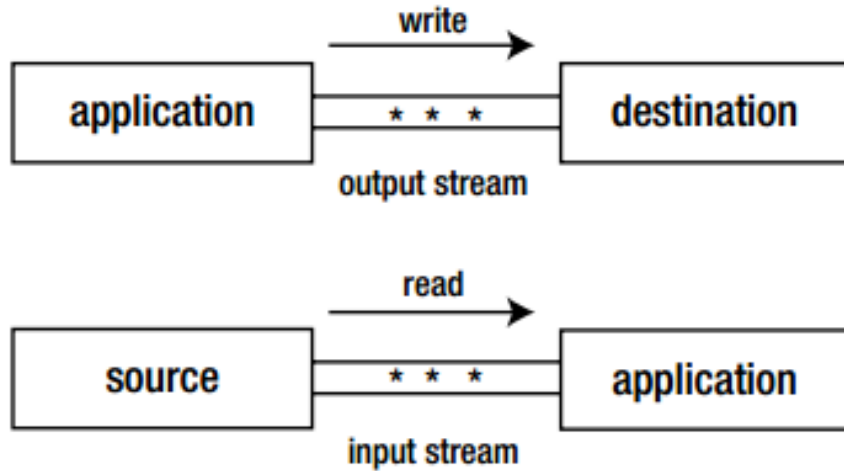




# РОБОТА З ПОТОКАМИ ДАНИХ, РАЙТЕРАМИ ТА РІДЕРАМИ

Питання 5.5.

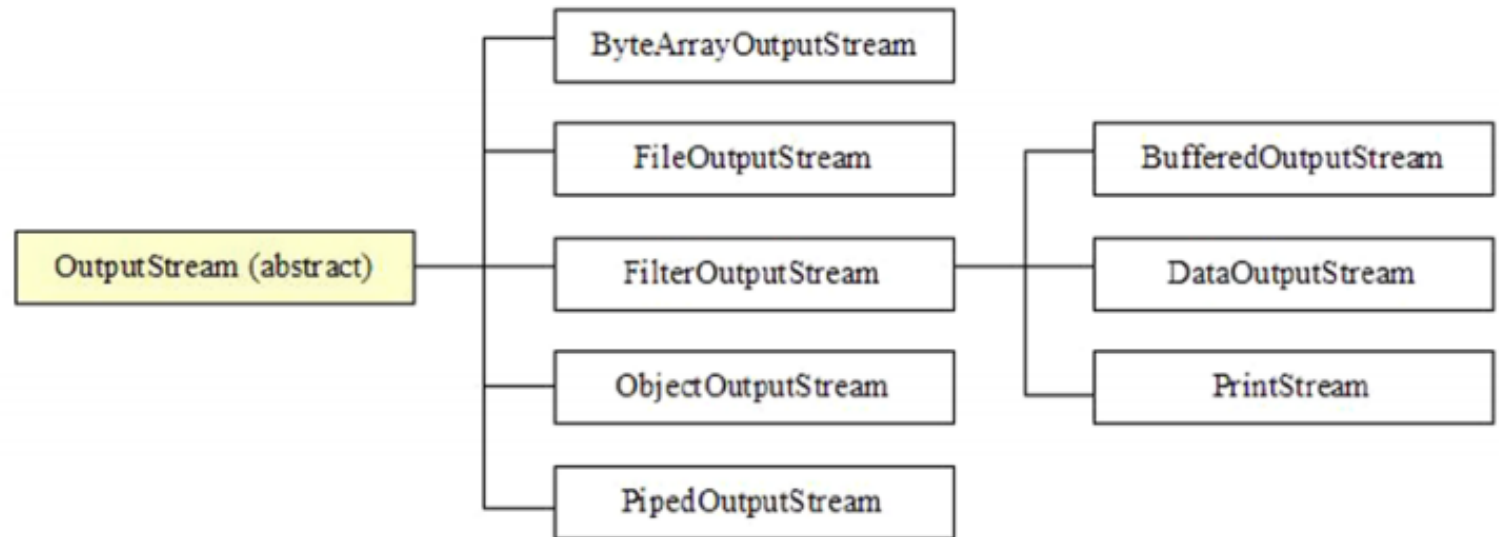
# Поняття потоків виводу та вводу даних



- Пакет `java.io` постачає різні класи для потоків вводу/виводу даних, похідні від абстрактних класів `OutputStream` та `InputStream`.

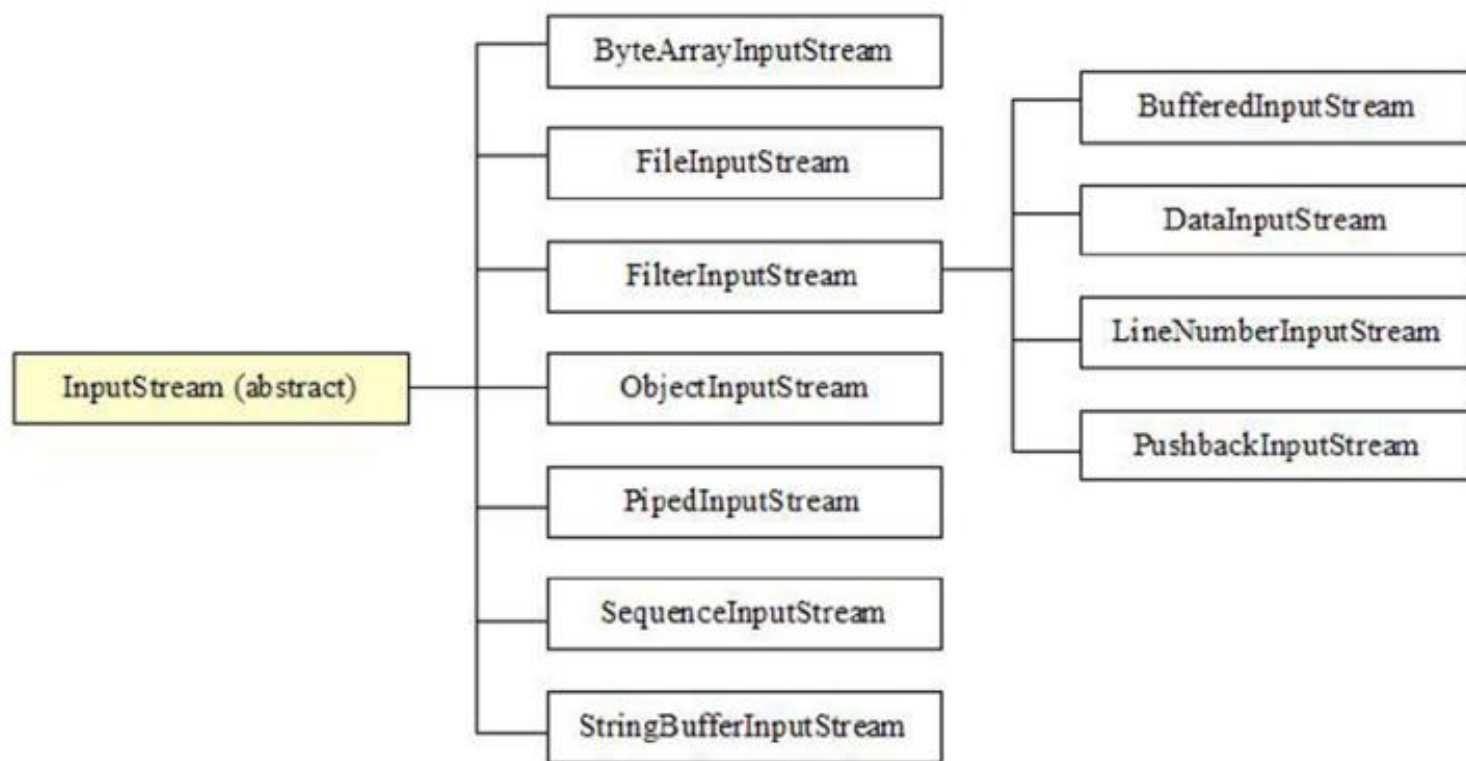
- Всі класи потоків виводу, крім `PrintStream`, мають суфікс `OutputStream` у назві

- Java розпізнає різні stream destinations (байтові масиви, файли, screens, *сокети* та канали (thread pipes)).
- Також Java розрізняє джерела потоку (stream sources) – байтові масиви, файли, клавіатури, сокети та канали.



# Ієрархія класів потоку вводу даних

---



- Класи `LineNumberInputStream` та `StringBufferInputStream` вважаються застарілими
  - Не підтримують різні кодування символів.
  - Їх заміняють класи `LineNumberReader` та `StringReader`.
  - Аналогічно клас `PrintStream` замінюється на `PrintWriter`, проте через часте використання першого у старих кодах, його не роблять застарілим.

# Інші пакети Java постачають додаткові класи для потоків виводу та вводу

---

- `java.util.zip` постачає 4 класи для потоку виводу даних, які стискають нестиснені дані в різні формати + 4 відповідних класи для потоку вводу даних для розпаковки:
  - `CheckedOutputStream`
  - `CheckedInputStream`
  - `DeflaterOutputStream`
  - `GZIPOutputStream`
  - `GZIPInputStream`
  - `InflaterInputStream`
  - `ZipOutputStream`
  - `ZipInputStream`
- Пакет `java.util.jar` постачає пари класів для роботи з потоком даних з метою запису/зчитування контенту JAR-файлу:
  - `JarOutputStream`
  - `JarInputStream`

# Методи класу OutputStream

---

<code>void close()</code>	Closes this output stream and releases any platform resources associated with the stream. This method throws <code>IOException</code> when an I/O error occurs.
<code>void flush()</code>	Flushes this output stream by writing any buffered output bytes to the destination. If the intended destination of this output stream is an abstraction provided by the underlying platform (for example, a file), flushing the stream only guarantees that bytes previously written to the stream are passed to the underlying platform for writing; it doesn't guarantee that they're actually written to a physical device such as a disk drive. This method throws <code>IOException</code> when an I/O error occurs.
<code>void write(byte[] b)</code>	Writes <code>b.length</code> bytes from byte array <code>b</code> to this output stream. In general, <code>write(b)</code> behaves as if you specified <code>write(b, 0, b.length)</code> . This method throws <code>NullPointerException</code> when <code>b</code> is null and <code>IOException</code> when an I/O error occurs.
<code>void write(byte[] b, int off, int len)</code>	Writes <code>len</code> bytes from byte array <code>b</code> starting at offset <code>off</code> to this output stream. This method throws <code>NullPointerException</code> when <code>b</code> is null; <code>java.lang.IndexOutOfBoundsException</code> when <code>off</code> is negative, <code>len</code> is negative, or <code>off + len</code> is greater than <code>b.length</code> ; and <code>IOException</code> when an I/O error occurs.
<code>void write(int b)</code>	Writes byte <code>b</code> to this output stream. Only the 8 low-order bits are written; the 24 high-order bits are ignored. This method throws <code>IOException</code> when an I/O error occurs.

# Методи класу OutputStream

---

- Метод `flush()` корисний у довготривалих додатках, у яких потрібно досить часто зберігати зміни: в тимчасовий файл кожні кілька хвилин.
  - Метод `flush()` лише передає байти платформі; у результаті платформа може і не скинути ці байти на диск.
- Метод `close()` автоматично очищає (`flush`) потік виводу.
  - Якщо додаток закінчує роботу до виклику `close()`, потік виводу даних (`output stream`) автоматично закривається, а його дані **is flushed**.

# Методи InputStream (суперкласу для всіх підкласів для роботи з вхідним потоком даних)

---

`int available()`

Returns an estimate of the number of bytes that can be read from this input stream via the next `read()` method call (or skipped over via `skip()`) without blocking the calling thread. This method throws `IOException` when an I/O error occurs.

It's never correct to use this method's return value to allocate a buffer for holding all of the stream's data because a subclass might not return the total size of the stream.

`void close()`

Closes this input stream and releases any platform resources associated with the stream. This method throws `IOException` when an I/O error occurs.

`void mark(int readlimit)`

Marks the current position in this input stream. A subsequent call to `reset()` repositions this stream to the last marked position so that subsequent read operations re-read the same bytes. The `readlimit` argument tells this input stream to allow that many bytes to be read before invalidating this mark (so that the stream cannot be reset to the marked position).

`boolean  
markSupported()`

Returns `true` when this input stream supports `mark()` and `reset()`; otherwise, returns `false`.

`int read()`

Reads and returns (as an `int` in the range 0 to 255) the next byte from this input stream, or returns -1 when the end of the stream is reached. This method blocks until input is available, the end of the stream is detected, or an exception is thrown. It throws `IOException` when an I/O error occurs.

<code>int read(byte[] b)</code>	Reads some number of bytes from this input stream and stores them in byte array <code>b</code> . Returns the number of bytes actually read (which might be less than <code>b</code> 's length but is never more than this length), or returns <code>-1</code> when the end of the stream is reached (no byte is available to read). This method blocks until input is available, the end of the stream is detected, or an exception is thrown. It throws <code>NullPointerException</code> when <code>b</code> is null and <code>IOException</code> when an I/O error occurs.
<code>int read(byte[] b, int off, int len)</code>	Reads no more than <code>len</code> bytes from this input stream and stores them in byte array <code>b</code> , starting at the offset specified by <code>off</code> . Returns the number of bytes actually read (which might be less than <code>len</code> but is never more than <code>len</code> ), or returns <code>-1</code> when the end of the stream is reached (no byte is available to read). This method blocks until input is available, the end of the stream is detected, or an exception is thrown. It throws <code>NullPointerException</code> when <code>b</code> is null; <code>IndexOutOfBoundsException</code> when <code>off</code> is negative, <code>len</code> is negative, or <code>len</code> is greater than <code>b.length - off</code> ; and <code>IOException</code> when an I/O error occurs.
<code>void reset()</code>	Repositions this input stream to the position at the time <code>mark()</code> was last called. This method throws <code>IOException</code> when this input stream has not been marked or the mark has been invalidated.
<code>long skip(long n)</code>	Skips over and discards <code>n</code> bytes of data from this input stream. This method might skip over some smaller number of bytes (possibly zero), for example, when the end of the file is reached before <code>n</code> bytes have been skipped. The actual number of bytes skipped is returned. When <code>n</code> is negative, no bytes are skipped. This method throws <code>IOException</code> when this input stream doesn't support skipping or when some other I/O error occurs.

---

# Методи InputStream (суперкласу для всіх підкласів для роботи з вхідним потоком даних)

---

- Підкласи InputStream, зокрема ByteArrayInputStream, підтримують позначення поточної позиції зчитування у вхідному потоці даних за допомогою методу mark() та пізніше повернення до цього місця за допомогою методу reset().
- **Обережно!** Не забувайте викликати markSupported(), щоб визначити, чи підтримує підклас методи mark() та reset().

# Класи `ByteArrayOutputStream` та `ByteArrayInputStream`

---

- Байтові масиви часто корисні в якості stream destinations та sources.
  - Клас `ByteArrayOutputStream` дозволяє записувати потік байтів у байтовий масив;
  - Клас `ByteArrayInputStream` дозволяє зчитувати потік байтів з байтового масиву.
- `ByteArrayOutputStream` оголошує 2 конструктори.
  - **`ByteArrayOutputStream()`** створює потік виводу на базі внутрішнього байтового масиву з початковим розміром 32 байти. За потреби масив розростається.
    - `ByteArrayOutputStream baos = new ByteArrayOutputStream();` створює byte array output stream із 32-байтним внутрішнім байтовим масивом
  - **`ByteArrayOutputStream(int size)`** створює потік виводу на базі внутрішнього байтового масиву із заданим розміром та розростанням за потреби. Викидає `IllegalArgumentException`, коли розмір менший за нуль.
  - Копію цього масиву можна повернути у результаті виклику методу **`byte[] toByteArray()`** класу `ByteArrayOutputStream`.

# Конструктори ByteArrayInputStream

---

- **ByteArrayInputStream(byte[] ba)** створює вхідний потік на базі байтового масиву, що напряду (без копіювання) використовує масив ba.
  - Значення position = 0, а кількість байтів = ba.length.
- **ByteArrayInputStream(byte[] ba, int offset, int count)** створює вхідний потік на базі байтового масиву, який напряду (без копіювання) використовує масив ba.
  - Значення position = offset, а кількість байтів для зчитування = count.
  - Також відстежується наступний байти для зчитування з масиву та кількість байтів на читання.
- Створимо вхідний потік на базі байтового масиву, чиїм джерелом є копія попереднього вихідного потоку на базі байтового масиву:
  - `ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());`
- Класи ByteArrayOutputStream та ByteArrayInputStream корисні для випадку, коли потрібно конвертувати зображення в масив байтів, обробити їх за деяким алгоритмом та сконвертувати їх назад в зображення.

# Приклад

---

- Наприклад, в додатку для Android з обробки зображень
  - файл із зображенням декодується у специфічний для цієї ОС екземпляр класу `android.graphics.Bitmap`.
  - Отриманий екземпляр стискається в екземпляр класу `ByteArrayOutputStream`
  - Одержується копія байтового масиву з потоку виводу,
  - Цей масив обробляється за допомогою алгоритмів,
  - Результиуючий масив конвертується в екземпляр `ByteArrayInputStream`,
  - Використовується `byte array input stream` для декодування цих байтів в інший екземпляр `Bitmap`

```
String pathname = ...; // Assume a legitimate pathname to an image.
Bitmap bm = BitmapFactory.decodeFile(pathname);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
if (bm.compress(Bitmap.CompressFormat.PNG, 100, baos))
{
    byte[] imageBytes = baos.toByteArray();
    // Do something with imageBytes.
    bm = BitmapFactory.decodeStream(new ByteArrayInputStream(imageBytes));
}
```

# Класи `FileOutputStream` та `FileInputStream`

---

- Конкретний клас `FileOutputStream` дозволяє записувати потік байтів у файл; `FileInputStream` – зчитувати його з файлу.
- `FileOutputStream` субкласує `OutputStream` та оголошує 5 конструкторів для створення потоків файлового виводу.
  - Наприклад, `FileOutputStream(String name)` створює потік файлового виводу для існуючого файлу, що визначається параметром `name`. Він перезаписує існуючий файл (якщо потрібно дописувати, використовують конструктор з параметром `append`).
  - Конструктор викидає `FileNotFoundException`, коли файл не існує та не може бути створеним; це папка, а не нормальний файл; існують інші причини, чому файл не може бути відкритим для виводу.
  - `FileOutputStream fos = new FileOutputStream("employee.dat");`
- `FileInputStream` субкласує `InputStream` та оголошує 3 конструктори для створення потоків файлового вводу.
  - Наприклад, `FileInputStream(String name)` створює потік файлового вводу з існуючого файлу за його назвою.
  - Конструктор викидає `FileNotFoundException` з тих же причин, що й `FileOutputStream()`.
  - `FileInputStream fis = new FileInputStream("employee.dat");`
- `FileOutputStream` та `FileInputStream` корисні в контексті копіювання файлів.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class Copy
{
    public static void main(String[] args)
    {
        if (args.length != 2)
        {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try
        {
            fis = new FileInputStream(args[0]);
            fos = new FileOutputStream(args[1]);
            int b; // I chose b instead of byte because byte is a reserved word.
            while ((b = fis.read()) != -1)
                fos.write(b);
        }
        catch (FileNotFoundException fnfe)
        {
            System.err.println(args[0] + " could not be opened for input, or " +
                               args[1] + " could not be created for output");
        }
        catch (IOException ioe)
        {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```

# Копіювання Source File в Destination File

---

- Метод main() спочатку перевіряє, щоб 2 аргументи командного рядка (назви source та destination файлів) були задані.
  - Потім інстанціюються FileInputStream та FileOutputStream і входимо в цикл while, який повторно зчитує байти з потоку файлового вводу та записує їх у потік файлового виводу.

Щось може піти не так.

- source file може не існувати
- destination file буде неможливо створити (наприклад, вже може існувати read-only файл з такою ж назвою).
- Викинеться виняток FileNotFoundException, який потрібно обробити.

Інші можливість – помилка вводу-виводу в процесі копіювання.

- У результаті – IOException.

```
finally
{
    if (fis != null)
        try
        {
            fis.close();
        }
        catch (IOException ioe)
        {
            assert false; // shouldn't happen in this context
        }

    if (fos != null)
        try
        {
            fos.close();
        }
        catch (IOException ioe)
        {
            assert false; // shouldn't happen in this context
        }
}
}
```

- Незалежно від появи виключення, потоки вводу і виводу потрібно закрити в блоці finally
- У простих додатках можна ігнорувати виклики методу close() і дозволити переривання виконання додатку.
  - Хоч Java автоматично закриває відкриті файли, хорошою практикою є явне закриття файлів у процесі виходу.
  - Оскільки close() здатний викидати екземпляр checked-виключення класу IOException, виклик цього методу обгортається в блок try-catch.
- Зверніть увагу на оператори if, що передують кожному блоку try.
  - Цей оператор обов'язковий для того, щоб уникати викидання екземпляру NullPointerException.

# Класи `PipedOutputStream` та `PipedInputStream`

---

- Часто потокам потрібна комунікація.
  - Один з підходів – спільні (shared) змінні.
  - Інший – використання конвеєрних потоків (pipedReaders).
  - Клас `PipedOutputStream` дозволяє відправляючому потоку (thread) записувати потік (stream) байтів в екземпляр класу `PipedInputStream`, який отримуючий потік використовує для послідовного зчитування цих байтів.
  - **Обережно!** Спроба використовувати об'єкти `PipedOutputStream` та `PipedInputStream` в рамках одного потоку не рекомендується, оскільки це може викликати взаємоблокування.
- Конструктори `PipedOutputStream`:
  - **`PipedOutputStream()`** створює конвеєрний потік виводу, що ще не під'єднаний до конвеєрного потоку вводу. Під'єднання повинно відбутись або отримувачем, або відправником перед використанням.
  - **`PipedOutputStream(PipedInputStream dest)`** створює конвеєрний потік виводу, що під'єднаний до конвеєрного потоку вводу `dest`. Записані в конвеєрний потік виводу байти можна зчитати з `dest`. Викидає `IOException`, коли виникає I/O error.
- `PipedOutputStream` оголошує метод `void connect(PipedInputStream dest)`, який під'єднує цей потік виводу до `dest`.
  - Метод викидає `IOException`, якщо цей потік уже під'єднаний до іншого конвеєрного потоку вводу.

# Конструктори PipedInputStream

---

- **PipedInputStream()** створює конвеєрний потік вводу, який ще не під'єднаний до конвеєрного потоку виводу. Повинен під'єднуватись до потоку виводу перед використанням.
- **PipedInputStream(int pipeSize)** створює конвеєрний потік вводу, який ще не під'єднаний до конвеєрного потоку виводу та використовує pipeSize, щоб визначати розмір буферу потоку вводу.
  - Повинен під'єднуватись до потоку виводу перед використанням.
  - Викидає IllegalArgumentException, коли pipeSize <= 0.
- **PipedInputStream(PipedOutputStream src)** створює конвеєрний потік вводу, який під'єднаний до конвеєрного потоку виводу src.
  - Записані в src байти можна зчитати з цього потоку виводу.
  - Викидає IOException при появі I/O error.
- **PipedInputStream(PipedOutputStream src, int pipeSize)** створює конвеєрний потік вводу, який під'єднаний до конвеєрного потоку виводу src та використовує pipeSize, щоб визначати розмір буферу потоку вводу.
  - Bytes written to src can be read from this piped input stream.
  - Викидає IOException (I/O error) та IllegalArgumentException (pipeSize <= 0).

- 
- `PipedInputStream` оголошує метод `void connect(PipedOutputStream src)`, що під'єднує цей конвеєрний потік вводу до `src`.
    - Викидає `IOException`, якщо даний потік вводу вже під'єднаний до іншого потоку виводу.

- Найпростіший спосіб створити пару конвеєрних потоків – в одному потоці. Наприклад

- ```
PipedOutputStream pos = new PipedOutputStream();  
PipedInputStream pis = new PipedInputStream(pos);
```

або

- ```
PipedInputStream pis = new PipedInputStream();  
PipedOutputStream pos = new PipedOutputStream(pis);
```

Можна залишити обидва потоки роз'єднаними (`unconnected`) і зробити це пізніше за допомогою доречного методу конвеєрного потоку `connect()`:

- ```
PipedOutputStream pos = new PipedOutputStream();  
PipedInputStream pis = new PipedInputStream();  
// ...  
pos.connect(pis);
```

```
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
```

```
public class PipedStreamsDemo
```

```
{
    public static void main(String[] args) throws IOException
    {
        final PipedOutputStream pos = new PipedOutputStream();
        final PipedInputStream pis = new PipedInputStream(pos);
        Runnable senderTask = new Runnable()
        {
            final static int LIMIT = 10;

            @Override
            public void run()
            {
                try
                {
                    for (int i = 0 ; i < LIMIT; i++)
                        pos.write((byte) (Math.random() * 256));
                }
                catch (IOException ioe)
                {
                    ioe.printStackTrace();
                }
                finally
                {
                    try
                    {
                        pos.close();
                    }
                    catch (IOException ioe)
                    {
                        ioe.printStackTrace();
                    }
                }
            }
        };
    }
}
```

# Конвеєризація випадково згенерованих байтів з Sender Thread у Receiver Thread

- Метод main() створює потоки конвеєрного виводу та вводу, які будуть використовуватись
  - потоком senderTask для передачі послідовності випадково згенерованих byte integers
  - потоком receiverTask для отримання цієї послідовності.
- Метод run() відправника явно закриває свій конвеєрний потік після завершення відправки даних.
  - Якщо цього не зробити, буде викинуто екземпляр IOException з повідомленням “write end dead”, коли приймаючий потік виконання викликатиме read() востаннє.

```

Runnable receiverTask = new Runnable()
{
    @Override
    public void run()
    {
        try
        {
            int b;
            while ((b = pis.read()) != -1)
                System.out.println(b);
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
        finally
        {
            try
            {
                pis.close();
            }
            catch (IOException ioe)
            {
                ioe.printStackTrace();
            }
        }
    }
};

Thread sender = new Thread(senderTask);
Thread receiver = new Thread(receiverTask);
sender.start();
receiver.start();
}
}

```

## Результати виводу

---

```

93
23
125
50
126
131
210
29
150
91

```

# Класи `FilterOutputStream` та `FilterInputStream`

---

- Байтові (Byte array), файлові та конвеєрні потоки передають байти unchanged to their destinations.
  - Java також підтримує фільтровані потоки (*filter streams*), які буферизують, стискають/розпаковують, шифрують/дешифрують та виконують інші маніпуляції з байтовою послідовністю потоку (that is input to the filter) до досягнення destination потоку.
- Фільтрований потік виводу (*filter output stream*) бере передані в його методи `write()` (input stream) дані, фільтрує їх та записує відфільтрований результат у відповідний потік виводу,
  - Поток виводу може виступати як інший фільтрований потік виводу, так і destination output stream, зокрема файловий потік виводу.
  - Фільтровані потоки виводу створюються з підкласів конкретного класу `FilterOutputStream` - підкласу `OutputStream`.
  - `FilterOutputStream` оголошує єдиний конструктор `FilterOutputStream(OutputStream out)`, який створює фільтрований потік виводу поверх out – underlying output stream.

# Скремблінг (Scrambling) потоку байтів

---

```
import java.io.FilterOutputStream;
import java.io.IOException;
import java.io.OutputStream;

public class ScrambledOutputStream extends FilterOutputStream
{
    private int[] map;

    public ScrambledOutputStream(OutputStream out, int[] map)
    {
        super(out);
        if (map == null)
            throw new NullPointerException("map is null");
        if (map.length != 256)
            throw new IllegalArgumentException("map.length != 256");
        this.map = map;
    }

    @Override
    public void write(int b) throws IOException
    {
        out.write(map[b]);
    }
}
```

- **Скремблювання** — шифрування потоку даних, в результаті якої він виглядає як потік випадкових бітів<sup>[1]</sup>.
- Субкласувати `FilterOutputStream` просто.
  - Мінімум, оголошуєте конструктор, який передає свій аргумент типу `OutputStream` в конструктор класу `FilterOutputStream` та переозначає метод `write(int)` з `FilterOutputStream`.
- Клас `ScrambledOutputStream` виконує просте шифрування свого потоку вводу за допомогою скремблінгу байтів вхідного потоку даних через remapping-операцію.
- Даний конструктор приймає пару аргументів:
  - **out** — визначає потік виводу, в який записуються скрембльовані байти.
  - **map** — визначає масив з 256 byte-integer значень, у які відображається вхідний потік даних.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import java.util.Random;

public class Scramble
{
    public static void main(String[] args)
    {
        if (args.length != 2)
        {
            System.err.println("usage: java Scramble srcpath destpath");
            return;
        }
        FileInputStream fis = null;
        ScrambledOutputStream sos = null;
        try
        {
            fis = new FileInputStream(args[0]);
            FileOutputStream fos = new FileOutputStream(args[1]);
            sos = new ScrambledOutputStream(fos, makeMap());
            int b;
            while ((b = fis.read()) != -1)
                sos.write(b);
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}
```

## Скремблювання байтів файлу

---

- Метод main() спочатку визначає кількість аргументів командного рядка:
  - 1) визначає source path файлу з нескрембльованим контентом;
  - 2) визначає destination path файлу, що зберігає скрембльований контент.
- Нехай введено 2 аргументи командного рядка.
  - У методі main() інстанціюється FileInputStream, таким чином створюється потік файлового вводу, приєднаний до визначеного в args[0] файлу.
  - Далі main() інстанціює FileOutputStream, створюючи потік файлового вводу, приєднаний до файлу, визначеного в args[1].
  - Потім створюється екземпляр ScrambledOutputStream та в його конструктор передається об'єкт класу FileOutputStream.

```

finally
{
    if (fis != null)
        try
        {
            fis.close();
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    if (sos != null)
        try
        {
            sos.close();
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
}
}

```

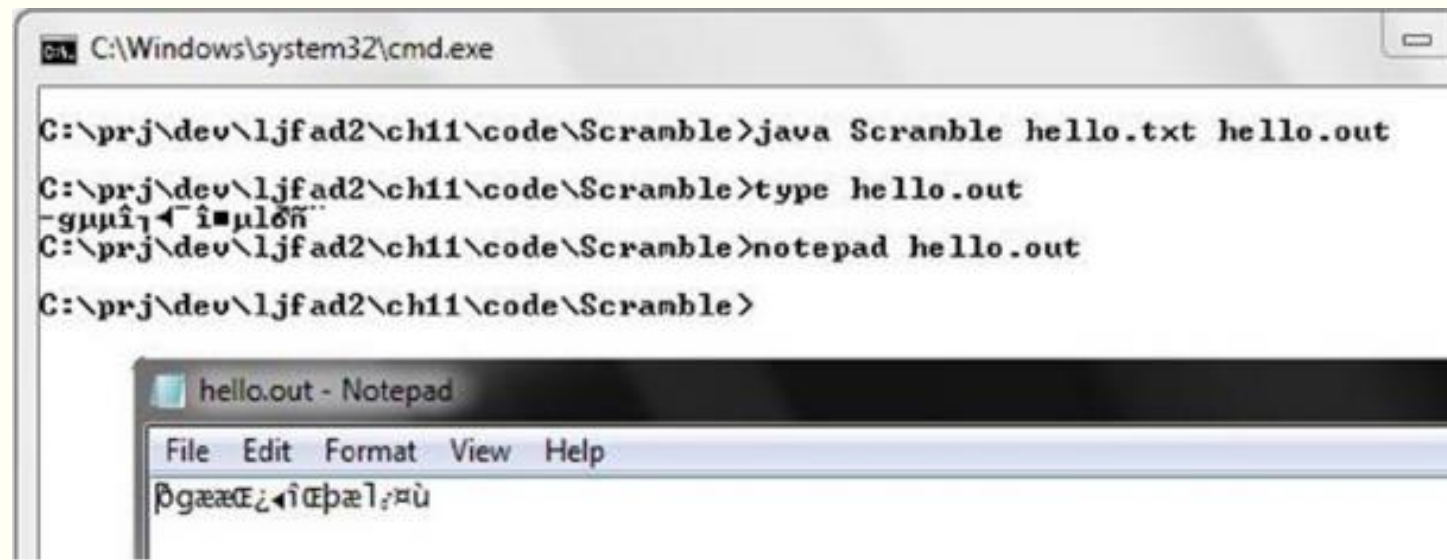
```

static int[] makeMap()
{
    int[] map = new int[256];
    for (int i = 0; i < map.length; i++)
        map[i] = i;
    // Shuffle map.
    Random r = new Random(0);
    for (int i = 0; i < map.length; i++)
    {
        int n = r.nextInt(map.length);
        int temp = map[i];
        map[i] = map[n];
        map[n] = temp;
    }
    return map;
}
}

```

- 
- **Зауважте!** Коли екземпляр потоку передається в конструктор іншого потоку, вони з'єднуються (*chain together*).
  - Тепер main() входить у цикл, зчитуючи байти reading bytes from the file input stream and writing them to the scrambled output stream by calling ScrambledOutputStream's write(int) method.
    - This loop continues until FileInputStream's read() method returns -1 (end of file).
  - The finally block closes the file input stream and scrambled output stream by calling their close() methods.
    - It doesn't call the file output stream's close() method because FilterOutputStream automatically calls the underlying output stream's close() method.
  - Метод makeMap() відповідає за створення the map array that's passed to ScrambledOutputStream's constructor.
    - The idea is to populate the array with all 256 byte-integer values, storing them in random order.

- 
- **Зауважте!** Я передаю 0 в якості зерна генератора ПВЧ при створенні об'єкту `java.util.Random`, щоб повернути прогнозовану послідовність випадкових чисел.
    - Така послідовність потрібна при створенні доповнюючого map array в Unscramble application.
    - Unscrambling не працюватиме без тієї ж послідовності чисел.
  - Візьмемо простий 15-байтний файл `hello.txt`, що містить “Hello, World!”, за яким іде перехід на новий рядок followed by a carriage return and a line feed).
    - `java Scramble hello.txt hello.out`



```
C:\Windows\system32\cmd.exe

C:\prj\dev\ljfad2\ch11\code\Scramble>java Scramble hello.txt hello.out
C:\prj\dev\ljfad2\ch11\code\Scramble>type hello.out
-gµµi1 4 i µlõñ
C:\prj\dev\ljfad2\ch11\code\Scramble>notepad hello.out
C:\prj\dev\ljfad2\ch11\code\Scramble>
```

hello.out - Notepad

File Edit Format View Help

ßgææŁ¼Œpæ1;µù

- 
- Фільтрований потік вводу (*filter input stream*) бере дані з underlying input stream (інший фільтрований потік вводу або source input stream, зокрема файловий потік вводу), фільтрує їх та робить доступними за допомогою методів read() (потіку виводу).
    - Фільтровані потоки вводу створюються з підкласів конкретного класу FilterInputStream – підкласу InputStream.
  - FilterInputStream оголошує один конструктор FilterInputStream(InputStream in), який створює фільтрований потік вводу, побудований на базі in - underlying input stream.
    - Як мінімум, оголошується конструктор, в який передається аргумент типу InputStream, а також переозначаються методи read() та read(byte[], int, int) класу FilterInputStream.

```

import java.io.FilterInputStream;
import java.io.InputStream;
import java.io.IOException;

public class ScrambledInputStream extends FilterInputStream
{
    private int[] map;
    public ScrambledInputStream(InputStream in, int[] map)
    {
        super(in);
        if (map == null)
            throw new NullPointerException("map is null");
        if (map.length != 256)
            throw new IllegalArgumentException("map.length != 256");
        this.map = map;
    }

    @Override
    public int read() throws IOException
    {
        int value = in.read();
        return (value == -1) ? -1 : map[value];
    }

    @Override
    public int read(byte[] b, int off, int len) throws IOException
    {
        int nBytes = in.read(b, off, len);
        if (nBytes <= 0)
            return nBytes;
        for (int i = 0; i < nBytes; i++)
            b[off + i] = (byte) map[off + i];
        return nBytes;
    }
}

```

## Unscrambling a Stream of Bytes

- Клас ScrambledInputStream виконує performs trivial decryption on its underlying input stream by unscrambling the underlying input stream's scrambled bytes via a remapping operation.
- Метод read() спочатку зчитує скрамбльований байт from its underlying input stream.
  - Якщо повертається -1 (кінець файлу), this value is returned to its caller.
  - Інакше, байт відображається to its unscrambled value, which is returned.
- Метод read(byte[], int, int) подібний до read(), проте зберігає байти, зчитані з underlying input stream in a byte array, taking an offset into this array and a length (number of bytes to read) into account.

- 
- 
- Отже, -1 може повертатись з виклику underlying методу read().
    - Якщо так, значення повинно повертатись.
    - Інакше, кожен байт масиву відображається to its unscrambled value, and the number of bytes read is returned.
  - **Note** Дуже важливо переозначити методи read() і read(byte[], int, int), тому що FilterInputStream's read(byte[]) method is implemented via the latter method.
    - Наступний лістинг показує вихідний код додатку Unscramble для experimenting with ScrambledInputStream by unscrambling a source file's bytes and writing these unscrambled bytes to a destination file.

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import java.util.Random;

public class Unscramble
{
    public static void main(String[] args)
    {
        if (args.length != 2)
        {
            System.err.println("usage: java Unscramble srcpath destpath");
            return;
        }
        ScrambledInputStream sis = null;
        FileOutputStream fos = null;
        try
        {
            FileInputStream fis = new FileInputStream(args[0]);
            sis = new ScrambledInputStream(fis, makeMap());
            fos = new FileOutputStream(args[1]);
            int b;
            while ((b = sis.read()) != -1)
                fos.write(b);
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}

```

## Unscrambling a File's Bytes

```

finally
{
    if (sis != null)
        try
        {
            sis.close();
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    if (fos != null)
        try
        {
            fos.close();
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
}
}

```

```
static int[] makeMap()
{
    int[] map = new int[256];
    for (int i = 0; i < map.length; i++)
        map[i] = i;
    // Shuffle map.
    Random r = new Random(0);
    for (int i = 0; i < map.length; i++)
    {
        int n = r.nextInt(map.length);
        int temp = map[i];
        map[i] = map[n];
        map[n] = temp;
    }
    int[] temp = new int[256];
    for (int i = 0; i < temp.length; i++)
        temp[map[i]] = i;
    return temp;
}
```

- Метод main() спочатку перевіряє кількість аргументів командного рядка:
  - 1) source path of the file with scrambled content;
  - destination path of the file that stores unscrambled content.
- Вважаючи, що введено 2 аргументи командного рядка, main() інстанціює `FileInputStream`, створюючи файловий потік вводу, під'єднаний до файлу, визначеного в `args[1]`.
  - Далі main() інстанціює `FileInputStream`, створюючи файловий потік вводу, під'єднаний до файлу з `args[0]`.
  - Потім істанціюється `ScrambledInputStream`, and passes the `FileInputStream` instance to `ScrambledInputStream's` constructor.

- 
- 
- **Note** When a stream instance is passed to another stream class's constructor, the two streams are *chained together*.
    - For example, the scrambled input stream is chained to the file input stream.
    - `main()` now enters a loop, reading bytes from the scrambled input stream and writing them to the file output stream.
    - This loop continues until `ScrambledInputStream`'s `read()` method returns -1 (end of file).
    - The finally block closes the scrambled input stream and file output stream by calling their `close()` methods.
    - It doesn't call the file input stream's `close()` method because `FilterOutputStream` automatically calls the underlying input stream's `close()` method.
  - Метод `makeMap()` method is responsible for creating the map array that's passed to `ScrambledInputStream`'s constructor.
    - The idea is to duplicate Listing 11-14's map array and then invert it so that unscrambling can be performed.
    - Continuing from the previous `hello.txt/hello.out` example, execute `java Unscramble hello.out hello.bak` and you'll see the same unscrambled content in `hello.bak` that's present in `hello.txt`.

# Класи `BufferedOutputStream` та `BufferedInputStream`

---

- `FileOutputStream` and `FileInputStream` мають проблему продуктивності.
  - Кожен виклик методу `write()` для файлового потоку виводу та методу `read()` для файлового потоку вводу призводить до виклику одного з нативних методів `underlying` платформи,
  - Ці нативні виклики сповільнюють I/O.
- **Note** Нативний (*native*) метод – функція API базової платформи, яку Java підключає до додатку за допомогою *Java Native Interface (JNI)*.
  - Java постачає зарезервоване слово `native`, щоб ідентифікувати нативний метод.
  - Наприклад, клас `RandomAccessFile` оголошує метод  
`private native void open(String name, int mode) method.`
  - Коли конструктор `RandomAccessFile` викликає цей метод, Java звертається до базової платформи (через JNI) щодо відкриття заданого файлу в заданому режимі on Java's behalf.

- 
- Конкретні класи `BufferedOutputStream` та `BufferedInputStream` фільтрованого потоку покращують продуктивність, мінімізуючи виклики методів `underlying output stream write()` and `underlying input stream read()`.
  - Замість них Java бере до уваги буфери.
    - Коли буфер для запису повний, `write()` викликає `underlying` метод `write()` потоку виводу, щоб очистити буфер. Наступні виклики методів `write()` з `BufferedOutputStream` зберігають байти в буфер до наступного переповнення.
    - Коли буфер для зчитування порожній, виклики `read()` ведуть до `underlying` методу `read()` потоку вводу для заповнення буферу. Наступні виклики цих методів повертають байти з буферу, поки він знову не стане порожнім.
  - Конструктори `BufferedOutputStream`:
    - **`BufferedOutputStream(OutputStream out)`** creates a buffered output stream that streams its output to out. An internal buffer is created to store bytes written to out.
    - **`BufferedOutputStream(OutputStream out, int size)`** creates a buffered output stream that streams its output to out. An internal buffer of length size is created to store bytes written to out.

- 
- У наступному прикладі поєднаємо екземпляр `BufferedOutputStream` з екземпляром `FileOutputStream`.
    - Відповідні виклики методу `write()` для екземпляру `BufferedOutputStream` буферизують байти та час від часу result in internal `write()` method calls on the encapsulated `FileOutputStream` instance.

```
FileOutputStream fos = new FileOutputStream("employee.dat");
BufferedOutputStream bos = new BufferedOutputStream(fos); // Chain bos to fos.
bos.write(0); // Write to employee.dat through the buffer.
// Additional write() method calls.
bos.close(); // This method call internally calls fos's close() method.
```

- Конструктори `BufferedInputStream`:
  - **`BufferedInputStream(InputStream in)`** creates a buffered input stream that streams its input from `in`. An internal buffer is created to store bytes read from `in`.
  - **`BufferedInputStream(InputStream in, int size)`** creates a buffered input stream that streams its input from `in`. An internal buffer of length `size` is created to store bytes read from `in`.

- 
- Наступний приклад поєднує екземпляр `BufferedInputStream` до екземпляру `FileInputStream`.
    - Subsequent `read()` method calls on the `BufferedInputStream` instance unbuffer bytes and occasionally result in internal `read()` method calls on the encapsulated `FileInputStream` instance.

```
FileInputStream fis = new FileInputStream("employee.dat");  
BufferedInputStream bis = new BufferedInputStream(fis); // Chain bis to fis.  
int ch = bis.read(); // Read employee.dat through the buffer.  
// Additional read() method calls.  
bis.close(); // This method call internally calls fis's close() method.
```

# Класи DataOutputStream та DataInputStream

---

- `FileOutputStream` і `FileInputStream` корисні для запису та зчитування байтів та масивів байтів.
  - Проте вони не постачають підтримки для запису/зчитування примітивних типів даних та рядків.
  - З цією метою Java постачає конкретні класи `DataOutputStream` та `DataInputStream` фільтрованого потоку.
- Кожен з них долає це обмеження, постачаючи методи для запису чи зчитування примітивних значень та рядків у платформозалежному стилі.
  - Цілі числа записуються/зчитуються в *big-endian format* (the most significant byte comes first).
    - <http://en.wikipedia.org/wiki/Endianness>
  - Floating-point і double precision floating-point values are written and read according to the IEEE 754 standard, which specifies 4 bytes per floating-point value and 8 bytes per double precision floating-point value.
  - Strings are written and read according to a modified version of *UTF-8*, a variable-length encoding standard for efficiently storing 2-byte Unicode characters.
    - <http://en.wikipedia.org/wiki/Utf-8>

- 
- `DataOutputStream` declares a single `DataOutputStream(OutputStream out)` constructor.
    - Because this class implements the `DataOutput` interface, `DataOutputStream` also provides access to the `write` methods as provided by `RandomAccessFile`.
  - `DataInputStream` declares a single `DataInputStream(InputStream in)` constructor.
    - Because this class implements the `DataInput` interface, `DataInputStream` also provides access to the `read` methods as provided by `RandomAccessFile`.
  - Наступний лістинг показує код додатку `DataStreamDemo`, який використовує екземпляр `DataOutputStream` для запису мультібайтових значень в екземпляр `FileOutputStream` та використовує `instance` and uses a `DataInputStream` instance to read multibyte values from a `FileInputStream` instance.

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class DataStreamsDemo
{
    final static String FILENAME = "values.dat";

    public static void main(String[] args)
    {
        DataOutputStream dos = null;
        DataInputStream dis = null;
        try
        {
            FileOutputStream fos = new FileOutputStream(FILENAME);
            dos = new DataOutputStream(fos);
            dos.writeInt(1995);
            dos.writeUTF("Saving this String in modified UTF-8 format!");
            dos.writeFloat(1.0F);
            dos.close(); // Close underlying file output stream.
            // The following null assignment prevents another close attempt on
            // dos (which is now closed) should IOException be thrown from
            // subsequent method calls.
            dos = null;
            FileInputStream fis = new FileInputStream(FILENAME);
            dis = new DataInputStream(fis);
            System.out.println(dis.readInt());
            System.out.println(dis.readUTF());
            System.out.println(dis.readFloat());
        }
        catch (IOException ioe)
        {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```

## *Outputting and Then Inputting a Stream of Multibyte Values*

```

        finally
        {
            if (dos != null)
            {
                try
                {
                    dos.close();
                }
                catch (IOException ioe2) // Cannot redeclare local variable ioe.
                {
                    assert false; // shouldn't happen in this context
                }
            }
            if (dis != null)
            {
                try
                {
                    dis.close();
                }
                catch (IOException ioe2) // Cannot redeclare local variable ioe.
                {
                    assert false; // shouldn't happen in this context
                }
            }
        }
    }
}

```

- DataStreamsDemo створює файл values.dat; calls DataOutputStream methods to write an integer, a string, and a floating-point value to this file; and calls DataInputStream methods to read back these values.
- Unsurprisingly, it generates the following output:

```

1995
Saving this String in modified UTF-8 format!
1.0

```

- 
- **Caution** При зчитуванні значень з файлу, записаних з послідовності `DataOutputStream` method calls, make sure to use the same method-call sequence.
    - Otherwise, you're bound to end up with erroneous data and, in the case of the `readUTF()` methods, thrown instances of the `java.io.UTFDataFormatException` class (a subclass of `IOException`).

# Серіалізація та десеріалізація об'єкту

---

- Java постачає класи `DataOutputStream` та `DataInputStream` з метою потокової передачі значень примітивного типу та об'єктів `String`.
  - Проте використовувати ці класи для передачі не-`String` об'єктів неможливо.
  - Замість цього використовують серіалізацію та десеріалізацію об'єкту, щоб передавати об'єкти довільного типу.
- *Серіалізація об'єкту (Object serialization)* – механізм ВМ для перетворення стану об'єкта в потік байтів.
  - Обернена операція називається десеріалізацією.
- Стан об'єкту складається з полів екземпляру, які зберігають значення примітивних типів та/або посилання на інші об'єкти.
  - Коли об'єкт серіалізовано, об'єкти, що є частинами його стану, також серіалізуються (якщо цьому не запобігає розробник) і т.д.
- Java підтримує
  - серіалізацію та десеріалізацію за замовчуванням,
  - custom serialization and deserialization,
  - externalization.

# Серіалізація та десеріалізація за замовчуванням

---

- Хоча Java виконує більшість роботи, декілька задач має прописати розробник:
  - Потрібно, щоб клас, об'єкт якого серіалізується, реалізовував інтерфейс `java.io.Serializable` interface (напрямую або від суперкласу).
  - The rationale for implementing `Serializable` is to avoid unlimited serialization.
- `Serializable` – порожній маркерний інтерфейс (методів для реалізації не передбачено), який клас реалізує, щоб повідомити VM, що it's okay to serialize the class's objects.
  - Коли механізм серіалізації encounters об'єкт, чий клас не реалізує `Serializable`, викидається екземпляр класу `java.io.NotSerializableException` (непрямий підклас `IOException`).

## Необмежена серіалізація (Unlimited serialization)

---

- Це процес серіалізації всього графу об'єкта (object graph).
- Java не підтримує необмежену серіалізацію, тому що:
  - **Безпека**: якби Java автоматично серіалізувала об'єкт, що містив sensitive information (пароль, номер кредитки тощо), хакеру за простою міг отримати цю інформацію and wreak havoc. Краще дати розробнику вибір, щоб уникати таких ситуацій.
  - **Продуктивність**: серіалізація використовує Reflection API, що сповільнює роботу додатку. Необмежена серіалізація може суттєво вплинути на продуктивність додатку.
  - **Об'єкти, які не піддаються серіалізації**: деякі об'єкти існують тільки в контексті запущеного додатку, і їх серіалізація безглузда. Наприклад, об'єкт файлового потоку даних, який десеріалізується, більше не представляє підключення до файлу.

# Implementing Serializable

---

```
import java.io.Serializable;

public class Employee implements Serializable
{
    private String name;
    private int age;

    public Employee(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }

    public int getAge() { return age; }
}
```

- Лістинг оголошує клас Employee, що реалізує інтерфейс Serializable.
  - Не буде викинуто NotSerializableException.
  - Не лише Employee реалізує Serializable, клас String – теж.
- Your second task is to work with the ObjectOutputStream class and its writeObject() method to serialize an object and the ObjectInputStream class and its readObject() method to deserialize the object.
- **Note** Although ObjectOutputStream extends OutputStream instead of FilterOutputStream, and although ObjectInputStream extends InputStream instead of FilterInputStream, these classes behave as filter streams.

- 
- 
- Java provides the concrete `ObjectOutputStream` class to initiate the serialization of an object's state to an object output stream.
    - This class declares an `ObjectOutputStream(OutputStream out)` constructor that chains the object output stream to the output stream specified by `out`.
    - When you pass an output stream reference to `out`, this constructor attempts to write a serialization header to that output stream.
    - It throws `NullPointerException` when `out` is null and `IOException` when an I/O error prevents it from writing this header.
  - `ObjectOutputStream` serializes an object via its `void writeObject(Object obj)` method.
    - This method attempts to write information about `obj`'s class followed by the values of `obj`'s instance fields to the underlying output stream.

- 
- `writeObject()` doesn't serialize the contents of static fields.
  - In contrast, it serializes the contents of all instance fields that are not explicitly prefixed with the `transient` reserved word. For example, consider the following field declaration:  
`public transient char[] password;`
    - This declaration specifies `transient` to avoid serializing a password for some hacker to encounter.
    - The virtual machine's serialization mechanism ignores any instance field that's marked `transient`.
  - **Note** Check out my "Transience" blog post ([www.javaworld.com/community/node/13451](http://www.javaworld.com/community/node/13451)) to learn more about `transient`.
  - `writeObject()` throws `IOException` or an instance of an `IOException` subclass when something goes wrong.
    - For example, this method throws `NotSerializableException` when it encounters an object whose class doesn't implement `Serializable`.
  - **Note** Because `ObjectOutputStream` implements `DataOutput`, it also declares methods for writing primitive-type values and strings to an object output stream.

- 
- Java provides the concrete `ObjectInputStream` class to initiate the deserialization of an object's state from an object input stream. This class declares an `ObjectInputStream(InputStream in)` constructor that chains the object input stream to the input stream specified by `in`.
    - When you pass an input stream reference to `in`, this constructor attempts to read a serialization header from that input stream.
    - It throws `NullPointerException` when `in` is null, `IOException` when an I/O error prevents it from reading this header, and `java.io.StreamCorruptedException` (an indirect subclass of `IOException`) when the stream header is incorrect.
  - `ObjectInputStream` deserializes an object via its `Object readObject()` method.
    - This method attempts to read information about obj's class followed by the values of obj's instance fields from the underlying input stream.
    - `readObject()` throws `java.lang.ClassNotFoundException`, `IOException`, or an instance of an `IOException` subclass when something goes wrong.
    - For example, this method throws `java.io.OptionalDataException` when it encounters primitive-type values instead of objects.
  - **Note** Because `ObjectInputStream` implements `DataInput`, it also declares methods for reading primitive-type values and strings from an object input stream.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

```
public class SerializationDemo
{
    final static String FILENAME = "employee.dat";

    public static void main(String[] args)
    {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try
        {
            FileOutputStream fos = new FileOutputStream(FILENAME);
            oos = new ObjectOutputStream(fos);
            Employee emp = new Employee("John Doe", 36);
            oos.writeObject(emp);
            oos.close();
            oos = null;
            FileInputStream fis = new FileInputStream(FILENAME);
            ois = new ObjectInputStream(fis);
            emp = (Employee) ois.readObject(); // (Employee) cast is necessary.
            ois.close();
            System.out.println(emp.getName());
            System.out.println(emp.getAge());
        }
        catch (ClassNotFoundException cnfe)
        {
            System.err.println(cnfe.getMessage());
        }
    }
}
```

## *Serializing and Deserializing an Employee Object*

- Listing 11-19 presents an application that uses these classes to serialize and deserialize an instance of Listing 11-18's Employee class to and from an employee.dat file.
- Listing 11-19's main() method first instantiates Employee and serializes this instance via writeObject() to employee.dat.
- It then deserializes this instance from this file via readObject(), and invokes the instance's getName() and getAge() methods.

```

        catch (IOException ioe)
        {
            System.err.println(ioe.getMessage());
        }
        finally
        {
            if (oos != null)
            {
                try
                {
                    oos.close();
                }
                catch (IOException ioe)
                {
                    assert false; // shouldn't happen in this context
                }
            }

            if (ois != null)
            {
                try
                {
                    ois.close();
                }
                catch (IOException ioe)
                {
                    assert false; // shouldn't happen in this context
                }
            }
        }
    }
}

```

Along with employee.dat, you'll discover the following output when you run this application:

John Doe

- There's no guarantee that the same class will exist when a serialized object is deserialized (perhaps an instance field has been deleted).
- During deserialization, this mechanism causes readObject() to throw `java.io.InvalidClassException`—an indirect subclass of the `IOException` class—when it detects a difference between the deserialized object and its class.

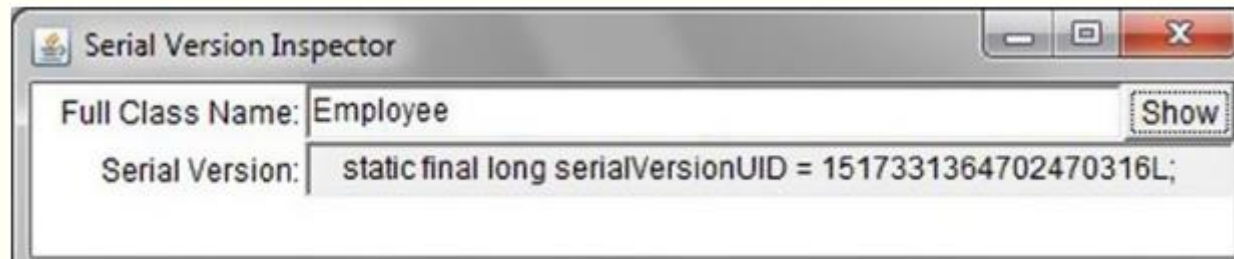
- 
- Every serialized object has an identifier.
    - The deserialization mechanism compares the identifier of the object being deserialized with the serialized identifier of its class (all serializable classes are automatically given unique identifiers unless they explicitly specify their own identifiers) and causes `InvalidClassException` to be thrown when it detects a mismatch.
    - Perhaps you've added an instance field to a class, and you want the deserialization mechanism to set the instance field to a default value rather than have `readObject()` throw an `InvalidClassException` instance.
    - (The next time you serialize the object, the new field's value will be written out.)
  - You can avoid the thrown `InvalidClassException` instance by adding a static final `long serialVersionUID = long integer value`; declaration to the class.
    - The *long integer value* must be unique and is known as a *stream unique identifier (SUID)*.

- 
- 
- During deserialization, the virtual machine will compare the deserialized object's SUID to its class's SUID. If they match, `readObject()` will not throw `InvalidClassException` when it encounters *acompatible class change* (such as adding an instance field).
    - However, it will still throw this exception when it encounters an *incompatible class change* (such as changing an instance field's name or type).
  - **Note** Whenever you change a class in some fashion, you must calculate a new SUID and assign it to `serialVersionUID`.

# The serialver user interface reveals Employee's SUID

---

- The JDK provides a serialver tool for calculating the SUID. For example, to generate an SUID for Listing 11-18's Employee class, change to the directory containing Employee.class and execute the following command:  
serialver Employee  
In response, serialver generates the following output, which you paste (except for Employee:) into Employee.java:  
Employee: static final long serialVersionUID = 1517331364702470316L;  
The Windows version of serialver also provides a graphical user interface that you might find more convenient to use. To access this interface, specify the following command line:  
serialver -show  
When the serialver window appears, enter Employee into the Full Class Name text field and click the Show button.



# Custom Serialization and Deserialization

---

- My previous discussion focused on default serialization and deserialization (with the exception of marking an instance field transient to prevent it from being included during serialization). However, situations arise where you need to customize these tasks. For example, suppose you want to serialize instances of a class that doesn't implement Serializable. As a workaround, you subclass this other class, have the subclass implement Serializable, and forward subclass constructor calls to the superclass. Although this workaround lets you serialize subclass objects, you cannot deserialize these serialized objects when the superclass doesn't declare a noargument constructor, which is required by the deserialization mechanism. Listing 11-20 demonstrates this problem.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
```

```
class Employee
{
    private String name;

    Employee(String name)
    {
        this.name = name;
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

```
class SerEmployee extends Employee implements Serializable
{
    SerEmployee(String name)
    {
        super(name);
    }
}
```

## Problematic Deserialization

```

public class SerializationDemo
{
    public static void main(String[] args)
    {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try
        {
            oos = new ObjectOutputStream(new FileOutputStream("employee.dat"));
            SerEmployee se = new SerEmployee("John Doe");
            System.out.println(se);
            oos.writeObject(se);
            oos.close();
            oos = null;
            System.out.println("se object written to file");
            ois = new ObjectInputStream(new FileInputStream("employee.dat"));
            se = (SerEmployee) ois.readObject();
            System.out.println("se object read from file");
            System.out.println(se);
        }
        catch (ClassNotFoundException cnfe)
        {
            cnfe.printStackTrace();
        }
    }
}

```

- Listing 11-20's main() method instantiates SerEmployee with an employee name. This class's SerEmployee(String) constructor passes this argument to its Employee counterpart.
- main() next calls Employee's toString() method indirectly via System.out.println() to obtain this name, which is then output.

```

        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
        finally
        {
            if (oos != null)
            {
                try
                {
                    oos.close();
                }
                catch (IOException ioe)
                {
                    assert false; // shouldn't happen in this context
                }
            }
            if (ois != null)
            {
                try
                {
                    ois.close();
                }
                catch (IOException ioe)
                {
                    assert false; // shouldn't happen in this context
                }
            }
        }
    }
}

```

- 
- Continuing, main() serializes the SerEmployee instance to an employee.dat file via writeObject().
  - It then attempts to deserialize this object via readObject(), and this is where the trouble occurs, as revealed by the following output:

```
John Doe  
se object written to file  
java.io.InvalidClassException: SerEmployee; no valid constructor  
    at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(Unknown Source)  
    at java.io.ObjectStreamClass.checkDeserialize(Unknown Source)  
    at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)  
    at java.io.ObjectInputStream.readObject0(Unknown Source)  
    at java.io.ObjectInputStream.readObject(Unknown Source)  
    at SerializationDemo.main(SerializationDemo.java:48)
```

- This output reveals a thrown instance of the InvalidClassException class.
  - This exception object was thrown during deserialization because Employee doesn't possess a noargument constructor.

- 
- 
- You can overcome this problem by taking advantage of the wrapper class pattern (Chapter 4).
    - Furthermore, you declare a pair of private methods in the subclass that the serialization and deserialization mechanisms look for and call.
  - Normally, the serialization mechanism writes out a class's instance fields to the underlying output stream.
    - However, you can prevent this from happening by declaring a private void `writeObject(ObjectOutputStream oos)` method in that class.
  - When the serialization mechanism discovers this method, it calls the method instead of automatically outputting instance field values.
    - The only values that are output are those explicitly output via the method.
  - Conversely, the deserialization mechanism assigns values to a class's instance fields that it reads from the underlying input stream.
    - However, you can prevent this from happening by declaring a private void `readObject(ObjectInputStream ois)` method.

- 
- 
- When the deserialization mechanism discovers this method, it calls the method instead of automatically assigning values to instance fields.
    - The only values that are assigned to instance fields are those explicitly assigned via the method.
    - Because SerEmployee doesn't introduce any fields, and because Employee doesn't offer access to its internal fields (assume you don't have the source code for this class), what would a serialized SerEmployee object include?
  - Although you cannot serialize Employee's internal state, you can serialize the argument(s) passed to its constructors, such as the employee name.
  - Listing 11-21 reveals the refactored SerEmployee and SerializationDemo classes.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
```

```
class Employee
{
    private String name;

    Employee(String name)
    {
        this.name = name;
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

# Solving Problematic Deserialization

```
class SerEmployee implements Serializable
{
    private Employee emp;
    private String name;

    SerEmployee(String name)
    {
        this.name = name;
        emp = new Employee(name);
    }

    private void writeObject(ObjectOutputStream oos) throws IOException
    {
        oos.writeUTF(name);
    }

    private void readObject(ObjectInputStream ois)
        throws ClassNotFoundException, IOException
    {
        name = ois.readUTF();
        emp = new Employee(name);
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

```

public class SerializationDemo
{
    public static void main(String[] args)
    {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try
        {
            oos = new ObjectOutputStream(new FileOutputStream("employee.dat"));
            SerEmployee se = new SerEmployee("John Doe");
            System.out.println(se);
            oos.writeObject(se);
            oos.close();
            oos = null;
            System.out.println("se object written to file");
            ois = new ObjectInputStream(new FileInputStream("employee.dat"));
            se = (SerEmployee) ois.readObject();
            System.out.println("se object read from file");
            System.out.println(se);
        }
        catch (ClassNotFoundException cnfe)
        {
            cnfe.printStackTrace();
        }
    }
}

```

- SerEmployee's writeObject() and readObject() methods rely on DataOutput and DataInput methods:
  - they don't need to call ObjectOutputStream's writeObject() method and ObjectInputStream's readObject() method to perform their tasks.

- Результат
 

```

John Doe
se object written to file
se object read from file
John Doe

```

```

        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
        finally
        {
            if (oos != null)
            {
                try
                {
                    oos.close();
                }
                catch (IOException ioe)
                {
                    assert false; // shouldn't happen in this context
                }
            }
            if (ois != null)
            {
                try
                {
                    ois.close();
                }
                catch (IOException ioe)
                {
                    assert false; // shouldn't happen in this context
                }
            }
        }
    }
}

```

- 
- 
- The `writeObject()` and `readObject()` methods can be used to serialize/deserialize data items beyond the normal state (non-transient instance fields), for example, serializing/deserializing the contents of a static field.
  - However, before serializing or deserializing the additional data items, you must tell the serialization and deserialization mechanisms to serialize or deserialize the object's normal state.
  - The following methods help you accomplish this task:
    - ☐ `ObjectOutputStream`'s `defaultWriteObject()` method outputs the object's normal state. Your `writeObject()` method first calls this method to output that state and then outputs additional data items via `ObjectOutputStream` methods such as `writeUTF()`.
    - `ObjectInputStream`'s `defaultReadObject()` method inputs the object's normal state. Your `readObject()` method first calls this method to input that state and then inputs additional data items via `ObjectInputStream` methods such as `readUTF()`.

# Externalization

---

- Along with default serialization/deserialization and custom serialization/deserialization, Java supports externalization.
  - Unlike default/custom serialization/deserialization, *externalization* offers complete control over the serialization and deserialization tasks.
- **Note** Externalization helps you improve the performance of the reflection-based serialization and deserialization mechanisms by giving you complete control over what fields are serialized and deserialized.
- Java supports externalization via `java.io.Externalizable`. This interface declares the following pair of public methods:
  - `void writeExternal(ObjectOutput out)` saves the calling object's contents by calling various methods on the out object. This method throws `IOException` when an I/O error occurs. (`java.io.ObjectOutput` is a subinterface of `DataOutput` and is implemented by `ObjectOutputStream`.)
  - `void readExternal(ObjectInput in)` restores the calling object's contents by calling various methods on the in object. This method throws `IOException` when an I/O error occurs and `ClassNotFoundException` when the class of the object being restored cannot be found. (`java.io.ObjectInput` is a subinterface of `DataInput` and is implemented by `ObjectInputStream`.)
- If a class implements `Externalizable`, its `writeExternal()` method is responsible for saving all field values that are to be saved.
  - Also, its `readExternal()` method is responsible for restoring all saved field values and in the order they were saved.

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
```

```
public class Employee implements Externalizable
{
    private String name;
    private int age;
    public Employee()
    {
        System.out.println("Employee() called");
    }

    public Employee(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }

    public int getAge() { return age; }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException
    {
        System.out.println("writeExternal() called");
        out.writeUTF(name);
        out.writeInt(age);
    }

    @Override
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException
    {
        System.out.println("readExternal() called");
        name = in.readUTF();
        age = in.readInt();
    }
}
```

## Refactoring Listing 11-18's Employee Class to Support Externalization

- Employee declares a public Employee() constructor because each class that participates in externalization must declare a public noargument constructor. The deserialization mechanism calls this constructor to instantiate the object.  
**Caution** The deserialization mechanism throws InvalidClassException with a “no valid constructor” message when it doesn't detect a public noargument constructor.

- 
- Initiate externalization by instantiating `ObjectOutputStream` and calling its `writeObject(Object)` method, or by instantiating `ObjectInputStream` and calling its `readObject()` method.
  - **Note** When passing an object whose class (directly/indirectly) implements `Externalizable` to `writeObject()`, the `writeObject()`-initiated serialization mechanism writes only the identity of the object's class to the object output stream.
  - Suppose you compiled Listing 11-19's `SerializationDemo.java` source code and Listing 11-22's `Employee.java` source code in the same directory.
  - Now suppose you executed `java SerializationDemo`. In response, you would observe the following output:  
writeExternal() called  
Employee() called  
readExternal() called  
John Doe  
36

- 
- Before serializing an object, the serialization mechanism checks the object's class to see if it implements `Externalizable`. If so, the mechanism calls `writeExternal()`. Otherwise, it looks for a private `writeObject(ObjectOutputStream)` method and calls this method when present. When this method isn't present, this mechanism performs default serialization, which includes only nontransient instance fields. Before deserializing an object, the deserialization mechanism checks the object's class to see if it implements `Externalizable`. If so, the mechanism attempts to instantiate the class via the public noargument constructor. Assuming success, it calls `readExternal()`. When the object's class doesn't implement `Externalizable`, the deserialization mechanism looks for a private `readObject(ObjectInputStream)` method. When this method isn't present, this mechanism performs default deserialization, which includes only non-transient instance fields.

# PrintStream

---

- Of all the stream classes, `PrintStream` is an oddball: it should have been named `PrintOutputStream` for consistency with the naming convention. This filter output stream class writes string representations of input data items to the underlying output stream.  
**Note** `PrintStream` uses the default character encoding to convert a string's characters to bytes.  
(I'll discuss character encodings when I introduce you to writers and readers in the next section.) Because `PrintStream` doesn't support different character encodings, you should use the equivalent `PrintWriter` class instead of `PrintStream`. However, you need to know about `PrintStream` because of Standard I/O (see Chapter 1 for an introduction to this topic).  
`PrintStream` instances are print streams whose various `print()` and `println()` methods print string representations of integers, floating-point values, and other data items to the underlying output stream. Unlike the `print()` methods, `println()` methods append a line terminator to their output.

- 
- **Note** The line terminator (also known as line separator) isn't necessarily the newline (also commonly referred to as line feed). Instead, to promote portability, the line separator is the sequence of characters defined by system property `line.separator`. On Windows platforms, `System.getProperty("line.separator")` returns the actual carriage return code (13), which is symbolically represented by `\r`, followed by the actual newline/line feed code (10), which is symbolically represented by `\n`. In contrast, `System.getProperty("line.separator")` returns only the actual newline/line feed code on Unix and Linux platforms. The `println()` methods call their corresponding `print()` methods followed by the equivalent of the `void println()` method, which eventually results in `line.separator`'s value being output. For example, `void println(int x)` outputs `x`'s string representation and calls this method to output the line separator.

- 
- **Caution** Never hard-code the `\n` escape sequence in a string literal that you are going to output via a `print()` or `println()` method. Doing so isn't portable. For example, when Java executes `System.out.print("first line\n");` followed by `System.out.println("second line");`, you will see first line on one line followed by second line on a subsequent line when this output is viewed at the Windows command line. In contrast, you'll see first linessecond line when this output is viewed in the Windows Notepad application (which requires a carriage return/line feed sequence to terminate lines). When you need to output a blank line, the easiest way to do this is to call `System.out.println();`, which is why you find this method call used elsewhere in my book. I confess that I don't always follow my own advice, so you might find instances of `\n` in literal strings being passed to `System.out.print()` or `System.out.println()` elsewhere in this book.

- 
- 
- `PrintStream` offers three other features that you'll find useful:
    - Unlike other output streams, a print stream never rethrows an `IOException` instance thrown from the underlying output stream. Instead, exceptional situations set an internal flag that can be tested by calling `PrintStream`'s boolean `checkError()` method, which returns `true` to indicate a problem.
    - `PrintStream` objects can be created to automatically flush their output to the underlying output stream. In other words, the `flush()` method is automatically called after a byte array is written, one of the `println()` methods is called, or a newline is written.
    - `PrintStream` declares a `PrintStream format(String format, Object ...args)` method for achieving formatted output. Behind the scene, this method works with the `Formatter` class that I introduce in Chapter 13. `PrintStream` also declares a `printf(String format, Object... args)` convenience method that delegates to the `format()` method. For example, invoking `printf()` via `out.printf(format, args)` is identical to invoking `out.format(format, args)`.

# Standard I/O Revisited

---

- In Chapter 1, I introduced you to Standard I/O. I stated that you input data items from the standard input stream by making `System.in.read()` method calls, that you output data items to the standard output stream by making `System.out.print()` and `System.out.println()` method calls, and that you output data items to the standard error stream by making `System.err.print()` and `System.err.println()` method calls. Finally, I discussed I/O redirection. `System.in`, `System.out`, and `System.err` are formally described by the following class fields in the `System` class:
  - `public static final InputStream in`
  - `public static final PrintStream out`
  - `public static final PrintStream err`These fields contain references to `InputStream` and `PrintStream` objects that represent the standard input, standard output, and standard error streams.

- 
- 
- When you invoke `System.in.read()`, the input is originating from the source identified by the `InputStream` instance assigned to `in`.
    - Similarly, when you invoke `System.out.print()` or `System.err.println()`, the output is being sent to the destination identified by the `PrintStream` instance assigned to `out` or `err`, respectively.
  - **Tip** On an Android device, you can view content sent to standard output and standard error by first executing `adb logcat` at the command line. `adb` is one of the tools included in the Android SDK.
  - Java initializes `in` to refer to the keyboard or a file when the standard input stream is redirected to the file. Similarly, Java initializes `out/err` to refer to the screen or a file when the standard output/error stream is redirected to the file.
  - You can programmatically specify the input source, output destination, and error destination by calling the following `System` class methods:
    - `void setIn(InputStream in)`
    - `void setOut(PrintStream out)`
    - `void setErr(PrintStream err)`
  - Listing 11-23 presents an application that shows you how to use these methods to programmatically redirect the standard input, standard output, and standard error destinations.

# Programmatically Specifying the Standard Input Source and Standard Output/Error Destinations

---

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;

public class RedirectIO
{
    public static void main(String[] args) throws IOException
    {
        if (args.length != 3)
        {
            System.err.println("usage: java RedirectIO stdinfile stdoutfile stderrfile");
            return;
        }

        System.setIn(new FileInputStream(args[0]));
        System.setOut(new PrintStream(args[1]));
        System.setErr(new PrintStream(args[2]));

        int ch;
        while ((ch = System.in.read()) != -1)
            System.out.print((char) ch);

        System.err.println("Redirected error output");
    }
}
```

Listing 11-23 presents a RedirectIO application that lets you specify (via command-line arguments) the name of a file from which System.in.read() obtains its content as well as the names of files to which System.out.print() and System.err.println() send their content. It then proceeds to copy standard input to standard output and then demonstrates outputting content to standard error.

- 
- 
- Next, new `FileInputStream(args[0])` provides access to the input sequence of bytes that is stored in the file identified by `args[0]`. Similarly, new `PrintStream(args[1])` provides access to the file identified by `args[1]`, which will store the output sequence of bytes, and new `PrintStream(args[2])` provides access to the file identified by `args[2]`, which will store the error sequence of bytes.
  - Compile Listing 11-23 ( `javac RedirectIO.java`). Then execute the following command line:  
`java RedirectIO RedirectIO.java out.txt err.txt`  
This command line produces no visual output on the screen. Instead, it copies the contents of `RedirectIO.java` to `out.txt`. It also stores Redirected error output in `err.txt`.



# WORKING WITH WRITERS AND READERS

- 
- 
- Java's stream classes are good for streaming sequences of bytes, but they're not good for streaming sequences of characters because bytes and characters are two different things: a byte represents an 8-bit data item and a character represents a 16-bit data item.
    - Also, Java's `char` and `String` types naturally handle characters instead of bytes.
    - More importantly, byte streams have no knowledge of *character sets* (sets of mappings between integer values, known as *code points*, and symbols, such as Unicode) and their *character encodings* (mappings between the members of a character set and sequences of bytes that encode these characters for efficiency, such as UTF-8).
  - If you need to stream characters, you should take advantage of Java's writer and reader classes, which were designed to support character I/O (they work with `char` instead of `byte`).
  - Furthermore, the writer and reader classes take character encodings into account.

# A BRIEF HISTORY OF CHARACTER SETS AND CHARACTER ENCODINGS

---

- Early computers and programming languages were created mainly by English-speaking programmers in countries where English was the native language.
- They developed a standard mapping between code points 0 through 127 and the 128 commonly used characters in the English language (such as A–Z).
- The resulting character set/encoding was named *American Standard Code for Information Interchange (ASCII)*.
- The problem with ASCII is that it's inadequate for most non-English languages. For example, ASCII doesn't support diacritical marks such as the cedilla used in French. Because a byte can represent a maximum of 256 different characters, developers around the world started creating different character sets/encodings that encoded the 128 ASCII characters, but also encoded extra characters to meet the needs of languages such as French, Greek, and Russian. Over the years, many legacy (and still important) data files have been created whose bytes represent characters defined by specific character sets/encodings.
- The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) worked to standardize these 8-bit character sets/encodings under a joint umbrella standard called ISO/IEC 8859. The result is a series of substandards named ISO/IEC 8859-1, ISO/IEC 8859-2, and so on. For example, ISO/IEC 8859-1 (also known as Latin-1) defines a character set/encoding that consists of ASCII plus the characters covering most Western European countries. Also, ISO/IEC 8859-2 (also known as Latin-2) defines a similar character set/encoding covering Central and Eastern European countries.

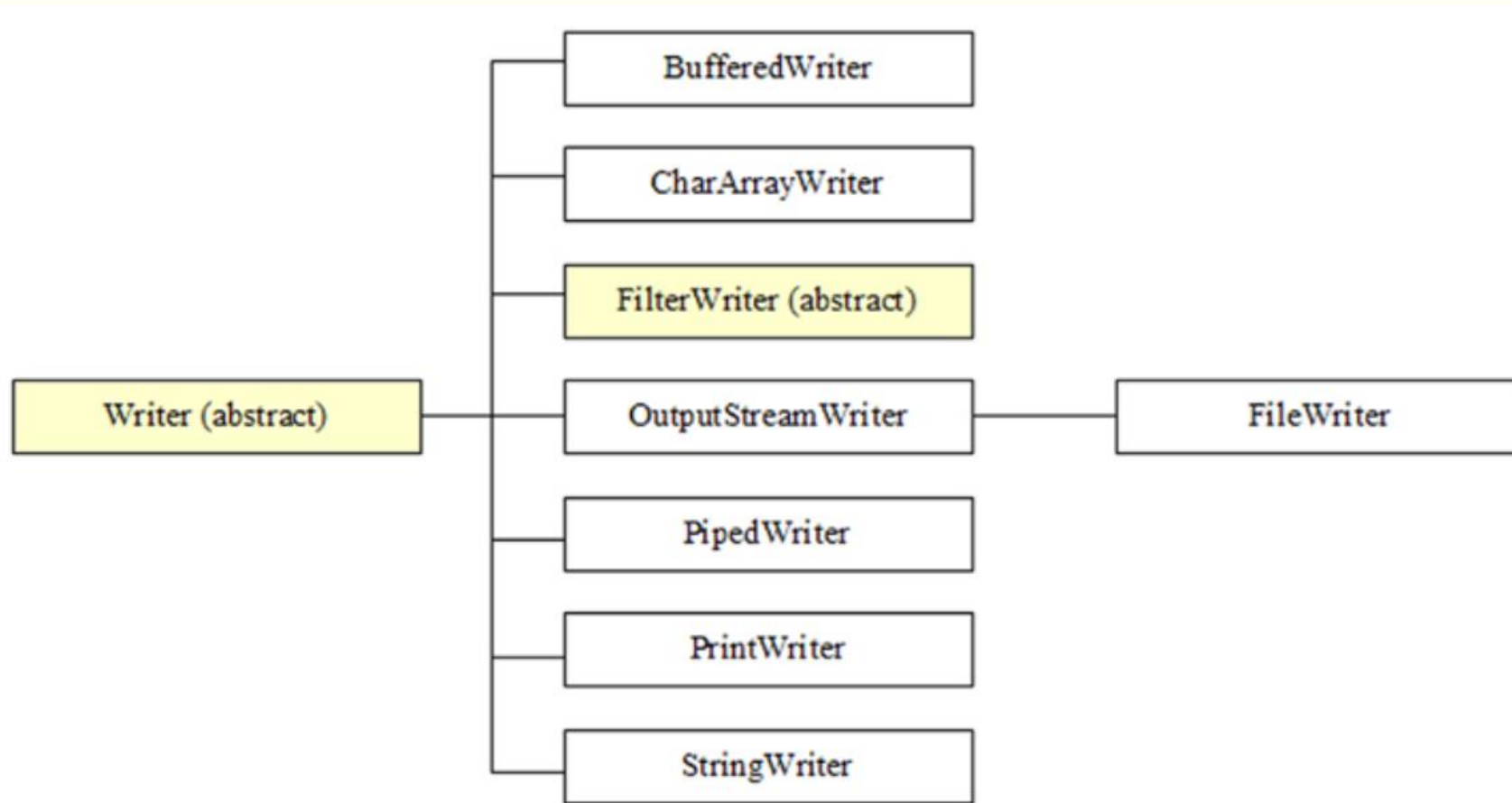
# A BRIEF HISTORY OF CHARACTER SETS AND CHARACTER ENCODINGS

---

- Despite the ISO's/IEC's best efforts, a plethora of character sets/encodings is still inadequate.
  - For example, most character sets/encodings only allow you to create documents in a combination of English and one other language (or a small number of other languages).
  - You cannot, for example, use an ISO/IEC character set/encoding to create a document using a combination of English, French, Turkish, Russian, and Greek characters.
- This and other problems are being addressed by an international effort that has created and is continuing to develop *Unicode*, a single universal character set.
  - Because Unicode characters are bigger than ISO/IEC characters, Unicode uses one of several variablelength encoding schemes known as *Unicode Transformation Format (UTF)* to encode Unicode characters for efficiency. For example, UTF-8 encodes every character in the Unicode character set in one to four bytes (and is backward-compatible with ASCII).
- Finally, the terms *character set* and *character encoding* are often used interchangeably.
  - They mean the same thing in the context of ISO/IEC character sets in which a code point is the encoding.
  - However, these terms are different in the context of Unicode in which Unicode is the character set and UTF-8 is one of several possible character encodings for Unicode characters.

## hierarchy of writer classes

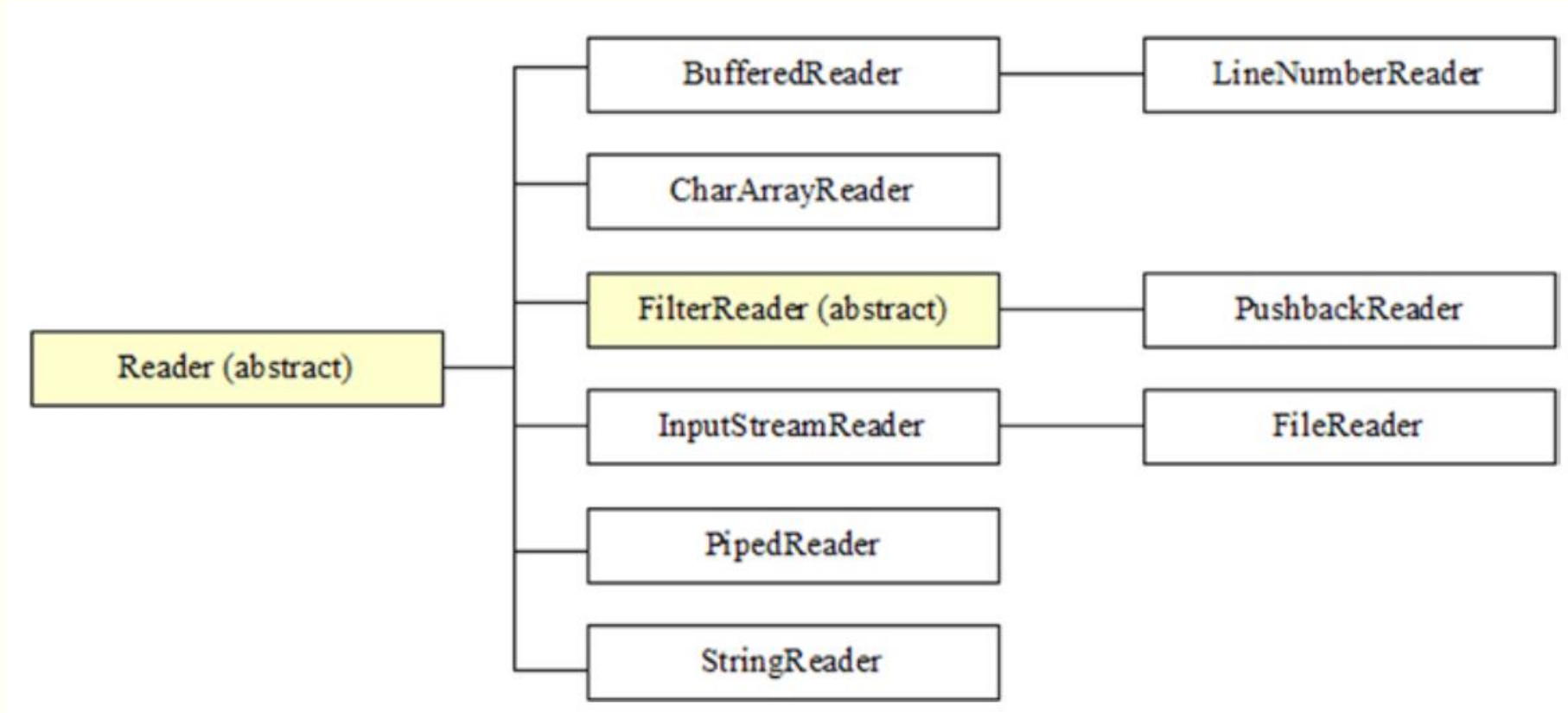
---



- Unlike *java.io.FilterOutputStream*, *FilterWriter* is abstract

## hierarchy of reader classes

---



- Unlike *java.io.FilterInputStream*, *FilterReader* is abstract

- 
- Although the writer and reader class hierarchies are similar to their output stream and input stream counterparts, there are differences.
    - For example, `FilterWriter` and `FilterReader` are abstract, whereas their `FilterOutputStream` and `FilterInputStream` equivalents are not abstract.
    - Also, `BufferedWriter` and `BufferedReader` don't extend `FilterWriter` and `FilterReader`, whereas `java.io.BufferedOutputStream` and `java.io.BufferedInputStream` extend `FilterOutputStream` and `FilterInputStream`.
  - The output stream and input stream classes were introduced in Java 1.0.
    - After their release, design issues emerged.
    - For example, `FilterOutputStream` and `FilterInputStream` should have been abstract.
    - However, it was too late to make these changes because the classes were already being used; making these changes would have resulted in broken code.
    - The designers of Java 1.1's writer and reader classes took the time to correct these mistakes

- 
- 
- Regarding `BufferedWriter` and `BufferedReader` directly subclassing `Writer` and `Reader` instead of `FilterWriter` and `FilterReader`, I believe that this change has to do with performance.
  - Calls to `BufferedOutputStream`'s `write()` methods and `BufferedInputStream`'s `read()` methods result in calls to `FilterOutputStream`'s `write()` methods and `FilterInputStream`'s `read()` methods.
  - Because a file I/O activity such as copying one file to another can involve many `write()/read()` method calls, you want the best performance possible.
  - By not subclassing `FilterWriter` and `FilterReader`, `BufferedWriter` and `BufferedReader` achieve better performance.

# Writer and Reader

---

- Java provides the Writer and Reader classes for performing character I/O.
  - Writer is the superclass of all writer subclasses.
- The following list identifies the differences between Writer and `java.io.OutputStream`:
  - Writer declares several `append()` methods for appending characters to this writer. These methods exist because Writer implements the `java.lang.Appendable` interface, which is used in partnership with the `java.util.Formatter` class to output formatted strings.
  - Writer declares additional `write()` methods, including a convenient `void write(String str)` method for writing a String object's characters to this writer.
- Reader is the superclass of all reader subclasses. The following list identifies differences between Reader and `java.io.InputStream`:
  - Reader declares `read(char[])` and `read(char[], int, int)` methods instead of `read(byte[])` and `read(byte[], int, int)` methods.
  - Reader doesn't declare an `available()` method.
  - Reader declares a boolean `ready()` method that returns true when the next `read()` call is guaranteed not to block for input.
  - Reader declares an `int read(CharBuffer target)` method for reading characters from a character buffer.

# OutputStreamWriter and InputStreamReader

---

- The concrete OutputStreamWriter class (a Writer subclass) is a bridge between an incoming sequence of characters and an outgoing stream of bytes.
- Characters written to this writer are encoded into bytes according to the default or specified character encoding.
- **Note** The default character encoding is accessible via the file.encoding system property.
- Each call to one of OutputStreamWriter's write() methods causes an encoder to be called on the given character(s).
- The resulting bytes are accumulated in a buffer before being written to the underlying output stream.
- The characters passed to the write() methods are not buffered.

- 
- `OutputStreamWriter` declares four constructors, including the following pair:
    - `OutputStreamWriter(OutputStream out)` creates a bridge between an incoming sequence of characters (passed to `OutputStreamWriter` via its `append()` and `write()` methods) and the underlying output stream `out`. The default character encoding is used to encode characters into bytes.
    - `OutputStreamWriter(OutputStream out, String charsetName)` creates a bridge between an incoming sequence of characters (passed to `OutputStreamWriter` via its `append()` and `write()` methods) and the underlying output stream `out`. `charsetName` identifies the character encoding used to encode characters into bytes. This constructor throws `java.io.UnsupportedEncodingException` when the named character encoding isn't supported.
  - **Note** `OutputStreamWriter` depends on the abstract `java.nio.charset.Charset` and `java.nio.charset.CharsetEncoder` classes to perform character encoding.

- 
- 
- The following example uses the second constructor to create a bridge to an underlying file output stream so that Polish text can be written to an ISO/IEC 8859-2-encoded file.
  - ```
FileOutputStream fos = new FileOutputStream("polish.txt");  
OutputStreamWriter osw = new OutputStreamWriter(fos, "8859_2");  
char ch = '\u0323'; // Accented N.  
osw.write(ch);
```
  - The concrete `InputStreamReader` class (a `Reader` subclass) is a bridge between an incoming stream of bytes and an outgoing sequence of characters. Characters read from this reader are decoded from bytes according to the default or specified character encoding.
  - Each call to one of `InputStreamReader`'s `read()` methods may cause one or more bytes to be read from the underlying input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation

- 
- InputStreamReader declares four constructors, including the following pair:
    - InputStreamReader(InputStream in) creates a bridge between the underlying input stream in and an outgoing sequence of characters (returned from InputStreamReader via its read() methods). The default character encoding is used to decode bytes into characters.
    - InputStreamReader(InputStream in, String charsetName) creates a bridge between the underlying input stream in and an outgoing sequence of characters (returned from InputStreamReader via its read() methods). charsetName identifies the character encoding used to decode bytes into characters. This constructor throws UnsupportedEncodingException when the named character encoding is not supported.
  - **Note** InputStreamReader depends on the abstract Charset and java.nio.charset.CharsetDecoder classes to perform character decoding.

- 
- The following example uses the second constructor to create a bridge to an underlying file input stream so that Polish text can be read from an ISO/IEC 8859-2-encoded file.

```
FileInputStream fis = new FileInputStream("polish.txt");  
InputStreamReader isr = new InputStreamReader(fis, "8859_2");  
char ch = isr.read(ch);
```

- **Note** OutputStreamWriter and InputStreamReader declare a String getEncoding() method that returns the name of the character encoding in use.
  - If the encoding has a historical name, that name is returned; otherwise, the encoding's canonical name is returned.

# FileWriter and FileReader

---

- `FileWriter` is a convenience class for writing characters to files. It subclasses `OutputStreamWriter`, and its constructors, such as `FileWriter(String path)`, call `OutputStreamWriter(OutputStream)`.
- An instance of this class is equivalent to the following code fragment:  

```
FileOutputStream fos = new FileOutputStream(path);  
OutputStreamWriter osw;  
osw = new OutputStreamWriter(fos, System.getProperty("file.encoding"));
```
- `FileReader` is a convenience class for reading characters from files. It subclasses `InputStreamReader`, and its constructors, such as `FileReader(String path)`, call `InputStreamReader(InputStream)`.
- An instance of this class is equivalent to the following code fragment:  

```
FileInputStream fis = new FileInputStream(path);  
InputStreamReader isr;  
isr = new InputStreamReader(fis, System.getProperty("file.encoding"));
```

- 
- 
- Neither `FileWriter` nor `FileReader` supply their own methods. Instead, you call their inherited methods, such as the following:
    - `void write(String str, int off, int len)`: Write `len` characters of string `str` starting at zero-based offset `off`. Throw `java.io.IOException` when an I/O error occurs.
    - `int read(char[] cbuf, int off, int len)`: Read `len` characters into `cbuf` starting at zero-based offset `off`. Throw `IOException` when an I/O error occurs.

## *Demonstrating the **FileWriter** and **FileReader** Classes*

---

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FWFRDemo
{
    final static String MSG = "Test message";

    public static void main(String[] args) throws IOException
    {
        try (FileWriter fw = new FileWriter("temp"))
        {
            fw.write(MSG, 0, MSG.length());
        }
        char[] buf = new char[MSG.length()];
        try (FileReader fr = new FileReader("temp"))
        {
            fr.read(buf, 0, MSG.length());
            System.out.println(buf);
        }
    }
}
```

- 
- 
- FWFRDemo first creates a `FileWriter` instance connected to a file named `temp`.
  - It then invokes `void write(String str, int off, int len)` to write a message to this file. The `try-with-resources` statement automatically closes the file following this operation.
  - Next, FWFRDemo creates a buffer for storing a line of text, and then creates a `FileReader` instance connected to `temp`. It then invokes `int read(char[] cbuf, int off, int len)` to read the previously written message and output it to the standard output stream. The file is then closed.
  - You should observe the following output (and a file named `temp`):  
Test message

# BufferedWriter and BufferedReader

---

- `BufferedWriter` writes text to a character-output stream (a `Writer` instance), buffering characters so as to provide for the efficient writing of single characters, arrays, and strings. Invoke either of the following constructors to construct a buffered writer:
  - `BufferedWriter(Writer out)`
  - `BufferedWriter(Writer out, int size)`The buffer size may be specified, or the default size (8,192 bytes) may be accepted. The default is large enough for most purposes.
- `BufferedWriter` includes a handy `void newLine()` method for writing a line-separator string, which effectively terminates the current line.
- `BufferedReader` reads text from a character-input stream (a `Reader` instance), buffering characters so as to provide for the efficient reading of characters, arrays, and lines. Invoke either of the following constructors to construct a buffered reader:
  - `BufferedReader(Reader in)`
  - `BufferedReader(Reader in, int size)`
- The buffer size may be specified, or the default size (8,192 bytes) may be used. The default is large enough for most purposes. `BufferedReader` includes a handy `String readLine()` method for reading a line of text, not including any line-termination characters.

```
public class BWBRDemo
{
    static String[] lines =
    {
        "It was the best of times, it was the worst of times,",
        "it was the age of wisdom, it was the age of foolishness,",
        "it was the epoch of belief, it was the epoch of incredulity,",
        "it was the season of Light, it was the season of Darkness,",
        "it was the spring of hope, it was the winter of despair."
    };
    public static void main(String[] args) throws IOException
    {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("temp")))
        {
            for (String line: lines)
            {
                bw.write(line, 0, line.length());
                bw.newLine();
            }
        }
        try (BufferedReader br = new BufferedReader(new FileReader("temp")))
        {
            String line;
            while ((line = br.readLine()) != null)
                System.out.println(line);
        }
    }
}
```

- 
- BWBRDemo first creates a BufferedWriter instance that wraps a created FileWriter instance that is connected to a file named temp. It then iterates over the line of strings, writing each line followed by a newline sequence.
  - Next, BWBRDemo creates a BufferedReader instance that wraps a created FileReader instance that is connected to temp. It then reads and outputs each line from the file until readLine() returns null.

```
It was the best of times, it was the worst of times,  
it was the age of wisdom, it was the age of foolishness,  
it was the epoch of belief, it was the epoch of incredulity,  
it was the season of Light, it was the season of Darkness,  
it was the spring of hope, it was the winter of despair.
```