Тема 5. Робота з файловою системою

Лекція 7

Потоки введення – виведення

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

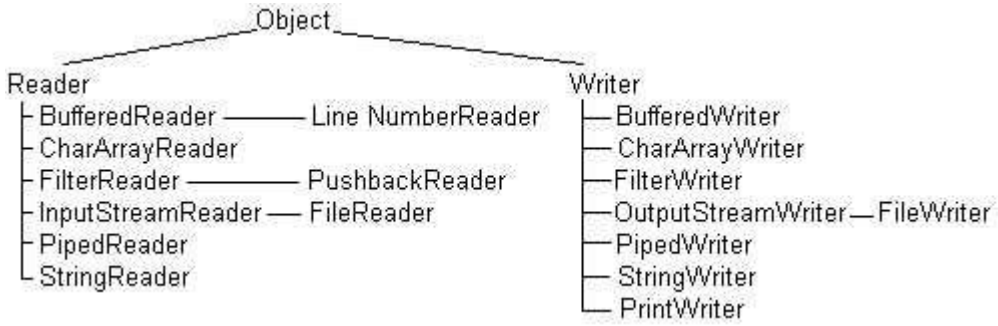

Рис 5.1. Класи символьних потоків

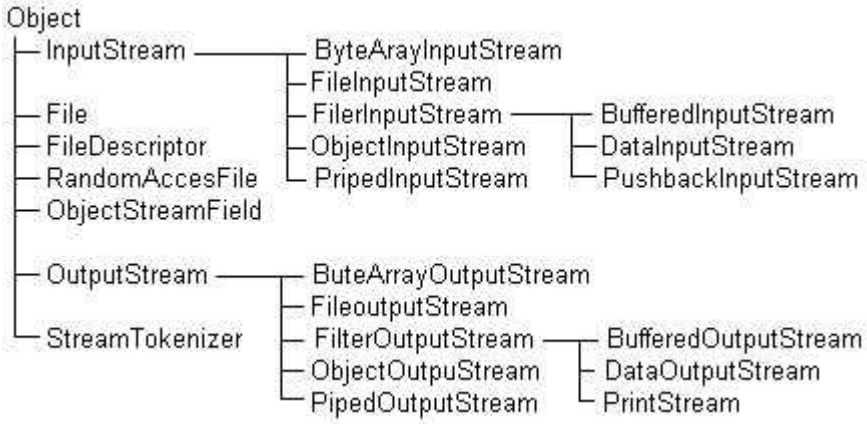

Рис. 5.2. Класи байтових потоків

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

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

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

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

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

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

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

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

`BufferedWriter` `BufferedOutputStream`

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

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

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

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

Файлове введення-виведення

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

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

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

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

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

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

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

Відразу після виконання конструктора можна читати файл:

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

чи записувати в нього:

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

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

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

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

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

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

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

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

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

Буферизоване введення-виведення

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

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

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

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

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

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

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

Розглянемо приклад буферизованного файлового введення-виведення.

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

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

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

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

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

Робота з бінарними файлами. Потік примітивів

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

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

Також в `DataOutputStream` можна записувати рядки:

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

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

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

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

Приклад запису та читання примітивів із файлу:

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

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

Робота з текстовими файлами

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

Приклад: читання чисел із файлу.

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

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

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

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

Серіалізація

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

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

Java має два способи серіалізації об'єктів:

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

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

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

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

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

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

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

Розглянемо приклад:

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

     // Будь-які поля і методи

}
```

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

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

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

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

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

```
// Десеріалізація
```

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

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

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

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

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

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

Табл. 5.1. Способи серіалізації.

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

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

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

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

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

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

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

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

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

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

Додаткові посилання за темою:

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

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

Маніпулювання файлами і директоріями

В Java існує два способи маніпулювати файлами і директоріями:

1) за допомогою класу java.io.File,

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

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

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

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

Конструктор не перевіряє, чи існує файл з таким ім'ям, тому після створення об'єкта слід це перевірити за допомогою метода `exists()`.

Виділення частин шляху:

Клас `File` має кілька методів для отримання різних частин шляху до файлу, наприклад:

```
String s = f.getAbsolutePath() + " " + f.getPath() + " " + f.getName();
```

Ці методи не мають відношення до фізично існуючих тек на диску, а лише розбирають рядок, що зберігається всередині змінної `File`. Тому, наприклад, виклик

```
new File("../").getParent()
```

поверне `null`, т. я. в даному випадку об'єкт `File` не зберігає імені батьківської директорії.

Отримати повний шлях до файлу:

```
String s1 = f.getCanonicalPath();
```

Цей метод, на відміну від попередніх, викликає функції операційної системи, щоб перетворити відносний шлях в повний фізичний шлях. Тому він може генерувати виключення.

Отримання інформації про файл

Клас `File` містить багато методів, що дозволяють дізнатися про різні властивості файлу або директорії.

Перш за все, логічними методами `isFile()`, `isDirectory()` можна з'ясувати, чи є шлях, вказаний в конструкторі, шляхом до файлу або директорії (папки).

Для директорії можна отримати її вміст – список імен файлів і вкладених директорій методом `list()`, що повертає масив рядків `String[]`. Можна отримати той же список у вигляді масиву об'єктів `File[]` методом `listFiles()`. Можна вибрати зі списку тільки деякі файли, реалізувавши інтерфейс `FileNameFilter` і звернувшись до методу `list(FileNameFilter filter)`.

Для файлу можна отримати його довжину в байтах методом `length()`, час останньої модифікації в секундах з 1 січня 1970 р методом `lastModified()`. Якщо файл не існує, ці методи повертають нуль.

Логічні методи `canRead()`, `canWrite()`, `canExecute()` показують права доступу до файлу.

Покажемо роботу цих методів на прикладі:

```
File f = new File( "path/name.ext" );
f.exists();                  // Чи існує файл/директорія з таким іменем?
f.isDirectory();             // Перевірка на директорію
f.length();                  // Для файла – його розмір, для директорії - 0
f.lastModified();            // Дата останньої зміни (в мс. з 1.01.1970)
f.canRead(), f.canWrite(), f.canExecute();   // Перевірка дозволів доступу
f.isHidden();                // Чи є файл прихованим?
File ff[] = f.listFiles();   // Для директорій – отримати список файлів
```

Операції над файлами та <u>порожніми</u> директоріями:

Якщо директорія із зазначеним в конструкторі `File()` шляхом не існує, її можна створити логічним методом `mkDir()`. Цей метод повертає true, якщо директорію вдалося створити. Логічний метод `mkDirs()` створює також і всі неіснуючі директорії, зазначені у шляху до файлу.

Порожня директорія видаляється методом `delete()`. Якщо директорія непорожня, то `delete()` нічого не робить – спершу необхідно видалити весь вміст директорії.

Файл можна перейменувати логічним методом `renameTo(File newName)` або видалити логічним методом `delete()`. Ці методи повертають true, якщо операція пройшла вдало.

Якщо файл з вказаним в конструкторі шляхом не існує, його можна створити логічним методом `createNewFile()`, що повертає true, якщо файл не існував, і його вдалося створити, і false, якщо файл вже існував.

Розглянемо тепер кілька прикладів використання методів роботи з файлами і директоріями для вирішення найбільш поширених задач.

Приклад 1. Виведемо основні властивості файлу і директорії. Результат запуску цієї програми показаний на рис. 5.3. Зверніть увагу, що для директорії ця програма виводить тільки вкладені файли першого рівня вкладеності.

```
import java.io.*;
class FileTest {
```

```
    public static void main(String[] args) throws IOException{
        // Змінюємо кодування потоку виведення
        PrintWriter pw = new PrintWriter(
            new OutputStreamWriter(System.out, "Cp866"), true);

        // Створюємо шлях до файлу
        File f = new File("FileTest.Java");
        pw.println();

        // Отримуємо властивості файла
        pw.println("Файл \"" + f.getName() + "\" " +
            (f.exists()?"":"не ") + "існує");
        pw.println("Ви " + (f.canRead()?"":"не ") + "можете читати файл");
        pw.println("Ви " + (f.canWrite()?"":"не ") + "можете записувати в файл");
        pw.println("Розмір файла " + f.length() + " б");
        pw.println() ;

        // Створюємо шлях до директорії
        File d = new File(" D:\\jdk1.3\\MyProgs ");
        pw.println("Вміст каталога:");
        // Можна запитувати вміст тільки якщо
        // d існує та являється директорією
        if (d.exists() && d.isDirectory()) {
            String[] s = d.list();
            for (int i = 0; i < s.length; i++)
                pw.println(s[i]);
            }
        }
    }
}
```

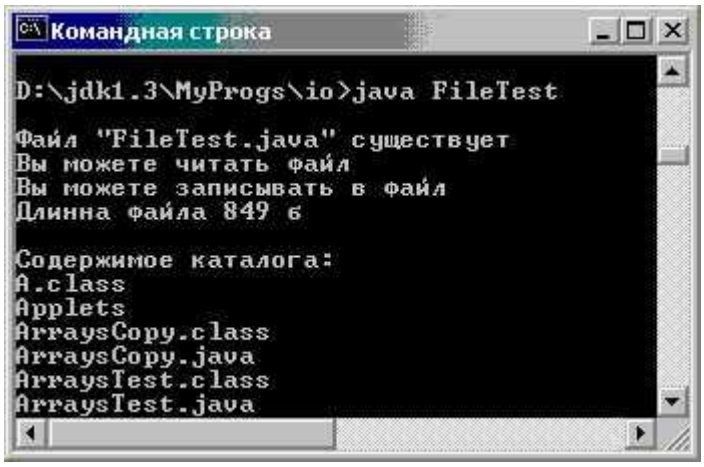

Рис. 5.3 Результат роботи програми

Приклад 2. Виведемо на екран вміст директорії з урахуванням всіх її піддиректорій і визначимо її загальний розмір (в байтах). Будемо виводити назви файлів і кількість днів з часу їх останньої зміни.

```
    try {
      // Створюємо об'єкт File і зв'язуємо його з директорією нашого проекту
      File dir = new File( "../" );
      System.out.println( dir.getCanonicalPath() );
      Викликаємо рекурсивну функцію перегляду вмісту директорії list()
      System.out.println( "Total size - "+list( dir, 0 )+" "+dir.length());
    } catch (Exception e) { e.printStackTrace(); }

    // Рекурсивна функція перегляду вмісту директорії
    static long list( File f, int depth ) {

        // Виводимо пробіли щоб показати рівень вкладеності
```

```
        for( int i = 0; i<depth; i++ ) System.out.print("  ");

        // Визначаємо кількість днів з часу останньої зміни файлу
        long days = (System.currentTimeMillis() - f.lastModified())
                          / 1000 / 60 / 60/ 24;
        System.out.println( f.getName() + " - " + days + " days ago" );

        // Якщо це директорія, то рекурсивно викликаємо list() для кожного
        // вкладеного файла
        if( f.isDirectory() ) {
             long sz = 0;
             for( File f1 : f.listFiles() )
                  sz += list( f1, depth+1 );   // Сумуємо розміри вкладених файлів
             return sz;                        // і повертаємо результат
        } else {
             return f.length();                // Для звичайного файлу –
                                               // повертаємо його розмір
        }
    }
```

Приклад 3. Напишемо метод видалення непорожньої директорії. Перш ніж викликати метод delete() для директорії перевіримо, чи має вона вкладені файли і директорії. Якщо має – то рекурсивно запустимо наш метод для кожного вкладеного файлу і директорії.

```
static void deleteDir( File dir ) {
    try {
        if( dir.isDirectory() )
             for( File f : dir.listFiles() )  deleteDir( f );
        dir.delete();
    } catch (Exception e) {
        System.out.println("Cannot delete " + dir.getAbsolutePath());
        // getAbsolutePath() не генерує виключень
    }
}
```

Клас `File` не містить методу копіювання файлу. Копіювання реалізується через потокові операції читання-запису (за допомогою класів `FileInputStream` і `FileOutputStream`, як це робилося на початку цієї лекції), або за допомогою класу `Files`, як буде показано нижче.

В цілому клас **Files** надає можливості, аналогічні `File`, але має наступні відмінності:

- для представлення файла (а точніше шляху до файлу) використовує інтерфейс **Path**;
- не потрібно створювати об'єкти класу `Files`, оскільки всі його методи є статичними;
- `Files` має деякі додаткові методи, аналогів яким немає в класі `File`, наприклад:
  `copy(), move();`
- багато методів мають додаткові параметри, що управляють їх роботою, наприклад:

```
Files.move(srcPath, destPath, REPLACE_EXISTING);
```

Приклад 4. Скопіюємо файл з допомогою класу `Files`.

```
static void copyFile( String name, String newName ) {
    try {

        // Створюємо об'єкти Path і зв'язуємо їх із шляхами
        Path p1 = new File(name).toPath(), p2 = new File(newName).toPath();

        // Перевіряємо, чи не є один із шляхів директорією
        if( Files.isDirectory(p1) || Files.isDirectory(p2) ) return;

        // Копіювати можна тільки якщо вихідний файл існує
        if( Files.exists(p1) )
             // Якщо другий файл вже існує, то його треба спочатку видалити
```

```
            if( Files.exists(p2) ) Files.delete(p2);
            // Тепер можна копіювати
            Files.copy( p1, p2 );   // FileNotFoundExxeption,
                                    // FileAlreadyExistsException !!!!
    } catch (Exception e) { e.printStackTrace(); }
}
```