завантаження об'єктів. Однак серіалізація об'єктів має більш глибокий зміст, включаючи повний контроль над процесом серіалізації та версифікації класу.

По суті, будь-який клас, який реалізує інтерфейс **`Serializable`**, може бути записаний та відтворений з потоку. Для серіалізації простих типів та об'єктів використовуються спеціальні потокові класи **`ObjectInputStream`** та **`ObjectOutputStream`**. Підкласи класу, який реалізує інтерфейс **`Serializable`**, також серіалізовані. За замовчуванням механізм серіалізації виконує запис значень нестатичних та нетимчасових змінних членів об'єкта.

5.1. Основи об'єктної серіалізації

Якщо клас розроблений для роботи із серіалізацією об'єктів, читання та запис екземплярів цього класу є доволі простим. Процес запису об'єкта до байтового потоку називається серіалізацією (*serialization*). Наприклад, запишемо об'єкт **`Color`** у файл:

```
FileOutputStream out =
          new FileOutputStream("tmp");
ObjectOutputStream objOut =
          new ObjectOutputStream(out);
objOut.writeObject(Color.red);
```

Все, що треба зробити, – це створити об'єкт **`ObjectOutputStream`** й опісля передати об'єкт, який планується записати, методу **`writeObject()`**. У процесі запису об'єкта в сокет чи будь-який інший пункт призначення, який чутливий в часі, необхідно викликати метод **`flush()`** після завершення передачі об'єкта в **`ObjectOutputStream`**.

Процес читання об'єкта з байтового потоку називається десеріалізацією (*deserialization*). Покажемо, як прочитати об'єкт **`Color`** із файла:

```
FileInputStream in =
          new FileInputStream("tmp");
ObjectInputStream objIn =
          new ObjectInputStream(in);
Color c = (Color)objIn.readObject();
```

Отже, для відтворення серіалізованого об'єкта необхідно створити об'єкт **ObjectInputStream** та викликати його метод **readObject()**.

5.2. Написання класів для роботи із серіалізацією

Створення класу для роботи із серіалізацією не є складнішим від простого застосування цього класу для серіалізації. По суті, **ObjectOutputStream** повинен записати достатню кількість інформації про стан об'єкта, щоб згодом можна було його відтворити. Якщо об'єкт містить посилання на інші об'єкти, ці об'єкти повинні бути записані, і так далі, допоки всі об'єктні посилання об'єкта-оригіналу, прямо чи опосередковано, не будуть записані.

У процесі створення нового класу необхідно вирішити, буде він серіалізованим чи ні. Немає сенсу проводити серіалізацію для всіх класів. Наприклад, об'єкт **Thread** інкапсулює інформацію, значущу тільки в процесі його створення, так що серіалізація тут недоречна. Щоб екземпляр класу був належним чином серіалізований, клас повинен реалізувати інтерфейс **Serializable**. Інтерфейс **Serializable** не визначає жодного методу чи змінної, він діє як індикатор серіалізованості. Метод **writeObject()** класу **ObjectOutputStream** генерує виняткову ситуацію **NotSerializableException**, якщо робиться спроба серіалізувати об'єкт, який не реалізує інтерфейс **Serializable**.

Механізм серіалізації за замовчуванням здійснюється методом **writeObject()** класу **ObjectOutputStream**. Коли виконується серіалізація об'єкта, клас об'єкта кодується разом з іменем класу, його сигнатурою, значеннями нестатичних (*non-static*) і нетимчасових (*non-transient*) полів об'єкта, включаючи будь-які інші об'єкти, на які посилається об'єкт (за винятком тих, які не реалізують інтерфейс **Serializable**). Чисельні посилання на такі самі об'єкти кодуються з використанням механізму *reference-sharing*, так що граф об'єктів може бути відповідним чином відтворений. Рядки та масиви є об'єктами Java, так що вони обробляються як об'єкти під час серіалізації та десеріалізації.

Механізм десеріалізації обернений до механізму серіалізації. За замовчуванням десеріалізація реалізується за допомогою методу **readObject()** класу **ObjectInputStream**. У процесі десеріалізації об'єкта нестатичні та нетимчасові поля об'єкта відтворюються з тими ж значеннями, які вони мали до серіалізації об'єкта, включаючи будь-які інші об'єкти, на які є об'єктні посилання (за винятком тих об'єктів, які не реалізують інтерфейс **Serializable**). Для попередження перезапису створених об'єктів нові екземпляри об'єктів завжди розміщуються під час

процесу десеріалізації. Десеріалізований об’єкт повертається як екземпляр класу **Object**, а тому необхідне зведення його до відповідного типу.

Деякі класи можуть просто реалізувати інтерфейс **Serializable** і використовувати вбудований механізм серіалізації та десеріалізації. Однак клас може вимагати обробки двох інших випадків при використанні серіалізації:

- якщо жоден суперклас класу не реалізує інтерфейс **Serializable**, клас повинен потурбуватися про запис будь-якої необхідної інформації про стан цих суперкласів під час серіалізації та зчитування інформації у процесі десеріалізації. Коли об’єкт серіалізований, вся інформація, що підлягає серіалізації, про стан, визначений цим класом і будь-яким суперкласом, який реалізує інтерфейс **Serializable**, записується в байтовий потік. Однак жодна інформація про стан суперкласу, який не реалізує інтерфейс **Serializable**, не буде записана в байтовий потік. Під час десеріалізації об’єкта, інформація про стан, визначений його серіалізованими суперкласами відтворюється з байтового потоку. За замовчуванням інформація про стан для несеріалізованого суперкласу ініціалізується шляхом виклику безаргументного конструктора суперкласу. Якщо ж суперклас не має конструктора без параметрів, десеріалізація зазнає невдачі і метод **readObject()** генерує виняткову ситуацію **NoSuchMethodError**;
- якщо об’єкт класу посилається на інший несеріалізований об’єкт, клас повинен потурбуватися про запис необхідної інформації про стан об’єкта під час серіалізації і зчитування інформації у процесі десеріалізації. Клас може перекрити логіку серіалізації за замовчуванням шляхом визначення такого методу:

```
private void
  writeObject(ObjectOutputStream stream)
          throws IOException
```

Тепер, під час серіалізації об’єкта, викликатиметься цей метод. Зауважимо, що **writeObject()** – **private**, так що він не може успадковуватися підкласами. Реалізація методу **writeObject()**, зазвичай, починається викликом методу **defaultWriteObject()** класу **ObjectOutputStream**, який реалізує логіку серіалізації за замовчуванням. Опісля **writeObject()**, звичайно, переходить до запису необхідної інформації, яка не була безпосередньо серіалізована.

Крім цього, клас може перекрити логіку десеріалізації за замовчуванням шляхом визначення такого методу:

```
private void
readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
```

Тепер, під час десеріалізації об'єкта, викликатимется цей метод **readObject()**. Цей метод також **private**, а тому не може успадковуватися підкласами. Реалізація методу **readObject()** зазвичай починається з виклику методу **defaultReadObject()** класу **ObjectInputStream**, який реалізує логіку десеріалізації за замовчуванням. Опісля метод **readObject()** переходить до зчитування відповідної інформації для відтворення значень, які не були безпосередньо серіалізовані.

Розглянемо клас, який реалізує інтерфейс **Serializable** та перекриває методи **writeObject()** і **readObject()**. Наведений нижче приклад є частиною роздруку класу, який здійснює доступ до даних з використанням об'єкта **RandomAccessFile**. Клас **RandomAccessFile** несеріалізований, оскільки він інкапсулює інформацію, яка має зміст лише в локальній системі і тільки протягом обмеженого проміжку часу:

```
public class TextFileReader
          implements Serializable {
    private transient RandomAccessFile file;
    private String browseFileName;
    ...
private void
  writeObject(ObjectOutputStream stream)
                        throws IOException{
        stream.defaultWriteObject();
stream.writeLong(file.getFilePointer());
    }
private void
readObject(ObjectInputStream stream)
                        throws IOException {
   try {
        stream.defaultReadObject();
   }catch (ClassNotFoundException e) {
        String msg = "Unable to find class";
            if (e.getMessage() != null)
                msg += ": " + e.getMessage();
            throw new IOException(msg);
        }
file =
  new RandomAccessFile(browseFileName, "r");
        file.seek(stream.readLong());
    }
```

```
}
```

Вищенаведений клас містить непридатний для серіалізації об’єкт **RandomAccessFile**. Проте він має достатньо інформації для відтворення об’єкта **RandomAccessFile**, подібного до оригіналу, під час десеріалізації. Ім’я файла, яке використовується об’єктом **RandomAccessFile**, визначається змінною **browseFileName**; ця інформація обробляється механізмом серіалізації за замовчуванням. Додатково метод **writeObject()** записує поточне значення вказівника у файлі об'єкта оригіналу за допомогою методу **getFilePointer()**, так що метод **readObject()** може отримати це значення за допомогою методу **seek()** нового об’єкта **RandomAccessFile**.

Використання інтерфейсу **Externalizable** робить можливим одержання повного контролю над серіалізованим поданням класу. Інтерфейс **Externalizable** успадковує інтерфейс **Serializable** та визначає два методи: **writeExternal()** та **readExternal()**, які автоматично викликаються під час серіалізації та десеріалізації об’єкта. Під час серіалізації об’єкта, який реалізує інтерфейс **Externalizable**, викликається метод **writeExternal()**. Цей метод відповідальний за запис всієї інформації про об’єкт. Аналогічно, під час десеріалізації об’єкта, що реалізує інтерфейс **Externalizable**, викликається його метод **readExternal()**. Цей метод відповідальний за зчитування всієї інформації про об’єкт. Зауважимо, що механізм **Externalizable** використовується замість (а не сумісно з) механізму **Serializable** у процесі обробки серіалізованих об’єктів. Розглянемо приклад реалізації інтерфейсу **Externalizable**:

```
import java.io.*;
import java.util.*;
class SimpleExternal implements
                  Externalizable {
int i;
String s; // відсутня ініціалізація
public SimpleExternal () {
System.out.println(
         "SimpleExternal конструктор");
// s, i не ініціалізовані
}
public SimpleExternal (String x, int a) {
System.out.println("SimpleExternal
         (String x, int a)");
   s = x; //s та i ініціалізовані
```

```
   i = a; //в конструкторі з параметрами
}
public String toString() { return s + i; }
public void writeExternal(ObjectOutput out)
                  throws IOException {
System.out.println(
          impleExternal.writeExternal");
// Це необхідно зробити:
   out.writeObject(s); out.writeInt(i);
}
public void readExternal(ObjectInput in)
throws IOException, ClassNotFoundException {
System.out.println(
          "SimpleExternal.readExternal");
// Це необхідно зробити:
   s = (String)in.readObject();
   i =in.readInt();
}
public static void main(String[] args) {
System.out.println(
          "Конструювання об'єктів:");
SimpleExternal b3 = new
          SimpleExternal ("A String ", 47);
System.out.println(b3);
   try {
ObjectOutputStream o = new ObjectOutputStream(
   new FileOutputStream("SimpleExternal.out"));
   System.out.println("Запис об'єкта:");
      o.writeObject(b3);
      o.close();
      // Отримання об'єкта:
ObjectInputStream in = new ObjectInputStream(
   new FileInputStream("SimpleExternal.out"));
   System.out.println("Відтворення об'єкта:");
      b3 = (SimpleExternal)in.readObject();
      System.out.println(b3);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

У вищенаведеному прикладі поля **s** та **i** ініціалізуються в конструкторі з параметрами, а не в конструкторі за замовчуванням. У цьому випадку слід потурбуватися про ініціалізацію цих змінних у методі **readExternal()**. Інакше відтворені значення будуть за замовчуванням нульовими.

Зауваження. Під час десеріалізації об'єкта, який реалізує інтерфейс **Serializable**, його відтворення відбувається на підставі даних серіалізації без виклику конструкторів об'єкта. У процесі відтворення об'єкта, який реалізує інтерфейс **Externalizable** викликається його відкритий конструктор без параметрів. Це слід враховувати під час використання інтерфейсу **Externalizable**.

Можлива ситуація, коли збереження деякого об'єкта члена є небажаним. Наприклад, дані про банківський рахунок, пароль чи інша конфедеційна інформація. Є три способи попередження небажаної серіалізації частини інформації:

- реалізувати інтерфейс **Externalizable**. В цьому випадку всю інформацію необхідно записати вручну;
- реалізувати інтерфейс **Serializable**. В цьому випадку вся інформація записується автоматично. Для заборони запису деяких полів використовується ключове слово **transient**;
- реалізувати інтерфейс **Serializable**, додати методи

```
private void writeObject
(ObjectOutputStream stream)throws IOException
private void
  readObject(ObjectInputStream stream)
  throws IOException, ClassNotFoundException
```

і в них визначити, які поля серіалізувати.

6. Стиснення даних і файлів

Починаючи з версії 1.1, Java містить новий пакет (java.util.zip), який охоплює класи для стиснення даних. Класи пакета java.util.zip підтримують два широко розповсюджені формати стиснення: GZIP та ZIP. Обидва формати базуються на ZLIB алгоритмі стиснення. Ієрархія класів стиснення даних зображена на рис. 6.1.

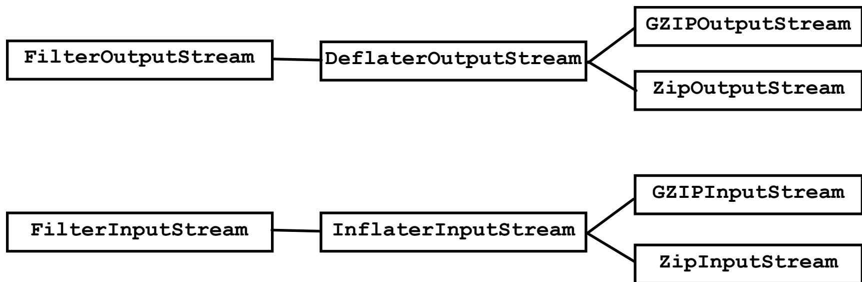

Рис. 6.1. Ієрархія класів стиснення даних

6.1. Стиснення даних

Пакет java.util.zip надає два підкласи класу **DeflaterOutputStream** для запису стиснених даних до потоку. Щоб записати стиснені дані в форматі GZIP, просто огортаємо **GZIPOutputStream** навколо потоку, що лежить в його основі, і записуємо в нього. Нижче наведено приклад стиснення файла з використанням формату GZIP:

```
import java.io.*;
import java.util.zip.*;
public class GZip {
  public static int sChunk = 8192;
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println(
          "Використання: GZip <джерело>");
      return; }
    // Створення вихідного потоку.
    String zipname = args[0] + ".gz";
    GZIPOutputStream zipout;
    try {
      FileOutputStream out =
          new FileOutputStream(zipname);
      zipout = new GZIPOutputStream(out);
    } catch (IOException e) {
      System.out.println("Неможливо створити "
                          + zipname + ".");
      return; }
```

```
      byte[] buffer = new byte[sChunk];
      // Compress the file.
      try {
        FileInputStream in =
            new FileInputStream(args[0]);
        int length;
        while ((length =
            in.read(buffer, 0, sChunk)) != -1)
          zipout.write(buffer, 0, length);
        in.close();
      } catch (IOException e) {
        System.out.println("Неможливо стиснути "
                           + args[0] + "."); }
      try { zipout.close(); }
      catch (IOException e) {}
    }
  }
```

Спершу перевіряємо наявність аргументу командного рядка, який подає ім'я файла Потім створюємо потік **GZIPOutputStream** навколо **FileOutputStream**, який подає ім'я файла із розширенням .gz. Далі відкриваємо файл-оригінал, зчитуємо частину інформації та записуємо її в потік **GZIPOutputStream**. Накінець, завершуємо роботу шляхом закриття відкритого потоку.

Запис даних у форматі ZIP більш заплутаний, проте має ширші можливості. Тоді як GZIP-файл містить тільки один стиснутий файл, ZIP-файл є фактично архівом файлів, один чи всі з яких можуть бути стиснуті. Кожен елемент ZIP-файла подається об'єктом **ZipEntry**. Коли здійснюється запис до **ZipOutputStream**, необхідно викликати **putNextEntry()** перед записом даних для кожного елемента. Нижче показано, як створити **ZipOutputStream**:

```
ZipOutputStream zipout;
try {
  FileOutputStream out =
          new FileOutputStream("archive.zip");
  zipout = new ZipOutputStream(out);
}
catch (IOException e) {}
```

Покажемо, як записати два файли до архіву. Перед записом необхідно викликати метод **putNextEntry()**. Таким чином ми створимо елемент з певним іменем. **ZipEntry** має інші поля, які можна встановити, але в більшості випадків в них немає необхідності.

```
try {
  ZipEntry entry = new ZipEntry("First");
  zipout.putNextEntry(entry);
}
catch (IOException e) {}
```

У цей момент можна записати вміст першого файла до архіву. Коли ми готові записати наступний файл до архіву, просто знову викликаємо метод **putNextEntry()**:

```
try {
  ZipEntry entry = new ZipEntry("Second");
  zipout.putNextEntry(entry);
}
catch (IOException e) {}
```

6.2. Розгортання даних

Щоб розгорнути дані, можна скористатися одним із двох підкласів класу **InflaterInputStream**, передбачених у пакеті java.util.zip. Для розгортання даних формату GZIP просто огортаємо **GZIPInputStream** навколо вхідного потоку та читаємо з нього. Нижче показано, як розгорнути GZIP-файл.

```
import java.io.*;
import java.util.zip.*;
public class GUnzip {
  public static int sChunk = 8192;
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println(
          "Використання: GUnzip <джерело>");
      return; }
    // Create input stream.
    String zipname, source;
    if (args[0].endsWith(".gz")) {
      zipname = args[0];
      source =
  args[0].substring(0, args[0].length() - 3);
    }
    else {
      zipname = args[0] + ".gz";
      source = args[0];
    }
    GZIPInputStream zipin;
```

```
try {
  FileInputStream in =
      new FileInputStream(zipname);
  zipin = new GZIPInputStream(in);
} catch (IOException e) {
  System.out.println("Неможливо відкрити "
                         + zipname + ".");
  return; }
byte[] buffer = new byte[sChunk];
// Decompress the file.
try {
  FileOutputStream out =
      new FileOutputStream(source);
  int length;
  while ((length =
      zipin.read(buffer, 0, sChunk)) != -1)
    out.write(buffer, 0, length);
  out.close();
} catch (IOException e) {
 System.out.println("Неможливо розгорнути "
                         + args[0] + ".");
}
try { zipin.close(); }
catch (IOException e) {}
    }
  }
```

Спершу перевіряємо наявність аргументу командного рядка, який подає ім'я файла. Якщо ім'я файла закінчується .gz, розшифровуємо, яке ім'я файла для розгорнутих даних буде використовуватися. В іншому випадку використовуємо отриманий аргумент і припускаємо, що стиснутий файл закінчується суфіксом .gz. Потім створюємо **GZIPInputStream** навколо **FileInputStream**, який подає стиснутий файл. Далі відкриваємо цільовий файл, зчитуємо порції даних з потоку **GZIPInputStream** та записуємо у файл призначення. Завершуємо роботу закриттям відкритих потоків.

Знову ж таки, ZIP-архів має складнішу структуру файла. Під час зчитування з потоку **ZipInputStream** необхідно викликати метод **getNextEntry()** перед зчитуванням кожного елемента. Якщо **getNextEntry()** повертає **null**, то більше немає елементів для читання. Нижче показано, як створити потік **ZipInputStream**:

```
ZipInputStream zipin;
```

```
try {
  FileInputStream in =
          new FileInputStream("archive.zip");
  zipin = new ZipInputStream(in);
}
catch (IOException e) {}
```

Припустимо, що ми хочемо прочитати два файли з цього архіву. Перед тим, як почати зчитування, необхідно викликати **getNextEntry()**:

```
try {
  ZipEntry first = zipin.getNextEntry();
}
catch (IOException e) {}
```

Починаючи з цього моменту, можна зчитати вміст першого елемента архіву. Коли ми завершимо зчитування першого елемента архіву, метод **read()** поверне –1. Тепер можемо знову викликати метод **getNextEntry()** для зчитування наступного елемента архіву:

```
try {
  ZipEntry second = zipin.getNextEntry();
}
catch (IOException e) {}
```

Якщо метод **getNextEntry()** поверне **null**, то це означає, що в архіві більше немає елементів. Отже, ми досягли кінця архіву.

Список літератури

1. *Арнолд К., Гослинг Дж., Холмс Д.* Язык программирования Java. М., 2002.
2. *Вебер Дж.* Технология Java в подлиннике. СПб., 2000.
3. *Морган М.* Java 2. М., 2000.
4. *Ноутон П., Шилдт Г.* Java 2 в подлиннике. СПб., 2001.
5. *Смирнов Н.* Java 2: Учебн. пособие. М., 2001.
6. *Хабибуллин И*. Самоучитель Java.– СПб., 2001.
7. *Эккель Б*. Философия Java. СПб., 2001.
8. http://www.sun.com, http://java.sun.com

ЗМІСТ

Вступ 3

1. Рядки та споріднені класи 4

1.1. Клас java.lang.String 5
1.2. Клас java.lang.StringBuffer 9
1.3. Клас java.util.StringTokenizer 11

2. Виняткові ситуації 12

2.1. Виявлення й обробка виняткових ситуацій 14
2.2. Виведення опису виняткових ситуацій 16
і роздрук стека викликів 16
2.3. Оголошення винятків 17
2.4. Генерація винятків 19
2.5. Створення власних класів виняткових ситуацій 21
2.6. Типи вбудованих виняткових ситуацій та їхня ієрархія 22

3. Багатопотокове програмування 25

3.1. Створення потоків 25
3.2. Атрибути потоків 29
3.3. Синхронізація потоків 32

4. Потоки введення-виведення 37

4.1. Класи потокового введення 37
4.2. Класи потокового виведення 49
4.3. Управління файлами 58

5. Серіалізація об’єктів 61

5.1. Основи об’єктної серіалізації 62
5.2. Написання класів для роботи із серіалізацією 63

6. Стиснення даних і файлів 68

6.1. Стиснення даних 69
6.2. Розгортання даних 71

Список літератури 73

Навчальне видання

Ірина Євстахіївна Бернакевич,
Петро Петрович Вагін

Програмування мовою Java: використання фундаментальних класів

Тексти лекцій

Редактор М. В. Ріпей

Технічний редактор С. З. Сеник

Підп. до друку . .2002. Формат 60×84/16. Папір друк. .
Різогр. друк. Умов. друк. арк. . Тираж 100. Зам. .
Видавничий центр Львівського національного університету
імені Івана Франка
79000, м. Львів, вул. Дорошенка, 41