

## Потоки Java

Детальніше:

<https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html>

У Java API об'єкт, з якого можна читати послідовність байт, називають вхідним потоком (input stream). Об'єкт, у який можна записувати послідовність байт, називають вихідним потоком (output stream).

Абстрактні класи `InputStream` та `OutputStream` формують основу ієрархії класів введення-виведення.

Найчастіше джерелами та місцями призначення байт є файли, хоча можуть бути і мережеві з'єднання чи блоки пам'яті.

Класи [FileInputStream](#), [FileOutputStream](#)

Конструктори класу `FileOutputStream`.

<a href="#">FileOutputStream</a> ( <a href="#">File</a> file)	Створює вихідний файловий потік для запису у файл, представлений об'єктом <code>File</code>
<a href="#">FileOutputStream</a> ( <a href="#">File</a> file, boolean append)	Створює вихідний файловий потік для запису у файл, представлений об'єктом <code>File</code> . Якщо параметр <code>append</code> має значення <code>true</code> , байти буде дописано у кінець файла
<a href="#">FileOutputStream</a> ( <a href="#">FileDescriptor</a> fdObj)	Створює вихідний файловий потік для запису до заданого параметром <code>fdObj</code> файлового дескриптора, котрий представляє існуюче з'єднання з існуючим файлом у файловій системі
<a href="#">FileOutputStream</a> ( <a href="#">String</a> name)	Створює вихідний файловий потік для запису у файл з вказаним іменем <code>name</code> .
<a href="#">FileOutputStream</a> ( <a href="#">String</a> name, boolean append)	Створює вихідний файловий потік для запису у файл з вказаним іменем <code>name</code> . Якщо параметр <code>append</code> має значення <code>true</code> , байти буде дописано у кінець файла

void	<a href="#">write</a> (byte[] b) Записує <code>b.length</code> байт зі вказаного масиву <code>b</code> до вихідного файлового потоку
void	<a href="#">write</a> (byte[] b, int off, int len) Записує <code>len</code> байт зі вказаного масиву <code>b</code> , починаючи з відступу <code>off</code> до вихідного файлового потоку
void	<a href="#">write</a> (int b) Записує вказаний байт <code>b</code> до вихідного файлового потоку

Зверніть увагу на те, що параметр методу `write( int b)` оголошено як `int`, хоча метод описано як такий, що записує байт до потоку. Даний метод класу `FileOutputStream` успадковано з класу `OutputStream`. Згідно із [документацією](#) тільки молодші 8 біт параметра буде використано, старші 24 біти буде проігноровано.

Розглянемо кожен із варіантів методу `write`. Запишемо до файла масив байт.

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

```

public class Test {
    public static void main(String[] args) {
        try {
            byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
            OutputStream os = new FileOutputStream("test.txt");
            os.write(text);
            os.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

У результаті буде створено файл test.txt, розмір файла: 5 байт. Вміст (для перегляду використано редактор bless) наведено на рис. 1.

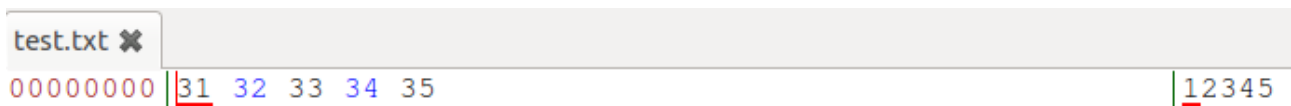


Рисунок 1 – Вміст файла test.txt (зліва – шістнадцяткові коди, справа – символічне представлення)

Оголосимо посилання на об'єкт потоку таким чином, що байти записувалися у кінець файла.

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
            OutputStream os = new FileOutputStream("test.txt", true);
            os.write(text);
            os.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

У результаті щоразу при запускові програми файл test.txt поповнюватимуть байти масиву text, записані у кінець файла.

Розглядемо методом write, котрий оперує зміщенням блоку байт усередині масиву та їхньою кількістю.

```

import java.io.*;

```

```

public class Test {
    public static void main(String[] args) {
        try {
            byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
            OutputStream os = new FileOutputStream("test.txt");
            os.write(text, 1, text.length - 1);
            os.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

У даному прикладі задано одиничне зміщення у масиві байт, призначених для запису у потік, тому, порівняно із попереднім прикладом, у результаті відсутнє значення 0x31 (початковий елемент масиву). Оскільки кількість байт, призначених для запису у потік, зменшилася на 1, значення другого параметра методу: `text.length - 1` (останній елемент масиву). Таким чином, у файл будуть записані елементи масиву з індексами від 1 до 4. Елемент з індексом 0 буде пропущений.

Результат:



test.txt ✕

00000000 | 32 33 34 35 | 2345

Рисунок 2 – Вміст файла test.txt. Випадок запису у файл частини масиву байт

Реалізуємо побайтовий запис елементів масиву до файла.

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
            OutputStream os = new FileOutputStream("test.txt");
            for(byte b : text)
                os.write(b);
            os.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

Результат буде аналогічним до випадку з методом `write(byte[] b)`:



Рисунок 3 – Вміст файлу test.txt. Побайтовий запис до файлу

Скористаємося щойно записаним файлом, і прочитаємо його за допомогою методів класу `FileInputStream` (таб. 2).

### Конструктори класу `FileInputStream`

#### `FileInputStream(File file)`

Створює `FileInputStream` шляхом відкриття з'єднання з існуючим файлом, ім'я котрого задають за допомогою об'єкта `file`

#### `FileInputStream(FileDescriptor fdObj)`

Створює `FileInputStream` шляхом використання файлового дескриптора `fdObj`, котрий представляє існуюче з'єднання з існуючим файлом файлової системи

#### `FileInputStream(String name)`

Створює `FileInputStream` шляхом відкриття з'єднання з існуючим файлом, ім'я котрого задають за допомогою шляху `name` у файловій системі

int	<code>read()</code> Читає байт даних з вхідного файлового потоку
int	<code>read(byte[] b)</code> Читає до <code>b.length</code> байт даних з вхідного файлового потоку у масив байт <code>b</code>
int	<code>read(byte[] b, int off, int len)</code> Читає до <code>len</code> байт даних з вхідного файлового потоку у масив байт <code>b</code> , записуючи їх зі зміщенням у масиві <code>off</code>

Для того, щоб дізнатися об'єм даних, доступних для читання, використаємо метод `available` класу `FileInputStream`. Метод не блокує виконання потоку. У деяких випадках може виглядати блокуючим у зв'язку із повільним виконанням (наприклад, при читанні у повільній мережі). Таким чином, читання з потоку не буде блокуючим, оскільки метод `read` спробує прочитати доступну для нього кількість байт і не виходитиме поза її межі.

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            InputStream is = new FileInputStream("test.txt");
            int size = is.available();
            for(int i = 0; i < size; i++) {
                byte text = (byte)is.read();
                System.out.printf("%h ", text);
            }
            is.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

```
}  
}
```

Результат:

31 32 33 34 35

Метод `read()` блокує виконання потоку, якщо читання все ще не є можливим (наприклад, при читанні з сокетів). Повертає `-1`, коли досягнуто кінець файла.

У наступному прикладі записаний файл прочитано до масиву байт цілим (так, щоб початковий елемент масиву було заповнено початковим байтом файла).

```
import java.io.*;  
public class Test {  
    public static void main(String[] args) {  
        try {  
            InputStream is = new FileInputStream("test.txt");  
            int size = is.available();  
            if (size > 0) {  
                byte[] text = new byte[size];  
                is.read(text);  
                for(int i = 0; i < size; i++)  
                    System.out.printf("%h ", text[i]);  
            }  
            is.close();  
        } catch (IOException e) {  
            System.out.println(e);  
        }  
    }  
}
```

Результат:

31 32 33 34 35

Аналогічно до запису частини масиву байт у файл, прочитаємо з файла декілька байт до масиву, вказавши зміщення у ньому та кількість байт.

```
import java.io.*;  
public class Test {  
    public static void main(String[] args) {  
        try {  
            InputStream is = new FileInputStream("test.txt");  
            int size = is.available();  
            if (size > 3) {  
                System.out.println("length: " + size);  
            }  
        }  
    }  
}
```

```

        byte[] text = new byte[5];
        is.read(text, 1, 2);
        for(int i = 0; i < 5; i++)
            System.out.printf("%h ", text[i]);
    }
    is.close();
} catch (IOException e) {
    System.out.println(e);
}
}
}

```

Результат:

```

length: 5
0 31 32 0 0

```

У даному прикладі перевіряють кількість байт, доступних для читання, і, якщо кількість більша за 3, створюють масив з 5 елементів типу byte та записують у нього 2 байти з файлового потоку зі зміщенням 1. Тому початковий елемент масиву має значення 0 (ініціалізація за замовчуванням), наступних два – відповідні байти з файлу і останні два – 0 (ініціалізація за замовчуванням).

Усі методи read – блокуючі.

За потреби можна пропустити певну кількість байт у потокові. Метод [skip\(long n\)](#) пропускає n байт і повертає кількість байт, яка насправді була пропущена.

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            InputStream is = new FileInputStream("test.txt");
            int size = is.available();
            if (size > 1) {
                System.out.println(is.skip(1) + " byte/s skipped");
                System.out.printf("%h\n", is.read());
                System.out.println(is.skip(-1) + " byte/s skipped");
                System.out.printf("%h", is.read());
            }
            is.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

Результат:

```

1 byte/s skipped

```

32  
-1 byte/s skipped  
32

Метод може припинити пропускати байти через деяку невелику кількість байт (навіть, 0). Якщо *n* – від’ємне число, метод намагатиметься здійснювати пропуск у зворотньому напрямі. Якщо файл не підтримує ріверс, генерується `IOException`. Знак повернутого методом числа залежить від напрямку: додатній відповідає прямому, від’ємний – зворотньому.

Метод може “пропускати” більше байт, ніж залишилося у файлі. У цьому випадкові не виникає виняткової ситуації і кількість пропущених байт може включати деяку кількість байт поза EOF файла. При спробі прочитати з потоку далі його останнього символу буде отримано -1, що відповідає EOF.

Виняткові ситуації:

[IOException](#) - якщо *n* від’ємне і вказує на передпочаток файла, якщо потік не підтримує переміщення чи виникають помилки I/O.

Класи [BufferedOutputStream](#) та [BufferedInputStream](#)

Клас `BufferedOutputStream` імплементує буферизований вихідний потік. За допомогою такого потоку програма може записувати байти до основного вихідного потоку без потреби виклику системи, що лежить в основі, для кожного байту окремо.

Поля

protected byte[]	<a href="#">buf</a> внутрішній буфер для збереження даних
protected int	<a href="#">count</a> кількість дійсних байт у буфері

Конструктори

<a href="#">BufferedOutputStream</a> ( <a href="#">OutputStream</a> out)
Створює новий буферизований вихідний потік для запису даних до вказаного вихідного потоку, що лежить в основі такої передачі
<a href="#">BufferedOutputStream</a> ( <a href="#">OutputStream</a> out, int size)
Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size

Деякі методи класу (детальніше – див. [посилання](#))

void	<a href="#">flush()</a> Очищає буферизований вихідний потік
void	<a href="#">write</a> (byte[] b, int off, int len) Записує len бакт з маиву b, починаючи з off до буферизованого вихідного потоку
void	<a href="#">write</a> (int b) Записує вказаний байт до буферизованого вихідного потоку

Методи, успадковані з `java.io`. [FilterOutputStream](#)

void	write(byte[] b) Записує до буферизованого вихідного потоку b.length байт
void	close() Закриває буферизований вихідний потік і звільняє системні ресурси, асоційовані з цим потоком. Метод close() класу BufferedOutputStream викликає метод flush() цього класу, після цього – викликає метод close() вихідного потоку, що лежить в основі

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
        try{
            BufferedOutputStream bufferedOutputStream =
                new BufferedOutputStream(
                    new FileOutputStream("test.txt")
                );
            bufferedOutputStream.write(text);
            bufferedOutputStream.close();
        } catch ( IOException e){
        }
    }
}
```

Результат:

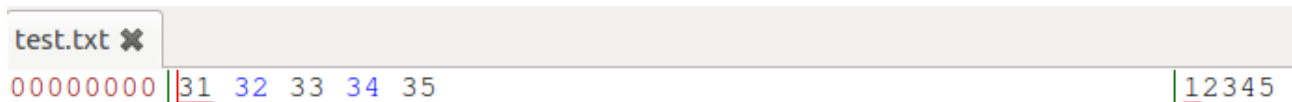


Рисунок 4 – Вміст файла test.txt. Запис за допомогою буферизованого потоку виведення

```
import java.io.*;
import java.util.Arrays;
public class Test {
    public static void main(String[] args) {
        byte[] text = new byte[5000000];
        Arrays.fill(text, (byte)0x31);
        try{
            FileOutputStream fileOutputStream =
                new FileOutputStream("test.txt");
            final long startTime = System.nanoTime();
            for(byte b : text)
                fileOutputStream.write(b);
            fileOutputStream.close();
            final long duration = System.nanoTime() - startTime;
            System.out.printf("%.2f s", duration / 1000000000.0);
        }
```



```
    } catch ( IOException e){  
    }  
}  
}
```

Результат:

7.27 s

```
import java.io.*;  
import java.util.Arrays;  
public class Test {  
    public static void main(String[] args) {  
        byte[] text = new byte[5000000];  
        Arrays.fill(text, (byte)0x31);  
        try{  
            BufferedOutputStream bufferedOutputStream =  
                new BufferedOutputStream(  
                    new FileOutputStream("test.txt")  
                );  
            final long startTime = System.nanoTime();  
            for(byte b : text)  
                bufferedOutputStream.write(b);  
            bufferedOutputStream.close();  
            final long duration = System.nanoTime() - startTime;  
            System.out.printf("%.2f s", duration / 1000000000.0);  
        } catch ( IOException e){  
        }  
    }  
}
```

Результат: 0.11 s

Конкретні результати при відтворенні прикладів залежатимуть від конфігурації системи. Легко помітити, що записування у файл 5 000 000 байт з використанням буферизованого потоку виведення відбувається набагато швидше за побайтовий запис без використання буфера.

Буферизоване виведення зручне у випадкові, якщо у файл часто записують невеликі об'єми даних.

Розмір буфера за замовчуванням: 8192 байт (оголошений у файлі `BufferedOutputStream.java`)

```
public BufferedOutputStream(OutputStream out) {  
    this(out, 8192);  
}
```

Важливо пам'ятати про закривання буферизованого потоку за допомогою методу `close()`. При цьому вивільняється буфер (еквівалент виклику методу `flush()`) та закривається основний файловий потік виведення.

Якщо не закрити потік і у буфері залишаться дані, вони не будуть записані до файла, оскільки не буде викликано метод `flush()`. У наведеному нижче прикладі змодельовано описану ситуацію.

```
import java.io.*;
public class Test extends BufferedOutputStream {
    public static void main(String[] args) {
        try {
            Test bufferedOutputStream =
                new Test(
                    new FileOutputStream("test.txt"){
                        @Override
                        public void close() throws IOException {
                            System.out.println("FileOutputStream is closed");
                            super.close();
                        }
                    }
                );
            bufferedOutputStream.write(0x31);
            bufferedOutputStream.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
    public Test(OutputStream os) {
        super(os);
    }
    @Override
    public synchronized void write(int b) throws IOException {
        System.out.println("buffer size: " + super.buf.length);
        super.write(b);
        System.out.println("bytes count: " + super.count);
    }
    @Override
    public synchronized void flush() throws IOException {
        System.out.println("BufferedOutputStream is flushed");
        super.flush();
    }
}
```

Результат:

```
buffer size: 8192
bytes count: 1
BufferedOutputStream is flushed
FileOutputStream is closed
```



Рисунок 4 – Вміст файлу test.txt. Результат запису у файл із наступним закриттям буферизованого потоку виведення

Для отримання доступу до полів класу `BufferedOutputStream`, котрі оголошені як `protected`, створимо клас `Test`, нащадок `BufferedInputStream`. Конструктор класу викликає конструктор батьківського класу, передаючи йому свій параметр – посилання на об'єкт класу `OutputStream` (у даному випадкові – `FileOutputStream`).

Власна реалізація методу `write(int b)` дає можливість дізнатися розмір буфера потоку та кількість елементів, що заповнюють цей буфер. Як видно з результату, розмір буфера 8192 байт, після запису 1 байту до буфера поле `count` батьківського класу дорівнює 1.

Зверніть увагу на те, що метод `flush()` не викликають явно, проте результат виконання програми демонструє, що виклик відбувається. Згідно із документацією, метод `close()` класу `BufferedOutputStream` вивільняє буфер за допомогою методу `flush()`. Для демонстрації цього створено власну реалізацію методу, котра повідомляє про його виклик.

Також метод `close()` повинен закрити потік, котрий лежить в основі передачі байт до файлу, у даному випадкові – `FileOutputStream`. При ініціалізації посилання на об'єкт класу `BufferedInputStream` як параметр у конструктор передано анонімний клас на основі `FileOutputStream` з власною реалізацією методу `close()`, котра повідомляє про його виклик.

Якщо закоментувати рядок `bufferedOutputStream.close()` – запис до файлу не відбудеться, оскільки не буде викликано метод `flush()`. Даний метод можна також викликати і вручну.

#### Клас [BufferedInputStream](#)

##### Поля

protected byte[]	<a href="#">buf</a> Внутрішній буфер, призначений для зберігання даних
protected int	<a href="#">count</a> Індекс, більший на 1 за індекс останнього значущого елемента буфера
protected int	<a href="#">marklimit</a> Максимальна кількість елементів, котру можна прочитати у прямому напрямі після виклику методу <code>mark()</code> перед викликом методу <code>reset()</code> , котрий може завершитися невдачею і спровокувати появу виняткової ситуації
protected int	<a href="#">markpos</a> Значення, котре мало поле <code>pos</code> на момент виклику методу <code>mark()</code>
protected int	<a href="#">pos</a> Поточна позиція у буфері

##### Конструктори

<a href="#">BufferedInputStream(InputStream in)</a> Створює <code>BufferedInputStream</code> та зберігає його аргумент, вхідний потік <code>in</code> , для подальшого використання
<a href="#">BufferedInputStream(InputStream in, int size)</a>

Створює `BufferedInputStream` з буфером заданого розміру та зберігає його аргумент, вхідний потік `in`, для подальшого використання

## Методи

int	<a href="#">available()</a> Повертає оціночну кількість байт, котра може бути пропущена чи прочитана з вхідного потоку без блокування наступного виклику методу цього вхідного потоку
void	<a href="#">close()</a> Закриває цей вхідний потік та звільняє ресурси системи, пов'язані з потоком
void	<a href="#">mark(int readlimit)</a> Запам'ятовує позицію у потоці, до якої пізніше можна повернутися за допомогою методу <code>reset()</code> . <a href="#">Детальніше</a> про метод – у батьківському класі <code>InputStream</code>
boolean	<a href="#">markSupported()</a> Перевіряє, чи даний вхідний потік підтримує методи <code>mark()</code> та <code>reset()</code>
int	<a href="#">read()</a> Читає байт з вхідного потоку. <a href="#">Детальніше</a> – у батьківському класі <code>InputStream</code> .
int	<a href="#">read(byte[] b, int off, int len)</a> Читає байти із байт-орієнтованого вхідного потоку у вказаний масив <code>b</code> , записуючи у нього елементи зі зміщенням <code>off</code> від початку; <code>len</code> – максимальна кількість елементів, котрі потрібно прочитати
void	<a href="#">reset()</a> Повертає позицію у буфері до такої, що була запам'ятована за допомогою методу <code>mark()</code> . <a href="#">Детальніше</a> – у батьківському класі <code>InputStream</code>
long	<a href="#">skip(long n)</a> Пропускає вказану кількість байт. <a href="#">Детальніше</a> – у батьківському класі <code>InputStream</code> .

Клас `BufferedInputStream` реалізує буферизацію вхідного потоку даних і забезпечує підтримку міток, що дозволяють за потреби повернутися до позиції у файлі, позначеної міткою.

При створенні `BufferedInputStream` створюється внутрішній буфер. У процесі читання чи пропуску байт зі вхідного потоку буфер оновлюється за потреби. При проставлянні мітки запам'ятовується позиція мітки у файлі. При потребі до цієї позиції можна повернутися (`reset`). При цьому усі байти, що були прочитані, починаючи з позиції мітки, будуть прочитані повторно. Потік може вважати мітку нечинною, якщо кількість байт, прочитаних після неї, перевищує кількість, встановлену при створенні мітки (`mark(int position)`).

Подібно до описаного, порівняємо швидкість читання з файла за допомогою буферизованого вхідного потоку і побайтове читання без буферизації. Для цього скористаємося одним із попередніх прикладів, за допомогою якого згенеруємо файл 5 000 000 байт даних.

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        byte[] text = new byte[5000000];
```

```

try {
    BufferedInputStream bufferedInputStream =
        new BufferedInputStream(
            new FileInputStream("test.txt"));
    long size = bufferedInputStream.available();
    final long startTime = System.nanoTime();
    if (size == 5000000) {
        int j = 0;
        int b;
        while ((b = bufferedInputStream.read()) != -1)
            text[j++] = (byte) b;
        System.out.println("cycle counter: " + j);
    }
    final long duration = System.nanoTime() - startTime;
    bufferedInputStream.close();
    System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
    System.out.printf("%.2f s", duration / 1000000000.0);
} catch (IOException e) {
    System.out.println(e);
}
}

```

Результат:

```

cycle counter: 5000000
0x31 0x31 ...
0.12 s

```

Тепер зробимо аналогічне, тільки без буферизації.

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        byte[] text = new byte[5000000];
        try {
            FileInputStream fileInputStream =
                new FileInputStream("test.txt");
            long size = fileInputStream.available();
            final long startTime = System.nanoTime();
            if (size == 5000000) {
                int j = 0;
                int b;
                while ((b = fileInputStream.read()) != -1)
                    text[j++] = (byte) b;
                System.out.println("cycle counter: " + j);
            }
            final long duration = System.nanoTime() - startTime;

```

```

        fileInputStream.close();
        System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
        System.out.printf("%.2f s", duration / 1000000000.0);
    } catch (IOException e) {
        System.out.println(e);
    }
}
}

```

Результат:

```

cycle counter: 5000000
0x31 0x31 ...
2.35 s

```

Для оцінки різниці у швидкості доступу прочитаємо файл блоками, а не побайтно:

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        byte[] text = new byte[5000000];
        try {
            BufferedInputStream bufferedInputStream =
                new BufferedInputStream(
                    new FileInputStream("test.txt")
                );
            final long startTime = System.nanoTime();
            int bytesRead = bufferedInputStream.read(text);
            final long duration = System.nanoTime() - startTime;
            System.out.println("bytes read: " + bytesRead);
            bufferedInputStream.close();
            System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
            System.out.printf("%.3f s", duration / 1000000000.0);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

```

0x31 0x31 ...
0.003 s

```

Легко помітити, що швидкість побайтового читання та буферизованого відрізняється на декілька порядків.

Якщо програма постійно читає невеликі об'єми даних, `BufferedInputStream` значно покращує продуктивність. Кожен запит на читання над небуферизованим потоком

виливається у системний запит до операційної системи на читання певної кількості байт. Таким чином може виникати значна перевитрата ресурсів операційної системи. Буферизований потік зменшує кількість таких запитів шляхом читання одного відносно великого (наприклад, 8192 байт) блоку даних у внутрішній буфер, після чого програма отримує дані з цього буфера. Таким чином, значно скорочується кількість системних запитів.

Проте, якщо розмір блоку, котрий має бути прочитаний, більше за розмір буфера, використання буферизованого потоку зменшуватиме продуктивність.

Особливістю класу `BufferedInputStream` є можливість зберегти за допомогою методу `mark(int readlimit)` поточну позицію у буфері `pos` і вказати кількість байт `readlimit`, котрі можуть бути прочитані у прямому напрямі, після чого за допомогою методу `reset()` повернутися до збереженої позиції і повторно прочитати байти з буфера. Потік не гарантує таку можливість, якщо з потоку прочитано більше за `readlimit` байт, починаючи з позиції маркера `pos`. Для кращого розуміння розглянемо частково реалізацію класу `BufferedInputStream` (java version "1.8.0\_77"):

```
private void fill() throws IOException {
    byte[] buffer = getBufIfOpen();
    if (markpos < 0)
        pos = 0;          /* no mark: throw away the buffer */
    else if (pos >= buffer.length) /* no room left in buffer */
        if (markpos > 0) { /* can throw away early part of the buffer */
            int sz = pos - markpos;
            System.arraycopy(buffer, markpos, buffer, 0, sz);
            pos = sz;
            markpos = 0;
        } else if (buffer.length >= marklimit) {
            markpos = -1; /* buffer got too big, invalidate mark */
            pos = 0;      /* drop buffer contents */
        } else if (buffer.length >= MAX_BUFFER_SIZE) {
            throw new OutOfMemoryError("Required array size too large");
        } else {          /* grow buffer */
            int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
                pos * 2 : MAX_BUFFER_SIZE;
            if (nsz > marklimit)
                nsz = marklimit;
            byte nbuf[] = new byte[nsz];
            System.arraycopy(buffer, 0, nbuf, 0, pos);
            if (!bufUpdater.compareAndSet(this, buffer, nbuf)) {
                // Can't replace buf if there was an async close.
                // Note: This would need to be changed if fill()
                // is ever made accessible to multiple threads.
                // But for now, the only way CAS can fail is via close.
                // assert buf == null;
                throw new IOException("Stream closed");
            }
            buffer = nbuf;
        }
    count = pos;
    int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
    if (n > 0)
```

```

        count = n + pos;
    }
    public synchronized int read() throws IOException {
        if (pos >= count) {
            fill();
            if (pos >= count)
                return -1;
        }
        return getBufIfOpen()[pos++] & 0xff;
    }
    public synchronized void mark(int readlimit) {
        marklimit = readlimit;
        markpos = pos;
    }
    public synchronized void reset() throws IOException {
        getBufIfOpen(); // Cause exception if closed
        if (markpos < 0)
            throw new IOException("Resetting to invalid mark");
        pos = markpos;
    }
}

```

Метод `reset()` генерує виняткову ситуацію, якщо `markpos < 0` (насправді, `-1` – це значення, котре присвоюється змінній у випадкові, якщо неможливо зберегти позицію маркера або він поки що не встановлений взагалі).

Метод `mark(int readlimit)` ініціалізує поле класу `marklimit` числом, величина котрого відповідає кількості байт, у межах якої гарантовано можливий відкат до позиції маркера (позиція маркера є поточною позицією у файлі на момент встановлення маркера). За межами `marklimit` таке повернення не гарантується. Нижче буде розглянуто, чому саме.

Метод `read()` або повертає значення елемента буфера `buf` з індексом `pos` у випадкові, якщо ще не досягнуто індекса останнього прочитаного до буфера елемента (після цього значення індекса інкрементують), або намагається перезаповнити буфер (метод `fill()`) новими елементами з основного вхідного потоку, що лежить в основі такого введення. Перезаповнення буфера відбувається у випадкові, коли `pos >= count` (згідно з наведеним кодом). Якщо після спроби викликати метод `fill()` ситуація `pos >= count` не міняється, метод `read()` повертає `-1`, що означає досягнення кінця файла.

Розглянемо роботу методу `fill()`. Для цього проаналізуємо деякі характерні варіанти заповнення буфера.

pos	-	0								buf.length
markpos	-1									

Рисунок – Стан полів до першого завантаження буфера або повне оновлення буфера із неможливістю відкату до маркера

pos	-	0								buf.length
markpos	-1									

Рисунок – Стан полів відповідає прочитаному останньому елементу буфера, маркер – не встановлено



У першому варіанті не прочитано жодного елемента, маркер не встановлено, тому читання наступного байта з потоку не викликає труднощів.

Другий варіант відрізняється від першого тим, що буфер заповнено повністю ( $pos = buf.length$ ). Позиція у буфері виходить за його межі. Маркер – не встановлено. За потреби прочитати наступний байт слід обнулити поле позиції у буфері  $i$ , фактично, перейти до першого варіанту.

Якщо маркер не встановлено, відбувається чергування першого та другого описаних станів із усіма проміжними варіантами, котрі не наведено. На кількість заповнених елементів у буфері вказує поле  $count$  (більше на 1 за індекс останнього заповненого елемента).

Алгоритм використання буфера ускладнюється, якщо встановлено маркер.

При цьому існують два важливих варіанти. Один із них: маркер знаходиться між початком та кінцем буфера. Необхідно прочитати деяку кількість байт. При цьому можна без шкоди для даних, захищених маркером, перезаписати елементи буфера, що знаходяться зліва від маркера, починаючи із нульового (зсунувши буфер уліво). Поля:  $pos = pos - markpos$ ;  $markpos = 0$ . Напряму зсуву показано на рисунку стрілками:  $\leftarrow$ .

pos	-	0								buf.length
markpos	-1	$\leftarrow$				$\leftarrow$				

Рисунок – Стан полів відповідає прочитаному останньому елементу буфера, позиція маркера – між початком та кінцем буфера

Інший варіант характерний тим, що позиція маркера  $markpos$  дорівнює 0. Варіант розгалужується у залежності від значення  $marklimit$ . Величина  $marklimit$  має або скінченне у межах даного буфера значення, або виходить за його межі.

pos	-	0								buf.length
markpos		0								
marklimit							$\rightarrow$			max

Рисунок – Стан полів відповідає прочитаному останньому елементу буфера, маркер знаходиться на початку буфера,  $marklimit$  вказує на елемент усередині буфера

З рисунка видно, що не існує жодної можливості зсунути елементи буфера уліво. У будь-якому випадку елементи, пам'ятати значення котрих зобов'язує маркер, буде втрачено.

Якщо ж  $marklimit$  вказує на елемент поза межами буфера (тобто, розмір буфера менший за кількість елементів, пам'ятати котрі зобов'язує маркер), розмір буфера намагаються збільшити удвічі (змінна  $psz$  у класі). У випадку, якщо значення змінної  $psz$  більше за  $marklimit$ , розмір буфера обмежують величиною  $marklimit$ . Описане повторюють щоразу при виклику методу  $fill()$ .

index	-	0								buf.length
pos	-									buf.length
markpos		0								
marklimit										$\rightarrow$

Рисунок – Прочитано останній елемент буфера, маркер знаходиться на початку буфера,  $marklimit$  вказує на елемент поза межами буфера

Наведений нижче код ілюструє алгоритм збільшення розміру буфера. На початкові перевіряють можливість збільшення розміру буфера удвічі.

```
int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
    pos * 2 : MAX_BUFFER_SIZE;
if (nsz > marklimit)
    nsz = marklimit;
```

Зверніть увагу на те, що розмір буфера вказано непрямо, замість `buf.length` використано `pos`. При цьому `pos` із формулювання задачі більший за індекс останнього елемента і за визначенням не може бути більшим ніж на 1, оскільки у такому випадкові мав би бути прочитаним ще один байт, а його фізично нікуди прочитати на даному етапі.

Розглянемо приклад. Підготуємо файл, у якому перший байт відрізняється від усіх інших. Такий підхід зручний, якщо потрібно продемонструвати повторне читання.

```
import java.io.*;
import java.util.Arrays;
public class Test {
    public static void main(String[] args) {
        byte[] text = new byte[5000000];
        Arrays.fill(text, (byte)0x31);
        text[0] = 0x35;
        try{
            BufferedOutputStream bufferedOutputStream =
                new BufferedOutputStream(
                    new FileOutputStream("test.txt")
                );
            bufferedOutputStream.write(text);
            bufferedOutputStream.close();
        } catch ( IOException e){
            e.printStackTrace();
        }
    }
}
```

Читаємо файл за допомогою буферизованого вхідного потоку.

```
import java.io.*;
public class Test extends BufferedInputStream {
    public static void main(String[] args) {
        byte[] text = new byte[8192];
        try {
            Test bufferedInputStream =
                new Test(
                    new FileInputStream("test.txt"));
            System.out.println("mark supported = " +
```

```

        bufferedInputStream.markSupported());
    bufferedInputStream.printPos();
    bufferedInputStream.mark(100);
    bufferedInputStream.printPos();
    bufferedInputStream.read(text, 0, 8192);
    bufferedInputStream.printPos();
    // bufferedInputStream.read();
    // bufferedInputStream.printPos();
    bufferedInputStream.reset();
    bufferedInputStream.printPos();
    bufferedInputStream.close();
    System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
} catch (IOException e) {
    e.printStackTrace();
}
}
}
public Test(InputStream i) {
    super(i);
}
int getPos() {
    return super.pos;
}
int getMarkPos(){
    return super.markpos;
}
int getCount() {return super.count;}
int getMarkLimit() {return super.marklimit; }
int getBufferLength() {return super.buf.length; }
void printPos(){
    System.out.println("pos = " + getPos());
    System.out.println("markpos = " + getMarkPos());
    System.out.println("count = " + getCount());
    System.out.println("marklimit = " + getMarkLimit());
    System.out.println("buffer.length = " + getBufferLength());
    System.out.println("-----");
}
}
}

```

Результат:

```

mark supported = true
pos = 0
markpos = -1
count = 0
marklimit = 0
buffer.length = 8192
-----
pos = 0
markpos = 0
count = 0

```

```

marklimit = 100
buffer.length = 8192
-----
pos = 8192
markpos = 0
count = 8192
marklimit = 100
buffer.length = 8192
-----
0x35 0x31 ...
pos = 0
markpos = 0
count = 8192
marklimit = 100
buffer.length = 8192
-----
0x35 0x31 ...

```

Видно, що на початкові усі поля-індекси класу дорівнюють 0, маркер – не встановлено. На другому кроці встановлюють маркер і поле marklimit набуває значення 100.

Наступний крок: читання 8192 байт з вхідного потоку. При цьому pos набуває значення 8192, count – 8192 (на 1 більше за індекс останнього значущого елемента буфера), marklimit – зберігає своє значення 100 незмінним.

Далі виконують метод reset(). Метод успішно завершується поверненням до маркера, встановленим на нульовій позиції (pos: 0, markpos: 0). Можна повторно читати елементи потоку (насправді – буфера).

Розкоментуйте закоментовані рядки прикладу. Ситуація цілком зміниться: при повністю заповненому буфері і нульовій позиції маркера та значенні marklimit, що не перекриває буфер повністю, прочитати ще один байт, не пошкодивши буфер немає можливості.

```

mark supported = true
pos = 0
markpos = -1
count = 0
marklimit = 0
buffer.length = 8192
-----
pos = 0
markpos = 0
count = 0
marklimit = 100
buffer.length = 8192
-----
pos = 8192
markpos = 0
count = 8192
marklimit = 100
buffer.length = 8192
-----
0x35 0x31 ...
pos = 1

```

```
markpos = -1
count = 8192
marklimit = 100
buffer.length = 8192
-----
java.io.IOException: Resetting to invalid mark
```

Якщо прибрати з тексту програми виклик `reset()`, то байти з потоку успішно читатимуться далі. Зрозуміло, що повернутися до маркера буде неможливо, хоча `marklimit` і встановлено.

Змінимо параметр методу `mark(100)` на `mark(10000)`. Таким чином, слід запам'ятати кількість байт, явно більшу за поточний розмір буфера.

```
mark supported = true
pos = 0
markpos = -1
count = 0
marklimit = 0
buffer.length = 8192
-----
pos = 0
markpos = 0
count = 0
marklimit = 10000
buffer.length = 8192
-----
pos = 8192
markpos = 0
count = 8192
marklimit = 10000
buffer.length = 8192
-----
0x35 0x31 ...
pos = 8193
markpos = 0
count = 10000
marklimit = 10000
buffer.length = 10000
-----
pos = 0
markpos = 0
count = 10000
marklimit = 10000
buffer.length = 10000
-----
0x35 0x31 ...
```

З результату видно, що розмір буфера збільшено таким чином, що він відповідає параметрові `marklimit`. Також показано, що у випадкові успішного виконання методу `reset()` відбувається перехід на позицію встановленого раніше маркера (під час читання отримують ті самі байти).

## Класи [DataOutputStream](#) та [DataInputStream](#)

Розглянуті вище класи працюють з байтами. [DataOutputStream](#) ([DataInputStream](#)) надає можливість записувати (читати) дані базових типів Java до вихідного (вхідного) потоку у бінарному вигляді. Прочитати їх можна за допомогою [DataInputStream](#).

### DataOutputStream

#### Поля

protected int	<a href="#">written</a> Кількість байт, записаних до data output stream на даний момент
---------------	--

#### Конструктор

##### [DataOutputStream](#)([OutputStream](#) out)

Створює новий потік data output stream для запису даних у вихідний потік out.

#### Методи

void	<a href="#">flush()</a> Вивільняє data output stream
int	<a href="#">size()</a> Повертає поточне значення лічильника written – кількість байт, записаних до data output stream на даний час
void	<a href="#">write</a> (byte[] b, int off, int len) Записує len байт з масиву байт зі зміщенням off у вихідний потік, котрий лежить в основі
void	<a href="#">write</a> (int b) Записує байт (молодші 8 біт аргумента) параметра у вихідний потік, котрий лежить в основі
void	<a href="#">writeBoolean</a> (boolean v) Записує boolean у вихідний потік, котрий лежить в основі, як 1-байт
void	<a href="#">writeByte</a> (int v) Записує байт (молодші 8 біт аргумента) параметра у вихідний потік, котрий лежить в основі
void	<a href="#">writeBytes</a> (String s) Записує рядок як послідовність байт у потік, що лежить в основі. Кожен символ рядка записують, відкидаючи його старші 8 біт
void	<a href="#">writeChar</a> (int v) Записує символ у потік, що лежить в основі, як 2 – байтове значення, першим – старший байт
void	<a href="#">writeChars</a> (String s) Записує рядок у потік, що лежить в основі, як послідовність символів
void	<a href="#">writeDouble</a> (double v) Перетворює аргументи типу double у long з використанням методу doubleToLongBits класу Double, після чого записує long до потоку, що лежить в основі, як 8 байт. Першим записують старший байт

void	<a href="#">writeFloat(float v)</a> Перетворює аргументи типу float у int з використанням методу doubleToIntBits класу Float, після чого записує int до потоку, що лежить в основі, як 4 байти. Першим записують старший байт
void	<a href="#">writeInt(int v)</a> Записує int до потоку, що лежить в основі, як 4 байти. Першим записують старший байт
void	<a href="#">writeLong(long v)</a> Записує long до потоку, що лежить в основі, як 8 байт. Першим записують старший байт
void	<a href="#">writeShort(int v)</a> Записує short до потоку, що лежить в основі, як 2 байти. Першим записують старший байт
void	<a href="#">writeUTF(String str)</a> Записує рядок до потоку, що лежить в основі, за допомогою модифікованого <a href="#">modified UTF-8</a> кодування

Зверніть увагу на методи `write(int b)` та `writeByte(int v)`. Функціонально вони однакові. Обидва записують у потік молодші 8 біт аргумента. Проте, `writeByte` – final (не може бути перевизначений дочірнім класом) та `unsynchronized`, а `write` – не є final та є `synchronized`.

## DataInputStream

### Конструктор

#### [DataInputStream\(InputStream in\)](#)

Створює `DataInputStream`, котрий використовує заданий `InputStream`.

### Методи

int	<a href="#">read(byte[] b)</a> Читає деяку кількість байт з вхідного потоку і зберігає їх у масиві b
int	<a href="#">read(byte[] b, int off, int len)</a> Читає до len байт з вхідного потоку і зберігає їх у масиві b зі зміщенням off
boolean	<a href="#">readBoolean()</a> Читає байт з вхідного потоку. Якщо значення байта – одиниця – повертає true, інакше – false. Парний метод – <code>writeBoolean()</code>
byte	<a href="#">readByte()</a> Читає байт з вхідного потоку, розглядаючи його як величину у межах від -128 до +127. Парний метод - <code>writeByte()</code>
char	<a href="#">readChar()</a> Читає char (2 байти) з вхідного потоку. Парний метод - <code>writeChar()</code>
double	<a href="#">readDouble()</a> Читає 8 байт з вхідного потоку і повертає double. Парний метод - <code>writeDouble()</code>
float	<a href="#">readFloat()</a> Читає 4 байти з вхідного потоку і повертає float. Парний метод – <code>writeFloat()</code>
void	<a href="#">readFully(byte[] b)</a> Читає деяку кількість байт з вхідного потоку та зберігає їх у масиві b. Кількість

	байт дорівнює розмірові масиву b
void	<a href="#">readFully</a> (byte[] b, int off, int len) Читає з вхідного потоку строго len елементів та зберігає їх у масиві b зі зміщенням off
int	<a href="#">readInt</a> () Читає з вхідного потоку 4 байти і повертає еквівалентний int. Парний метод - writeInt()
<a href="#">String</a>	<a href="#">readLine</a> () Застарілий Слід використовувати клас <code>BufferedReader</code>
Long	<a href="#">readLong</a> () Читає з вхідного потоку 8 байт і повертає еквівалентний Long. Парний метод – writeLong()
short	Читає з вхідного потоку 2 байт і повертає еквівалентний short. Парний метод – writeShort()
int	<a href="#">readUnsignedByte</a> () Метод читає 1 байт, розтягує його за допомогою нулів у старших розрядах до типу int і повертає результат у діапазоні 0 – 255
int	<a href="#">readUnsignedShort</a> () Читає 2 байти з вхідного потоку і повертає результат у діапазоні 0 – 65535
<a href="#">String</a>	<a href="#">readUTF</a> () Читає рядок у <a href="#">модифікованому UTF-8</a> форматі
static <a href="#">String</a>	<a href="#">readUTF</a> ( <a href="#">DataInput</a> in) Статичний метод. Читає рядок у <a href="#">модифікованому UTF-8</a> форматі з потоку in та повертає рядок символів як String
int	<a href="#">skipBytes</a> (int n) Пропускає n байт у вхідному потокові

Методи, успадковані від `java.io.FilterInputStream`: [available](#), [close](#), [mark](#), [markSupported](#), [read](#), [reset](#), [skip](#).

Методи, успадковані від `java.lang.Object`: [clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#).

Метод `close()` класу `DataOutputStream` викликає метод `flush()` цього класу, після чого викликає метод `close()` потоку, що є параметром конструктора. Метод `close()` класу `DataInputStream` викликає метод `close()` потоку, що є параметром конструктора.

Розглянемо кілька прикладів.

```
import java.io.*;
import java.util.Arrays;
public class Test {
    public static void main(String[] args) {
        try {
            byte[] arr = {1,2,3};
            DataOutputStream dataOutputStream =
                new DataOutputStream(new FileOutputStream("test.txt"));
            dataOutputStream.write(arr);
        }
    }
}
```



```

dataOutputStream.write(arr, 1, arr.length - 1);
dataOutputStream.writeFloat(1.4f);
dataOutputStream.writeDouble(5.6);
dataOutputStream.writeBoolean(true);
dataOutputStream.write(-1);
dataOutputStream.writeByte(-127);
dataOutputStream.writeShort(32767);
dataOutputStream.writeInt(32768);
dataOutputStream.writeLong(2147483648L);
dataOutputStream.close();

DataInputStream dataInputStream =
    new DataInputStream(new FileInputStream("test.txt"));
Arrays.fill(arr, (byte)0);
dataInputStream.read(arr);
System.out.println(Arrays.toString(arr));
Arrays.fill(arr, (byte)0);
dataInputStream.read(arr, 1, 2);
System.out.println(Arrays.toString(arr));
System.out.println(dataInputStream.readFloat());
System.out.println(dataInputStream.readDouble());
System.out.println(dataInputStream.readBoolean());
System.out.println(dataInputStream.read());
System.out.println(dataInputStream.readByte());
System.out.println(dataInputStream.readShort());
System.out.println(dataInputStream.readInt());
System.out.println(dataInputStream.readLong());
dataInputStream.close();
} catch (IOException e){
    e.printStackTrace();
}
}
}

```

Результат:

```

[1, 2, 3]
[0, 2, 3]
1.4
5.6
true
255
-127
32767
32768
2147483648

```

Зверніть увагу на невідповідний, на перший погляд, результат виконання read(). За допомогою write(int) записано число -1. А прочитано воно як 255. Цілі типи характерні тим,

що старший двійковий розряд числа, піднятий до ступеня з показником, який відповідає номерів розряду, у випадкові від'ємних значень обчислюють зі знаком мінус. Наприклад:

Номер роздряду	3	2	1	0
Біт	1	1	1	1
Значення	$-(2^3)$	$2^2$	$2^1$	$2^0$

Отже, двійкове 1111 число у Java для 4-розрядного цілого типу позначає десяткове -1. Від кількості розрядів значення не змінюється у випадкові, якщо усі розряди – одиниці. Фактично, `-1 int = 11111111111111111111111111111111`. Молодші 8 біт показано на чорному тлі. При записові у потік метод `write()` ігнорує старші 24 біти і запише 8 молодших. Метод `read()` читає байт і повертає результат як `int` у діапазоні від 0 до 255. Якщо немає доступних для читання байт – метод повертає -1.

Таким чином, двійкові 11111111 метод розглядає як десяткове 255.

Натомість, метод `readByte()` розглядає байт як число у діапазоні від -128 до +127. Метод `readByte()` придатний для роботи у парі з методом `writeByte()`.

Метод `writeChar(int v)` записує `char` до потоку у вигляді 2 байт, першим – старший байт.

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            int i = 0x31; // ASCII код цифри "1", відповідає молодшому байтові utf-16
            DataOutputStream dataOutputStream =
                new DataOutputStream(new FileOutputStream("test.txt"));
            dataOutputStream.writeChar(i);
            dataOutputStream.close();
            DataInputStream dataInputStream =
                new DataInputStream(new FileInputStream("test.txt"));
            System.out.println(dataInputStream.readChar());
            dataInputStream.close();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Результат:

1

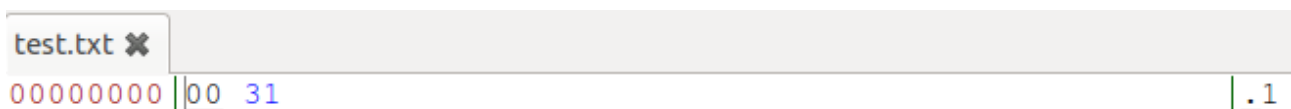


Рисунок – збереження char. Оскільки символ “1” входить до таблиці ASCII, у utf-16 старший байт – нульовий.

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

```

public class Test {
    public static void main(String[] args) {
        try {
            DataOutputStream dataOutputStream =
                new DataOutputStream(new FileOutputStream("test.txt"));
            String s = "привіт hello";
            int l = s.length();
            dataOutputStream.writeInt(l);
            dataOutputStream.writeChars(s);
            dataOutputStream.close();
            DataInputStream dataInputStream =
                new DataInputStream(new FileInputStream("test.txt"));
            int n = dataInputStream.readInt();
            for(int i = 0; i < n; i++)
                System.out.print(dataInputStream.readChar());
            dataInputStream.close();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}

```

Результат: привіт hello

test.txt ✕	
00000000	00 00 00 0C 04 3F 04 40 04 38 04 32 04 56 04 42 00 20   .....?.@.8.2.V.B.
00000012	00 68 00 65 00 6C 00 6C 00 6F   .h.e.l.l.o

Рисунок – збереження String за допомогою writeChars(String s)

Зверніть увагу на те, що до файлу додатково додається інформація про кількість записаних char. Методу readChars() не існує, оскільки не відомо, скільки байт було записано попередньо.

Дане зроблено з метою демонстрації. Аналогічного результату можна досягнути, скориставшись методом available і поділивши результат на 2 (підставою є те, що кожен символ займає 2 байти).

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            DataOutputStream dataOutputStream =
                new DataOutputStream(new FileOutputStream("test.txt"));
            String s = "привіт hello";
            dataOutputStream.writeBytes(s);
            dataOutputStream.close();
        }
    }
}

```

```

        DataInputStream dataInputStream =
            new DataInputStream(new FileInputStream("test.txt"));
        int n = dataInputStream.available();
        for(int i = 0; i < n; i++)
            System.out.printf("%h ", dataInputStream.read());
        dataInputStream.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Результат:

3f 40 38 32 56 42 20 68 65 6c 6c 6f

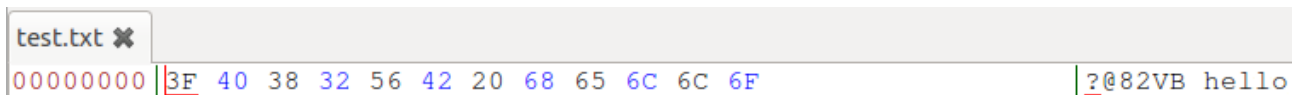


Рисунок – Збереження даних за допомогою writeBytes()

Оскільки попередньо було записано по 1 байтові до потоку, кількість

У попередньому прикладові символи кирилиці кодовані 2 байтами (так, як і латинські літери, проте для латинських літер з ASCII кодами до 128 старший байт дорівнює 0). Тобто, відновити кирилицю – можливо.

Даний приклад характерний тим, що старший байт при збереженні – відкинуто.

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            DataOutputStream dataOutputStream =
                new DataOutputStream(new FileOutputStream("test.txt"));
            String s = "привіт hello";
            dataOutputStream.writeUTF(s);
            dataOutputStream.close();
            DataInputStream dataInputStream =
                new DataInputStream(new FileInputStream("test.txt"));
            System.out.println(dataInputStream.readUTF());
            dataInputStream.close();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}

```

Результат:

привіт hello

```
test.txt x
00000000 | 00 12 D0 BF D1 80 D0 B8 D0 B2 D1 96 D1 82 20 68 65 6C | ..... hel
00000012 | 6C 6F | lo
```

Перші два байти – розмір даних, що записані у форматі modified UTF-8.  $12^{(16)}=18^{(2)}$ . Розмір файлу – 20 байт, з них 2 – заголовок текстового блоку, 18 – текст.

## ASCII control characters (character code 0-31)

Перших 32 символи таблиці ASCII – недруковані керуючі символи, їх використовують для контролю периферії. Такої, як принтери.

Dec	Hex	Symbol	Description	Dec	Hex	Symbol	Description
0	00	NUL	Null char	16	10	DLE	Data Line Escape
1	01	SOH	Start of Heading	17	11	DC1	Device Control 1 (oft. XON)
2	02	STX	Start of Text	18	12	DC2	Device Control 2
3	03	ETX	End of Text	19	13	DC3	Device Control 3 (oft. XOFF)
4	04	EOT	End of Transmission	20	14	DC4	Device Control 4
5	05	ENQ	Enquiry	21	15	NAK	Negative Acknowledgement
6	06	ACK	Acknowledgment	22	16	SYN	Synchronous Idle
7	07	BEL	Bell	23	17	ETB	End of Transmit Block
8	08	BS	Back Space	24	18	CAN	Cancel
9	09	HT	Horizontal Tab	25	19	EM	End of Medium
10	0A	LF	Line Feed	26	1A	SUB	Substitute
11	0B	VT	Vertical Tab	27	1B	ESC	Escape
12	0C	FF	Form Feed	28	1C	FS	File Separator
13	0D	CR	Carriage Return	29	1D	GS	Group Separator
14	0E	SO	Shift Out / X-On	30	1E	RS	Record Separator
15	0F	SI	Shift In / X-Off	31	1F	US	Unit Separator

## ASCII printable characters (character code 32-127)

Коди 32-127 є спільними для усіх варіантів ASCII-таблиць, їх називають друкованими.

Dec	Hex	Символ	Опис	Dec	Hex	Символ	Опис
32	20		Space	80	50	P	
33	21	!	Exclamation mark	81	51	Q	
34	22	"	Double quotes (or speech marks)	82	52	R	
35	23	#	Number	83	53	S	
36	24	\$	Dollar	84	54	T	
37	25	%	Procenttecken	85	55	U	
38	26	&	Ampersand	86	56	V	
39	27	'	Single quote	87	57	W	
40	28	(	Open parenthesis (or open bracket)	88	58	X	
41	29	)	Close parenthesis (or close bracket)	89	59	Y	
42	2A	*	Asterisk	90	5A	Z	
43	2B	+	Plus	91	5B	[	Opening bracket
44	2C	,	Comma	92	5C	\	Backslash
45	2D	-	Hyphen	93	5D	]	Closing bracket
46	2E	.	Period, dot or full stop	94	5E	^	Caret - circumflex
47	2F	/	Slash or divide	95	5F	_	Underscore
48	30	0	Zero	96	60	`	Grave accent
49	31	1	One	97	61	a	
50	32	2	Two	98	62	b	
51	33	3	Three	99	63	c	
52	34	4	Four	100	64	d	
53	35	5	Five	101	65	e	
54	36	6	Six	102	66	f	
55	37	7	Seven	103	67	g	
56	38	8	Eight	104	68	h	
57	39	9	Nine	105	69	i	
58	3A	:	Colon	106	6A	j	
59	3B	;	Semicolon	107	6B	k	
60	3C	<	Less than (or open angled bracket)	108	6C	l	
61	3D	=	Equals	109	6D	m	
62	3E	>	Greater than (or close angled bracket)	110	6E	n	
63	3F	?	Question mark	111	6F	o	
64	40	@	At symbol	112	70	p	

Dec	Hex	Символ	Опис	Dec	Hex	Символ	Опис
65	41	A		113	71	q	
66	42	B		114	72	r	
67	43	C		115	73	s	
68	44	D		116	74	t	
69	45	E		117	75	u	
70	46	F		118	76	v	
71	47	G		119	77	w	
72	48	H		120	78	x	
73	49	I		121	79	y	
74	4A	J		122	7A	z	
75	4B	K		123	7B	{	Opening brace
76	4C	L		124	7C		Vertical bar
77	4D	M		125	7D	}	Closing brace
78	4E	N		126	7E	~	Equivalency sign - tilde
79	4F	O		127	7F		Delete

### [OutputStreamWriter](#)

OutputStreamWriter перетворює потік символів у потік байт з використанням вказаного кодування ([charset](#)). Кодування може бути вказано за його іменем, у явному вигляді або буде прийнято кодування за замовчуванням даної платформи.

Кожний виклик методу write() включає виклик конвертера над переданими йому символами. Отримані байти акумулюються у буфері перед записом до вихідного потоку. Символи, передані методів write(), не акумулюються.

Для максимальної ефективності зручно обгорнути OutputStreamWriter за допомогою BufferedWriter для уникнення частих викликів конвертера. Наприклад:

```
Writer out = new BufferedWriter(new OutputStreamWriter(System.out));
```

Поля

[lock](#) – об'єкт, що використовують для синхронізації операцій над потоком

Конструктори

<a href="#">OutputStreamWriter</a> ( <a href="#">OutputStream</a> out)
Створє OutputStreamWriter із кодуванням за замовчуванням
<a href="#">OutputStreamWriter</a> ( <a href="#">OutputStream</a> out, <a href="#">Charset</a> cs)
Створює OutputStreamWriter, що використовує кодування-параметр
<a href="#">OutputStreamWriter</a> ( <a href="#">OutputStream</a> out, <a href="#">CharsetEncoder</a> enc)
Створює OutputStreamWriter, що використовує вказаний кодер
<a href="#">OutputStreamWriter</a> ( <a href="#">OutputStream</a> out, <a href="#">String</a> charsetName)
Створює OutputStreamWriter, що використовує кодування, задане іменем

## Методи

void	<a href="#">close()</a> Закриває потік, вивільняючи його перед цим
void	<a href="#">flush()</a> Вивільняє потік від даних
<a href="#">String</a>	<a href="#">getEncoding()</a> Повертає ім'я кодування, котре використовує потік
void	<a href="#">write(char[] cbuf, int off, int len)</a> Записує з масиву char[] cbuf у потік len символів зі зміщенням від початку масиву off
void	<a href="#">write(int c)</a> Записує char у потік.
void	<a href="#">write(String str, int off, int len)</a> Записує у потік підрядок рядка str

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            OutputStreamWriter osr;
            Writer w = new BufferedWriter(
                osr = new OutputStreamWriter(
                    new BufferedOutputStream(
                        new FileOutputStream("test.txt")),
                    "Cp1251"
                )
            );
            w.write("привіт hello");
            System.out.println("encoding: " + osr.getEncoding());
            w.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Зверніть увагу на те, що у даному випадкові OutputStreamWriter – не анонімний клас. Змінну його оголошено попередньо, інакше не можна було б отримати доступ до методу getEncoding().

Результат:

encoding: Cp1251

Інформацію про кодування можна знайти [тут](#)



Рисунок – збереження даних за допомогою  
OutputStreamWriter (кодування – “Windows-1251”)

Якщо змінити “Windows-1251” на “UTF-8”, отримаємо наступний результат:

Рисунок – збереження даних за допомогою  
OutputStreamWriter (кодування – “Windows-1251”)

Зверніть увагу на те, що латинка кодується 1 байтом на символ.

### [InputStreamReader](#)

InputStreamReader поєднує потік байт з потоком символів. Потік читає байти, декодує їх у char, використовуючи задане кодування. Кодування може бути вказано за іменем, у явному вигляді або буде використано кодування за замочуванням.

Кожен виклик методу read() класу InputStreamReader вимагає отримання одного чи більше байт з байтового потоку, що лежить в основі читання. Для забезпечення ефективного декодування байт у символи може бути наперед прочитано більше байт, ніж потрібно на даний момент.

Максимальної ефективності можна досягнути шляхом огортання InputStreamReader за допомогою BufferedReader:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

Поле класу

[lock](#) – використовують для синхронізації операцій над потоком.

Конструктори

<u><a href="#">InputStreamReader(InputStream in)</a></u> Creates an InputStreamReader that uses the default charset.
<u><a href="#">InputStreamReader(InputStream in, Charset cs)</a></u> Creates an InputStreamReader that uses the given charset.
<u><a href="#">InputStreamReader(InputStream in, CharsetDecoder dec)</a></u> Creates an InputStreamReader that uses the given charset decoder.
<u><a href="#">InputStreamReader(InputStream in, String charsetName)</a></u> Creates an InputStreamReader that uses the named charset.

## Методи

void	<a href="#">close()</a> Closes the stream and releases any system resources associated with it.
<a href="#">String</a>	<a href="#">getEncoding()</a> Returns the name of the character encoding being used by this stream.
int	<a href="#">read()</a> Reads a single character.
int	<a href="#">read(char[] cbuf, int offset, int length)</a> Reads characters into a portion of an array.
boolean	<a href="#">ready()</a> Tells whether this stream is ready to be read.

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            OutputStreamWriter isr;
            BufferedWriter w = new BufferedWriter(
                isr = new OutputStreamWriter(
                    new BufferedOutputStream(
                        new FileOutputStream("test.txt"){
                            @Override
                            public void close() throws IOException {
                                super.close();
                                System.out.println("closed");
                            }
                        }
                    ),
                    "Cp1251"
                )
            );
            StringBuilder sb = new StringBuilder();
            sb.append(String.format("%d\n", 1));
            sb.append(String.format("2 x 2 = %d", 2 * 2));
            w.write(sb.toString());
            System.out.println("encoding: " + isr.getEncoding());
            w.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Резултат:

encoding: Cp1251  
closed

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            InputStreamReader isr;
            BufferedReader r = new BufferedReader(
                isr = new InputStreamReader(
                    new BufferedInputStream(
                        new FileInputStream("test.txt"){
                            @Override
                            public void close() throws IOException {
                                super.close();
                                System.out.println("closed");
                            }
                        }
                    ),
                    "Cp1251"
                )
            );
            String line;
            while ((line = r.readLine()) != null)
                System.out.println(line);
            System.out.println("encoding: " + isr.getEncoding());
            r.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Результат:

```

1
2 x 2 = 4
encoding: Cp1251
closed

```

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            FileWriter fileWriter =
                new FileWriter("test.txt");
            final long startTime = System.nanoTime();
            for(int i = 0; i < 5000000; i++)
                fileWriter.write("7");
            final long duration = System.nanoTime() - startTime;
            System.out.printf("%.3f s", duration / 1000000000.0);
        }
    }
}

```

```

        fileWriter.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Результат:

0.479 s

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            BufferedWriter bufferedWriter =
                new BufferedWriter(
                    new FileWriter("test.txt"));
            final long startTime = System.nanoTime();
            for(int i = 0; i < 5000000; i++)
                bufferedWriter.write("д");
            final long duration = System.nanoTime() - startTime;
            System.out.printf("%.3f s", duration / 1000000000.0);
            bufferedWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Результат:

0.218 s

PrintWriter

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            PrintWriter printWriter =
                new PrintWriter("test.txt");
            printWriter.printf("привіт %d", 12345);
            printWriter.close();
        } catch (IOException e) {

```

```

        e.printStackTrace();
    }
}

```

Результат:

```

test.txt x
00000000 D0 BF D1 80 D0 B8 D0 B2 D1 96 D1 82 20 31 32 33 34 35 ..... 12345
00000012

```

Рисунок

```

import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            PrintWriter printWriter =
                new PrintWriter("test.txt", "Cp1251");
            printWriter.printf("привіт %d", 12345);
            printWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Результат:

```

test.txt x
00000000 EF F0 E8 E2 B3 F2 20 31 32 33 34 35 ..... 12345

```

Рисунок