Basic Input & Output (I/O)

Agenda

- File class
- What is an I/O stream?
- Types of Streams
- Stream class hierarchy
- Control flow of an I/O operation using Streams
- Byte streams
- Character streams
- Buffered streams
- Standard I/O streams
- Data streams
- Object streams
- Serialization

File and Directory

- **Class *java.io.File***
  - **The class *java.io.File* can represent either a file or a directory**
  - **A *path string* is used to locate a *file* or a *directory***

File and Directory

- Unfortunately, path strings are system dependent
  - Windows use back-slash '\' as the directory separator; while Unixes/Mac use forward-slash '/'
    - E.g., "c:\myproject\java\Hello.java" in Windows or "/myproject/java/Hello.java" in Unix/Mac
  - Windows use semi-colon ';' as path separator to separate a list of paths; while Unixes/Mac use colon ':'
  - The "c:\" or "\" is called the root. Windows supports multiple roots, each maps to a drive (e.g., "c:\", "d:\"). Unixes/Mac has a single root ("\")
  - Windows use "\r\n" as line delimiter for text file; while Unixes use "\n" and Mac uses "\r"

File and Directory

- A path could be absolute (beginning from the root) or relative (which is relative to a reference directory)
  - Special notations "." and ".." denote the current directory and the parent directory

- The *java.io.File* class maintains these system-dependent properties for you to write programs that are portable:
  - *Directory Separator*: in static fields *File.separator* (as String) and *File.separatorChar*. As mentioned, Windows use backslash '\'; while Unixes/Mac use forward slash '/'
  - *Path Separator*: in static fields *File.pathSeparator* (as String) and *File.pathSeparatorChar*. As mentioned, Windows use semi-colon ';' to separate a list of paths; while Unixes/Mac use colon ':'

File and Directory

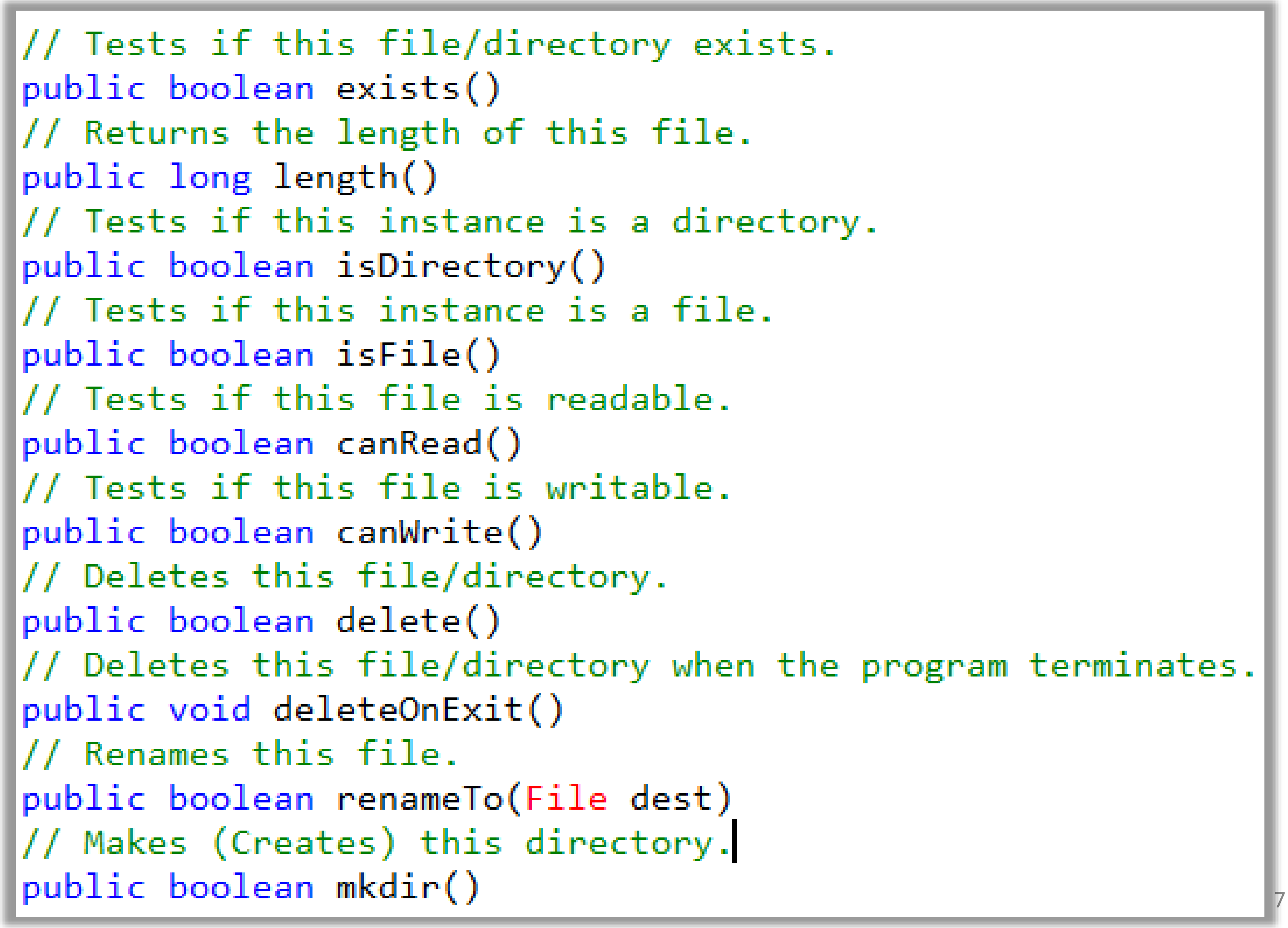
- We can construct a File instance with a path string or URI, as follows

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Constructs a File instance based on the given path string.

public File(URI uri)
// Constructs a File instance by converting from the given file-URI "file://...."
```

- For example,

```
File file = new File("in.txt");      // A file relative to the current working directory
File file = new File("d:\\myproject\\java\\Hello.java");  // A file with absolute path
File dir  = new File("c:\\temp");    // A directory
```

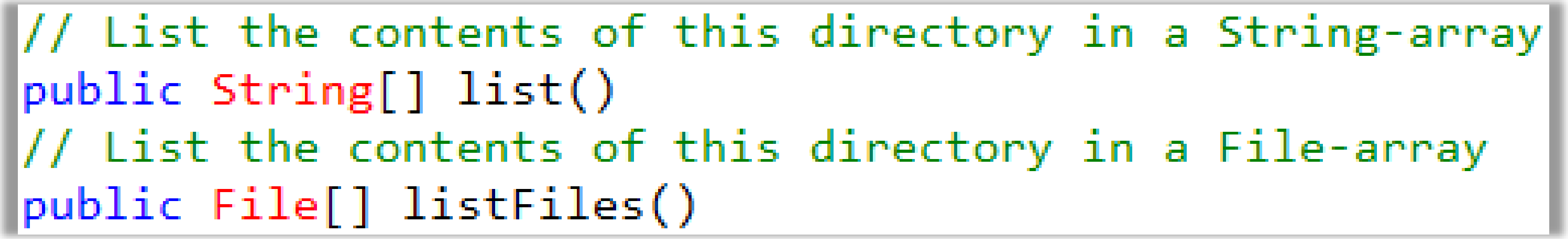
Verifying Properties of a File/Directory



```
// Tests if this file/directory exists.
public boolean exists()
// Returns the length of this file.
public long length()
// Tests if this instance is a directory.
public boolean isDirectory()
// Tests if this instance is a file.
public boolean isFile()
// Tests if this file is readable.
public boolean canRead()
// Tests if this file is writable.
public boolean canWrite()
// Deletes this file/directory.
public boolean delete()
// Deletes this file/directory when the program terminates.
public void deleteOnExit()
// Renames this file.
public boolean renameTo(File dest)
// Makes (Creates) this directory.
public boolean mkdir()
```

List Directory

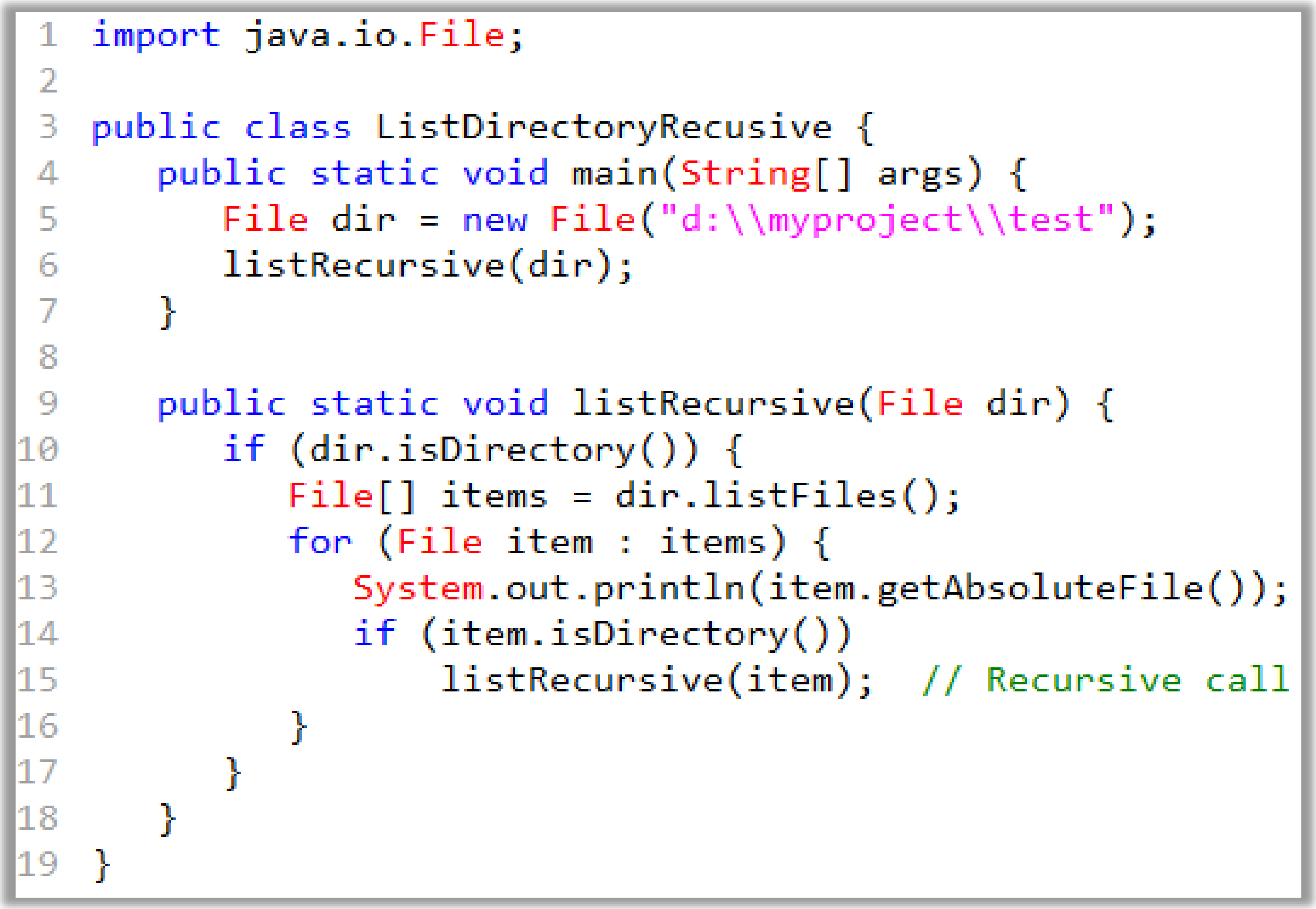
- For a directory, you can use the following methods to list its contents:



```
// List the contents of this directory in a String-array
public String[] list()
// List the contents of this directory in a File-array
public File[] listFiles()
```

List Directory (Cont.)

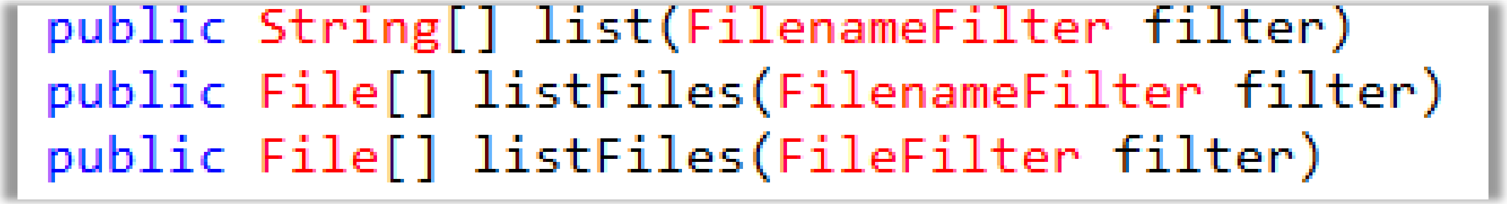
- **Example: The following program recursively lists the contents of a given directory (similar to Unix's "ls -r" command)**



```java
import java.io.File;

public class ListDirectoryRecusive {
    public static void main(String[] args) {
        File dir = new File("d:\\myproject\\test");
        listRecursive(dir);
    }

    public static void listRecursive(File dir) {
        if (dir.isDirectory()) {
            File[] items = dir.listFiles();
            for (File item : items) {
                System.out.println(item.getAbsoluteFile());
                if (item.isDirectory())
                    listRecursive(item);  // Recursive call
            }
        }
    }
}
```

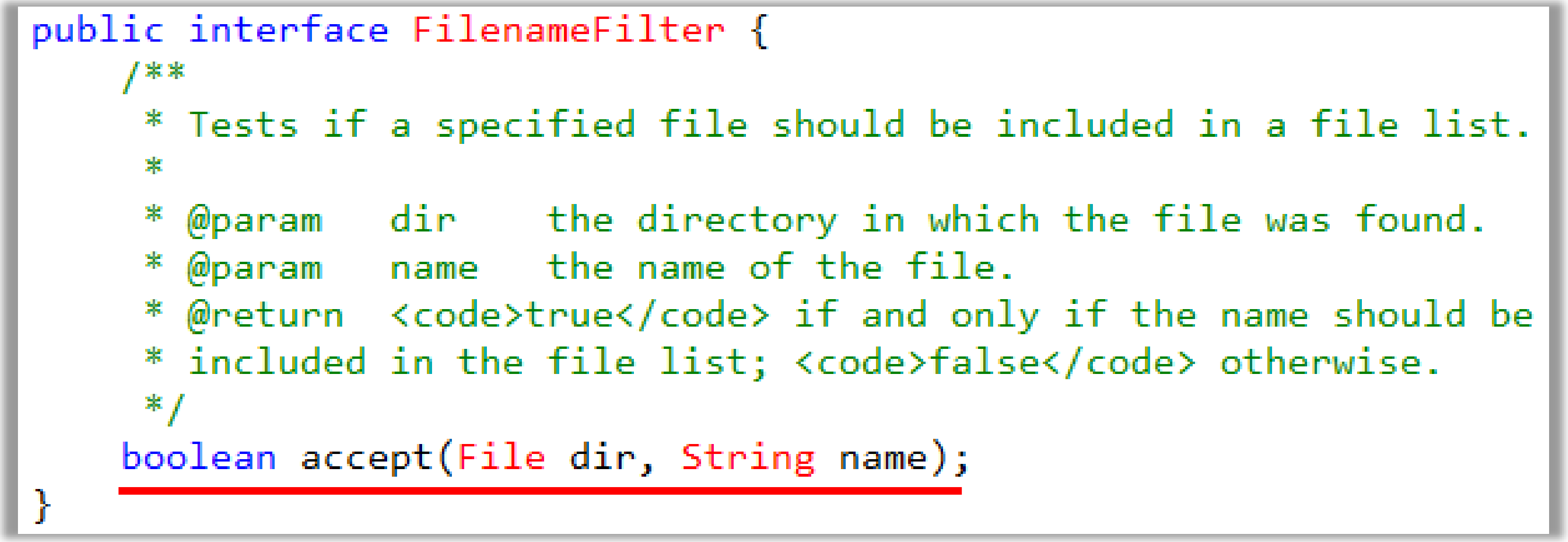
List Directory with Filter

- Apply a filter to list() and listFiles(), to list only files that meet a certain criteria



```java
public String[] list(FilenameFilter filter)
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
```

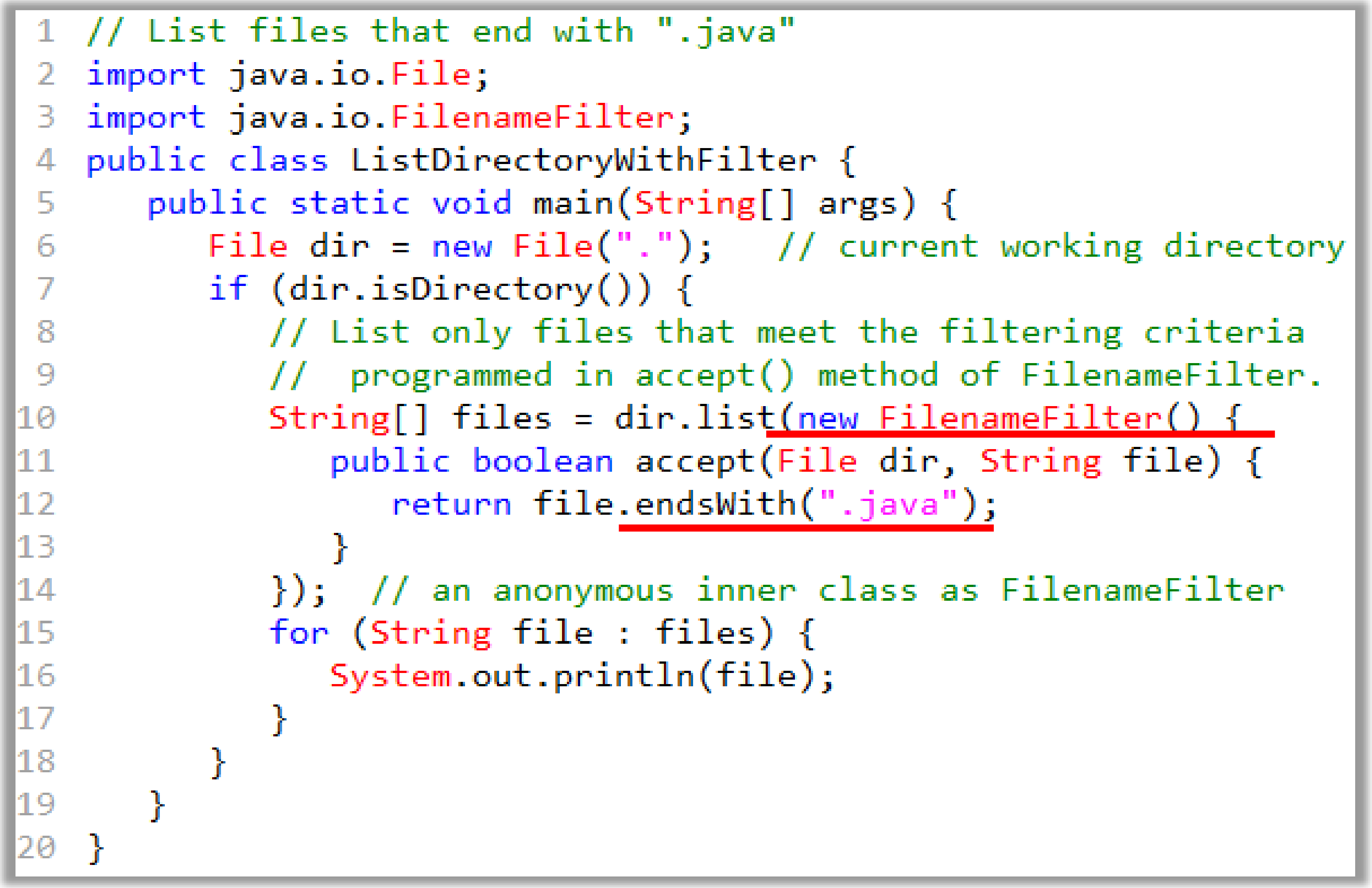
- The interface ***java.io.FilenameFilter*** declares one abstract method:



```java
public interface FilenameFilter {
    /**
     * Tests if a specified file should be included in a file list.
     *
     * @param   dir    the directory in which the file was found.
     * @param   name   the name of the file.
     * @return  <code>true</code> if and only if the name should be
     * included in the file list; <code>false</code> otherwise.
     */
    boolean accept(File dir, String name);
}
```

List Directory with Filter (Cont.)

- **Example: The following program lists only files that meet a certain filtering criteria**



```java
// List files that end with ".java"
import java.io.File;
import java.io.FilenameFilter;
public class ListDirectoryWithFilter {
   public static void main(String[] args) {
      File dir = new File(".");   // current working directory
      if (dir.isDirectory()) {
         // List only files that meet the filtering criteria
         //  programmed in accept() method of FilenameFilter.
         String[] files = dir.list(new FilenameFilter() {
            public boolean accept(File dir, String file) {
               return file.endsWith(".java");
            }
         });  // an anonymous inner class as FilenameFilter
         for (String file : files) {
            System.out.println(file);
         }
      }
   }
}
```

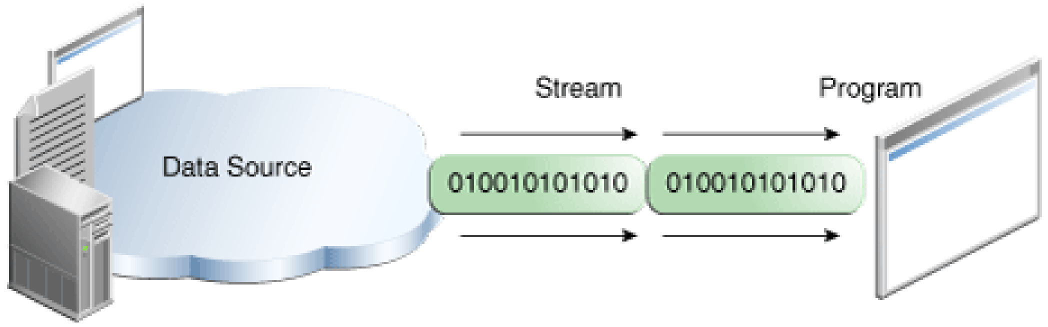
I/O Streams

- An I/O Stream represents an input source or an output destination

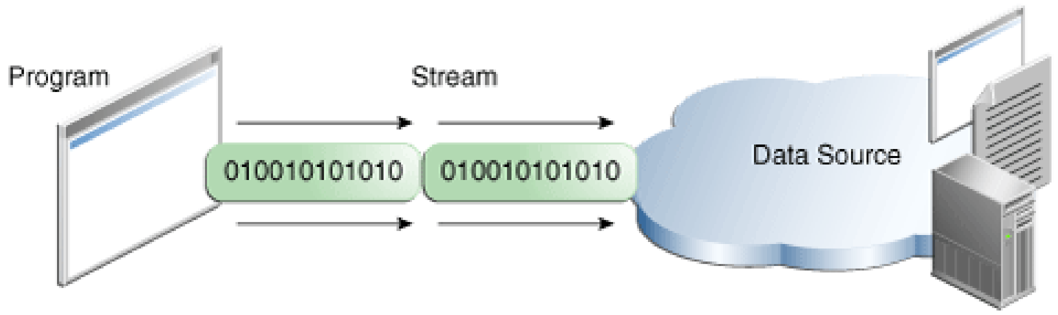
- A stream can represent many different kinds of sources and destinations:
  - Display Console
  - File
  - Memory Buffer
  - Network
  - Other programs
  - Etc.

I/O Streams (Cont.)

- Reading information into a program (INPUT)



- Writing information from a program (OUTPUT)



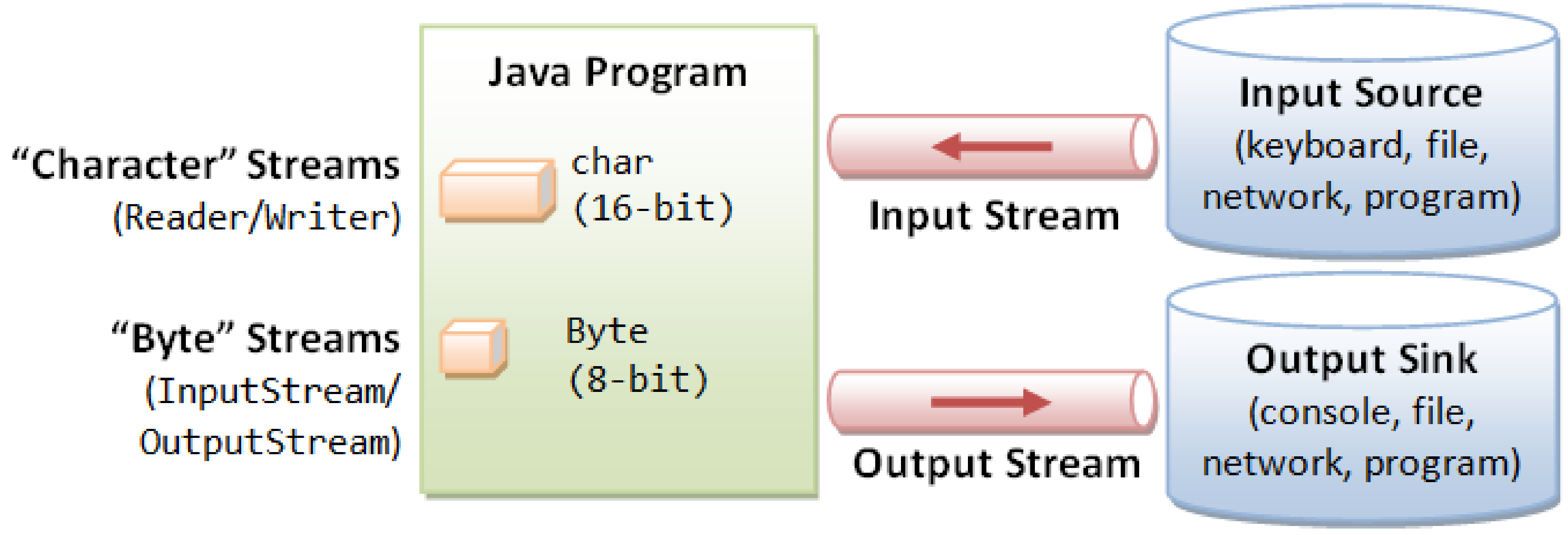
I/O Streams (Cont.)

- A stream is a sequential and contiguous one-way flow of data
- Java does not differentiate between the various types of data sources or sinks in stream I/O
- Input and output streams can be established from/to any data source/sink
  - such as files, network, keyboard/console or another program
- The Java program receives data from a source by opening an input stream, and sends data to a sink by opening an output stream

I/O Streams (Cont.)

- **Stream I/O operations involve three steps:**
  - **Open a stream with associated source**
  - **Read from the opened input stream until "end-of-stream" encountered, or write to the opened output**
  - **Close the stream**

I/O Streams types



Internal Data Formats:

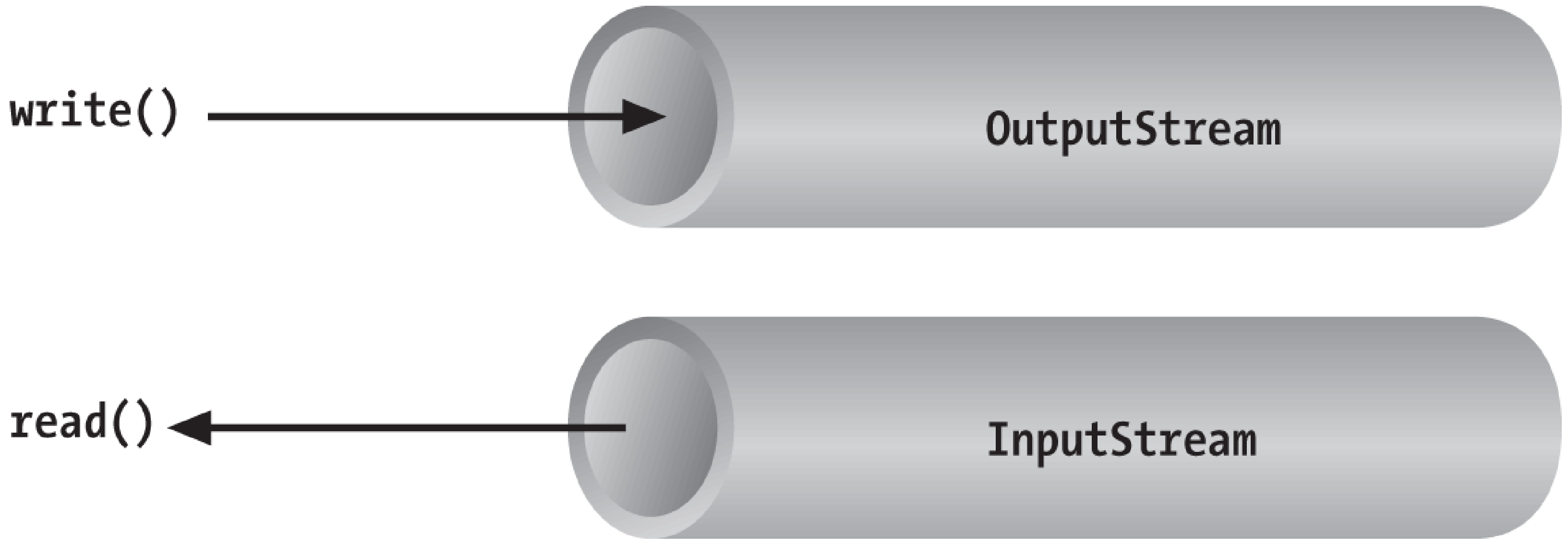
- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Byte-Based I/O & Byte Streams

- **8 bits, data-based**
- **Two parent *abstract* classes:**
  - ***InputStream***
  - ***OutputStream***



Reading from an InputStream

- **Reading bytes:**
  - ***InputStream*** **class defines an** ***abstract*** **method**

```
public abstract int read() throws IOException
```

    - **Designer of a concrete input stream class overrides this method to provide useful functionality**
    - **E.g. in the** ***FileInputStream*** **class, the method reads one byte from a file**

  - **The** ***read()*** **method** ***blocks*** **until a byte is available, an I/O error occurs, or the "end-of-stream" is detected**

Reading from an InputStream

- **Reading bytes:**
  - ***InputStream*** **class also contains nonabstract methods to read an array of bytes or skip a number of bytes**
  - **Two variations of *read()* methods are implemented in the *InputStream* for reading a block of bytes into a byte-array**

```
// Read "length" number of bytes, store in bytes array starting from offset of index.
public int read(byte[] bytes, int offset, int length) throws IOException
// Same as read(bytes, 0, bytes.length)
public int read(byte[] bytes) throws IOException
```

Writing to an OutputStream

- **Writing bytes:**
  - ***OutputStream*** **class defines an *abstract* method**

```
public abstract void write(int b) throws IOException
```

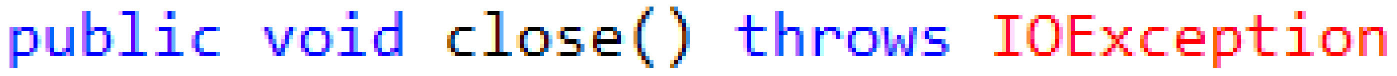
  - ***OutputStream*** **class also contains nonabstract methods for tasks such as writing bytes from a specified byte array**
  - **Two variations of the *write()* method to write a block of bytes from a byte-array are implemented:**

```
// Write "length" number of bytes, from the bytes array starting from offset of index.
public void write(byte[] bytes, int offset, int length) throws IOException
// Same as write(bytes, 0, bytes.length)
public void write(byte[] bytes) throws IOException
```

Opening & Closing I/O Streams

- ***Open*** **an I/O stream by constructing an instance of the stream**

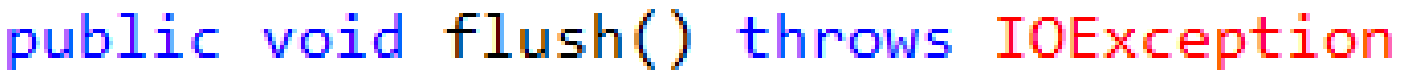
- **Both the *InputStream* and the *OutputStream* provides a *close()* method to close the stream**
  - **Perform the necessary clean-up operations to free up the system resources**



```
public void close() throws IOException
```

Flushing the OutputStream

- the ***OutputStream*** provides a ***flush()*** method to flush the remaining bytes from the output buffer



```
public void flush() throws IOException
```

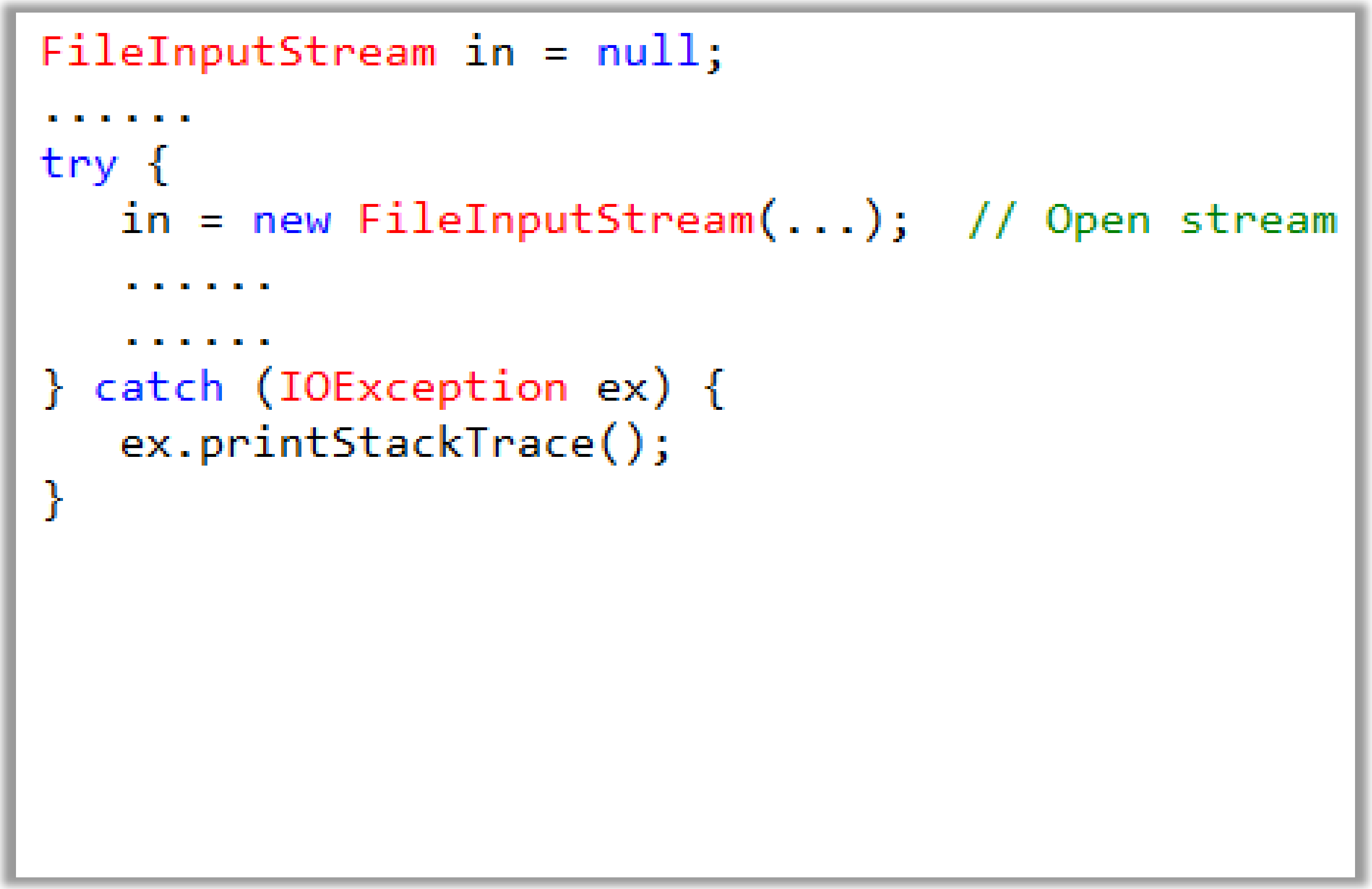
Implementations of abstract InputStream/OutputStream

- *InputStream* and *OutputStream* are abstract classes that cannot be instantiated

- We need to choose an appropriate concrete subclass to establish a connection to a physical device
  - For example, we instantiate a *FileInputStream* or *FileOutputStream* to establish a stream to a physical disk file

Implementations of abstract InputStream/OutputStream

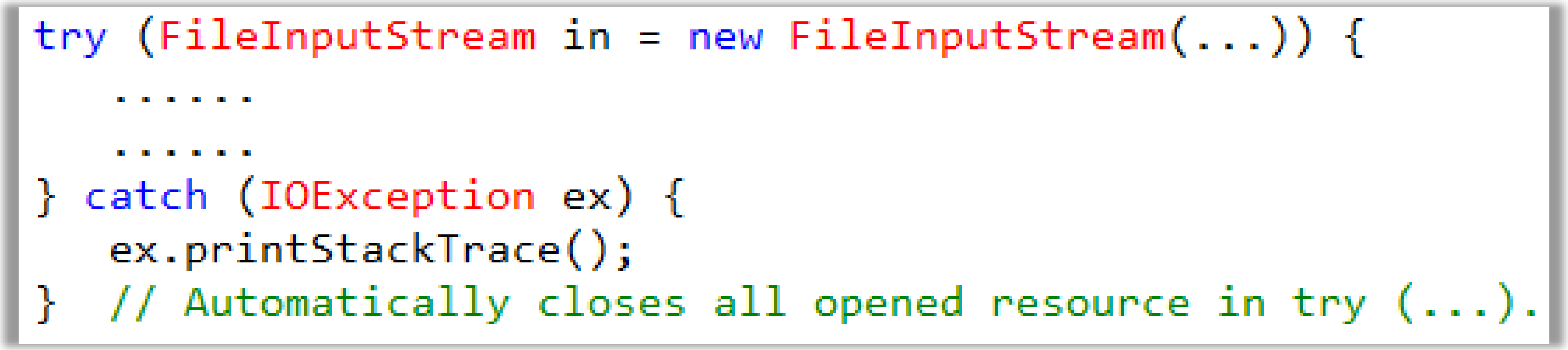
- **Example for Opening and Closing I/O Streams**



```
FileInputStream in = null;
.......
try {
    in = new FileInputStream(...);  // Open stream
    .......
    .......
} catch (IOException ex) {
    ex.printStackTrace();
}
```

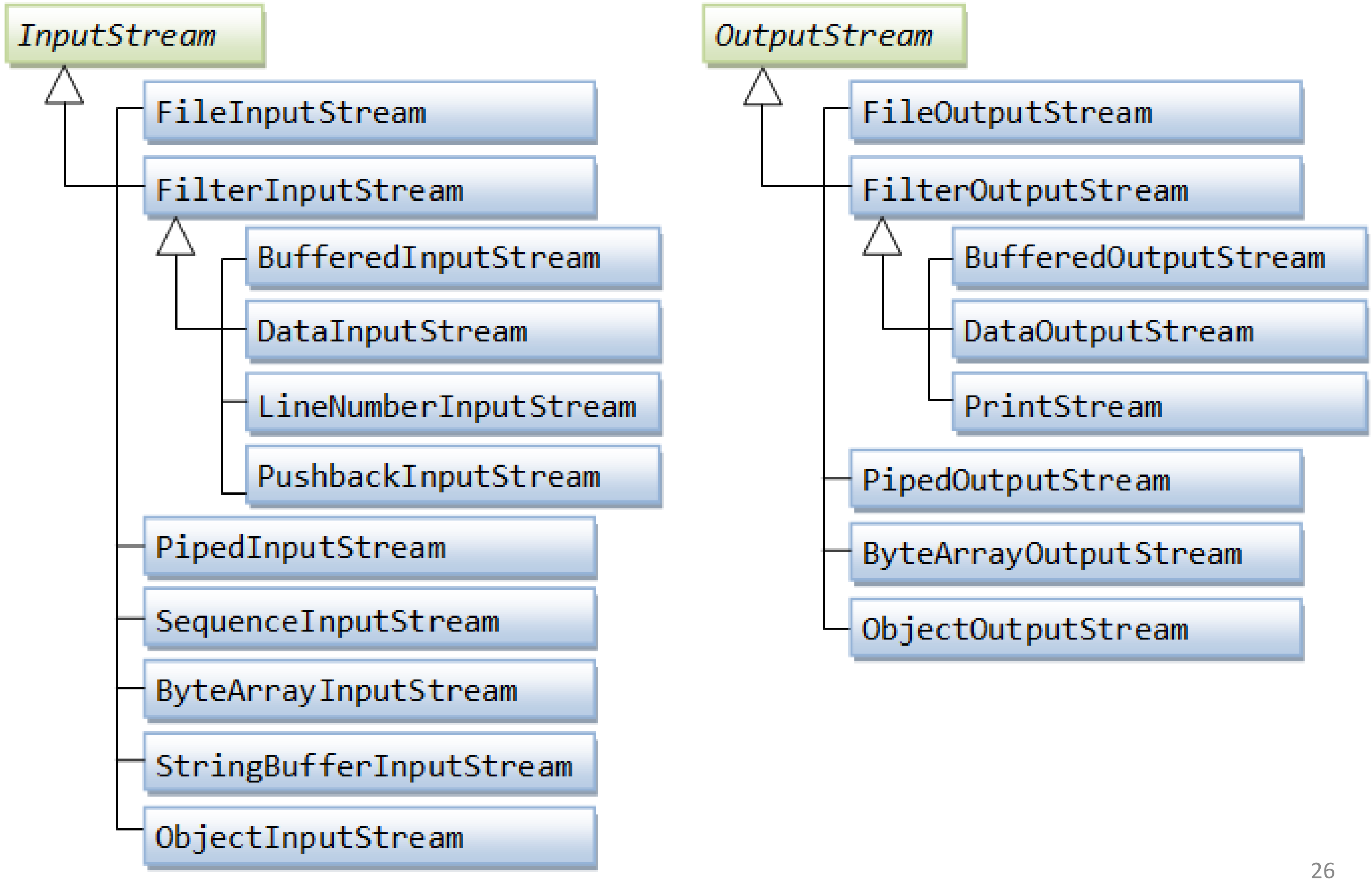
Implementations of abstract InputStream/OutputStream

- **Example for Opening and Closing I/O Streams**
  - **JDK 1.7 introduces a new *try-with-resources* syntax, which automatically closes all the opened resources after try or catch**



```
try (FileInputStream in = new FileInputStream(...)) {
    ......
    ......
} catch (IOException ex) {
    ex.printStackTrace();
}  // Automatically closes all opened resource in try (...).
```

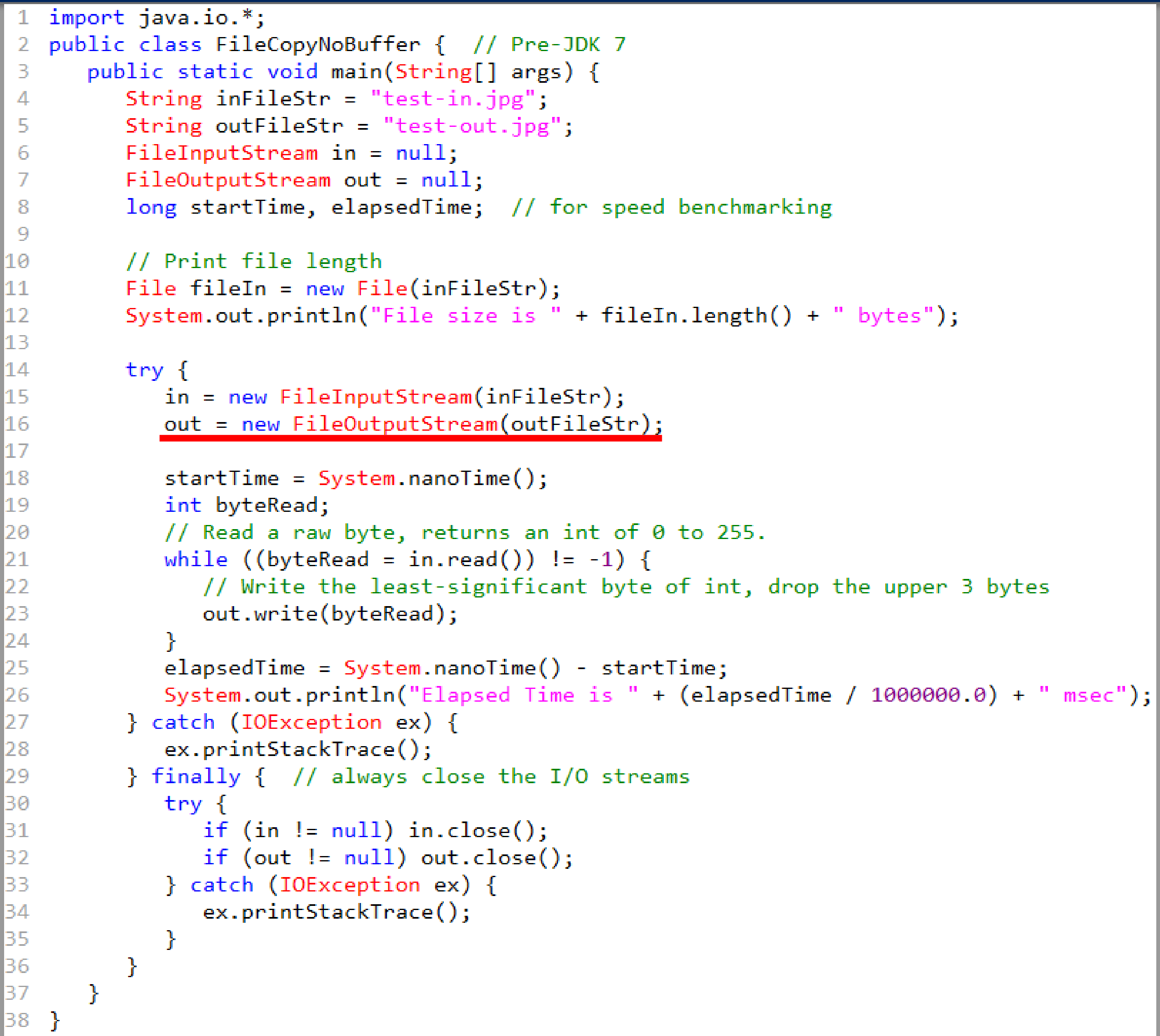
Byte Streams implemnetations



File I/O Byte-Streams

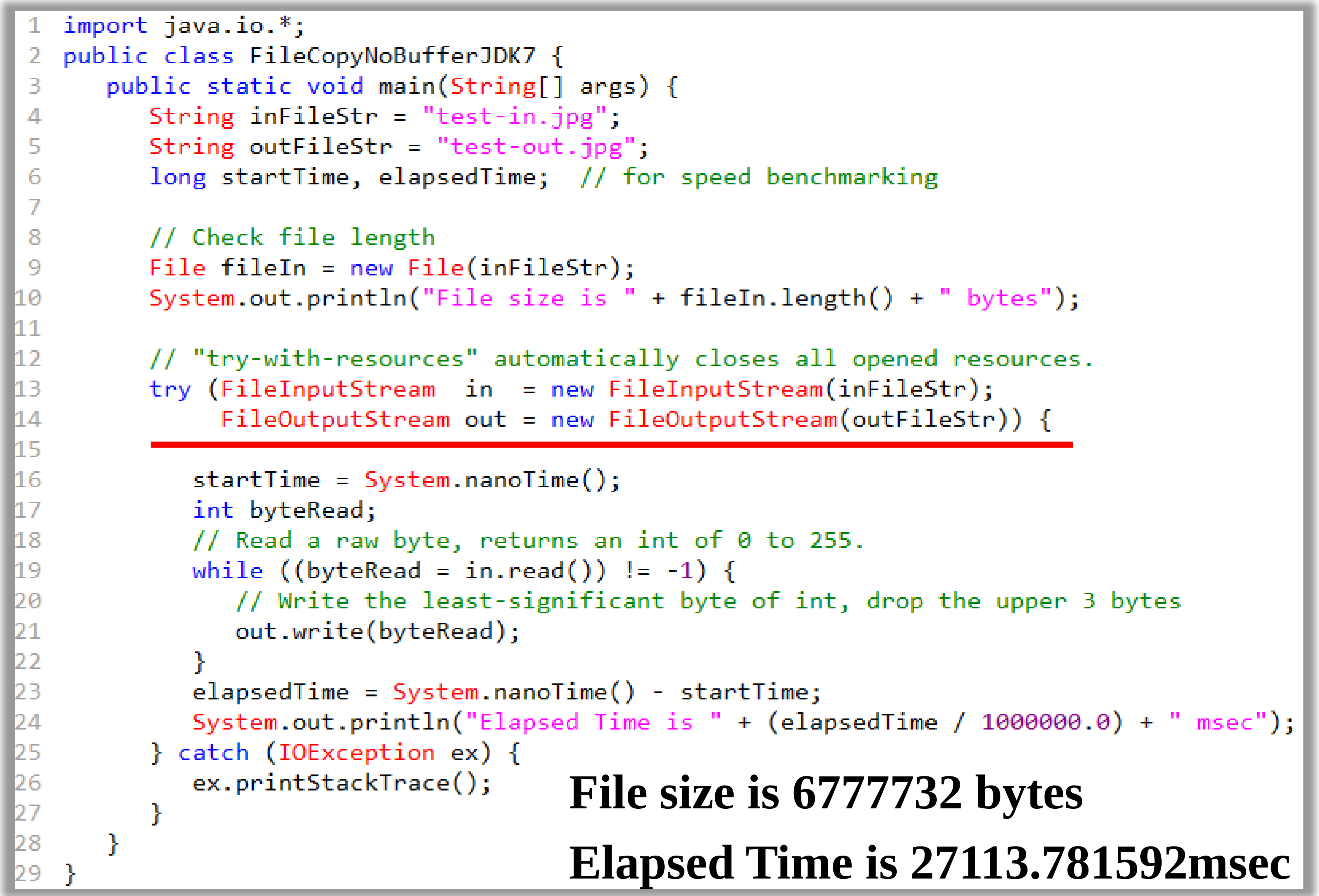
- *FileInputStream* and *FileOutputStream* are concrete implementations to the abstract classes *InputStream* and *OutputStream*, to support I/O from disk files

Copying a file byte-by-byte



```
import java.io.*;
public class FileCopyNoBuffer {  // Pre-JDK 7
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      FileInputStream in = null;
      FileOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      // Print file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try {
         in = new FileInputStream(inFileStr);
         out = new FileOutputStream(outFileStr);

         startTime = System.nanoTime();
         int byteRead;
         // Read a raw byte, returns an int of 0 to 255.
         while ((byteRead = in.read()) != -1) {
            // Write the least-significant byte of int, drop the upper 3 bytes
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {  // always close the I/O streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

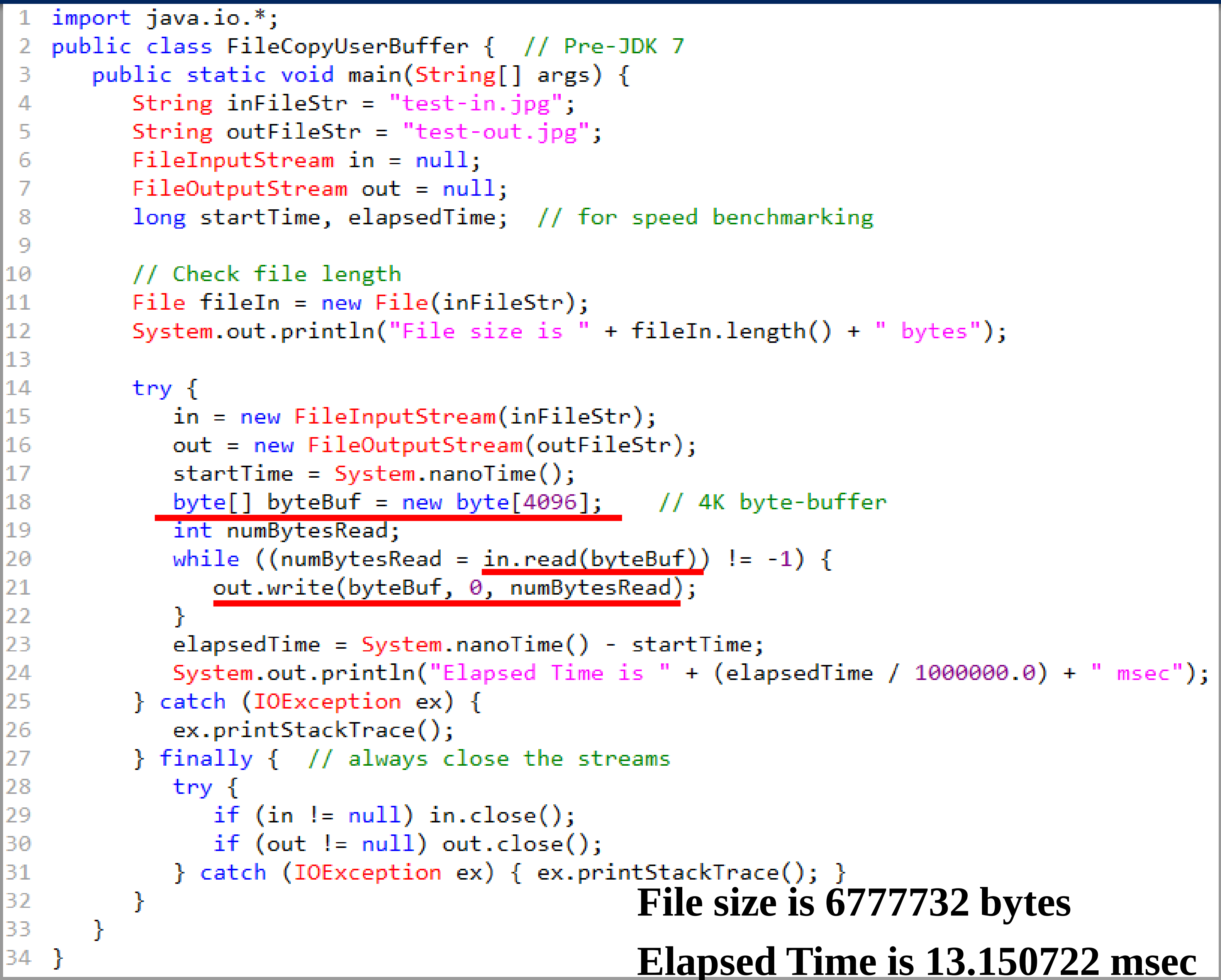
Using *try-with-resources* syntax



```java
import java.io.*;
public class FileCopyNoBufferJDK7 {
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      // "try-with-resources" automatically closes all opened resources.
      try (FileInputStream  in  = new FileInputStream(inFileStr);
           FileOutputStream out = new FileOutputStream(outFileStr)) {

         startTime = System.nanoTime();
         int byteRead;
         // Read a raw byte, returns an int of 0 to 255.
         while ((byteRead = in.read()) != -1) {
            // Write the least-significant byte of int, drop the upper 3 bytes
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

```
File size is 6777732 bytes
Elapsed Time is 27113.781592msec
```

Copying a file with a Programmer-Managed Buffer

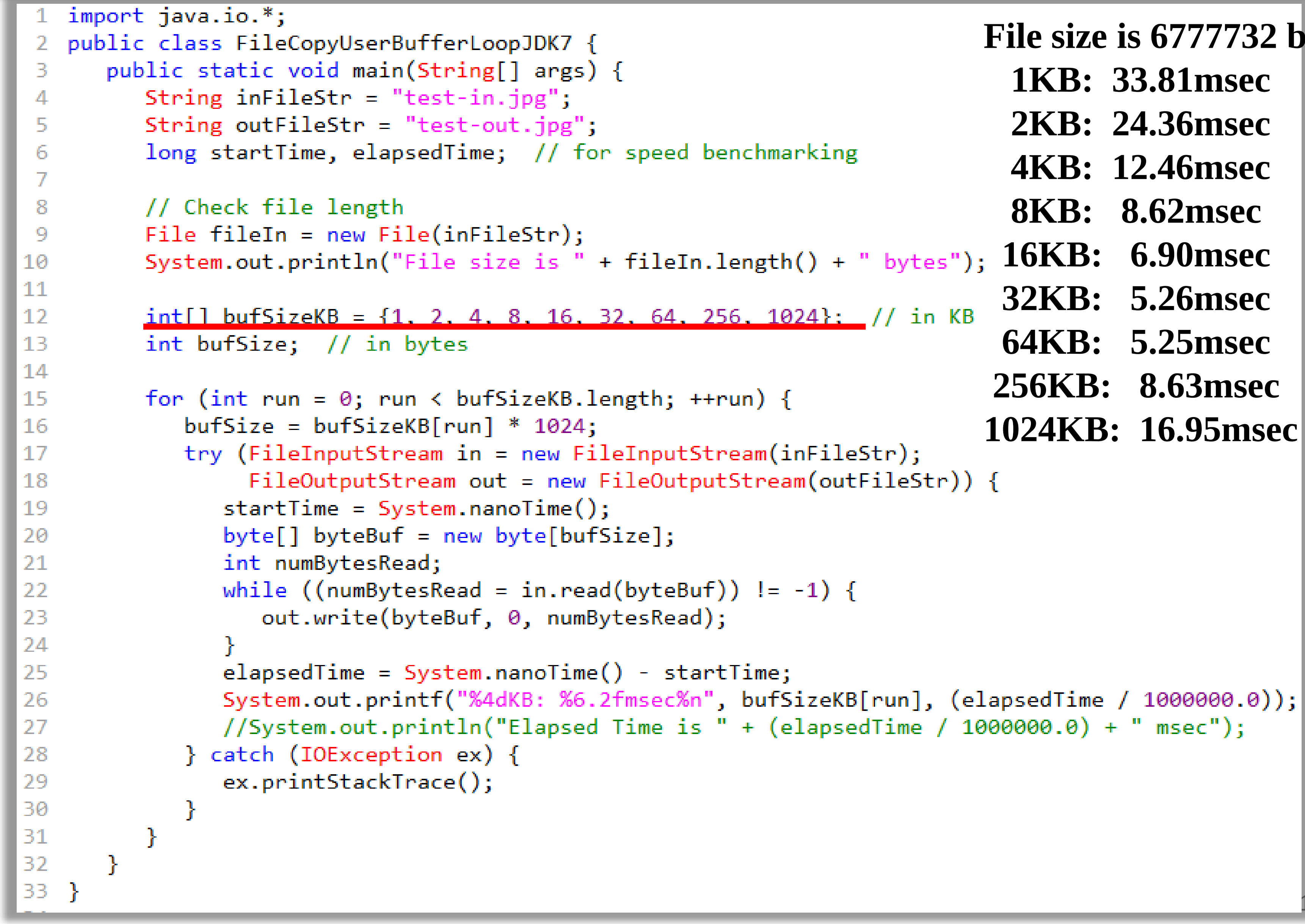


```java
import java.io.*;
public class FileCopyUserBuffer {  // Pre-JDK 7
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      FileInputStream in = null;
      FileOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try {
         in = new FileInputStream(inFileStr);
         out = new FileOutputStream(outFileStr);
         startTime = System.nanoTime();
         byte[] byteBuf = new byte[4096];    // 4K byte-buffer
         int numBytesRead;
         while ((numBytesRead = in.read(byteBuf)) != -1) {
            out.write(byteBuf, 0, numBytesRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {  // always close the streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) { ex.printStackTrace(); }
      }
   }
}
```

File size is 6777732 bytes

Elapsed Time is 13.150722 msec

Copying a file using various buffer sizes



```java
import java.io.*;
public class FileCopyUserBufferLoopJDK7 {
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      int[] bufSizeKB = {1, 2, 4, 8, 16, 32, 64, 256, 1024};  // in KB
      int bufSize;  // in bytes

      for (int run = 0; run < bufSizeKB.length; ++run) {
         bufSize = bufSizeKB[run] * 1024;
         try (FileInputStream in = new FileInputStream(inFileStr);
              FileOutputStream out = new FileOutputStream(outFileStr)) {
            startTime = System.nanoTime();
            byte[] byteBuf = new byte[bufSize];
            int numBytesRead;
            while ((numBytesRead = in.read(byteBuf)) != -1) {
               out.write(byteBuf, 0, numBytesRead);
            }
            elapsedTime = System.nanoTime() - startTime;
            System.out.printf("%4dKB: %6.2fmsec%n", bufSizeKB[run], (elapsedTime / 1000000.0));
            //System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

File size is 6777732 bytes

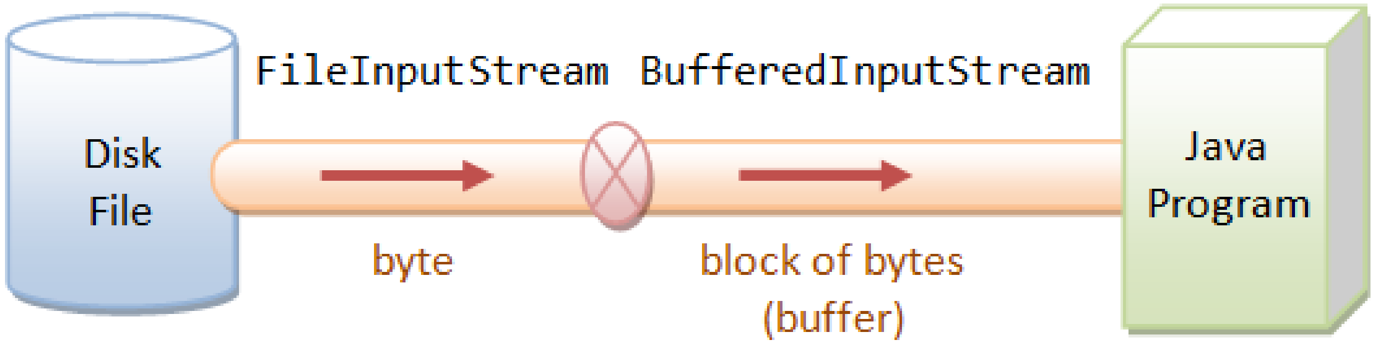
1KB:	**33.81msec**
2KB:	**24.36msec**
4KB:	**12.46msec**
8KB:	**8.62msec**
16KB:	**6.90msec**
32KB:	**5.26msec**
64KB:	**5.25msec**
256KB:	**8.63msec**
1024KB:	**16.95msec**

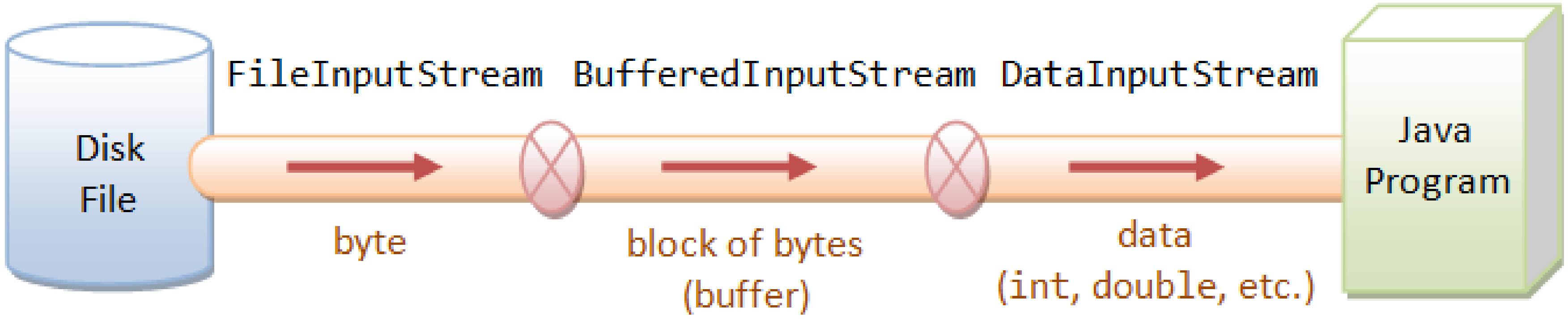
Buffered I/O Byte-Streams

- ***BufferedInputStream*** **and** ***BufferedOutputStream***
- ***Buffering*****, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O**

Layered (or Chained) I/O Streams

- **The I/O streams are often layered or chained with other I/O streams, for purposes such as buffering, filtering, or data-format conversion (between raw bytes and primitive types)**



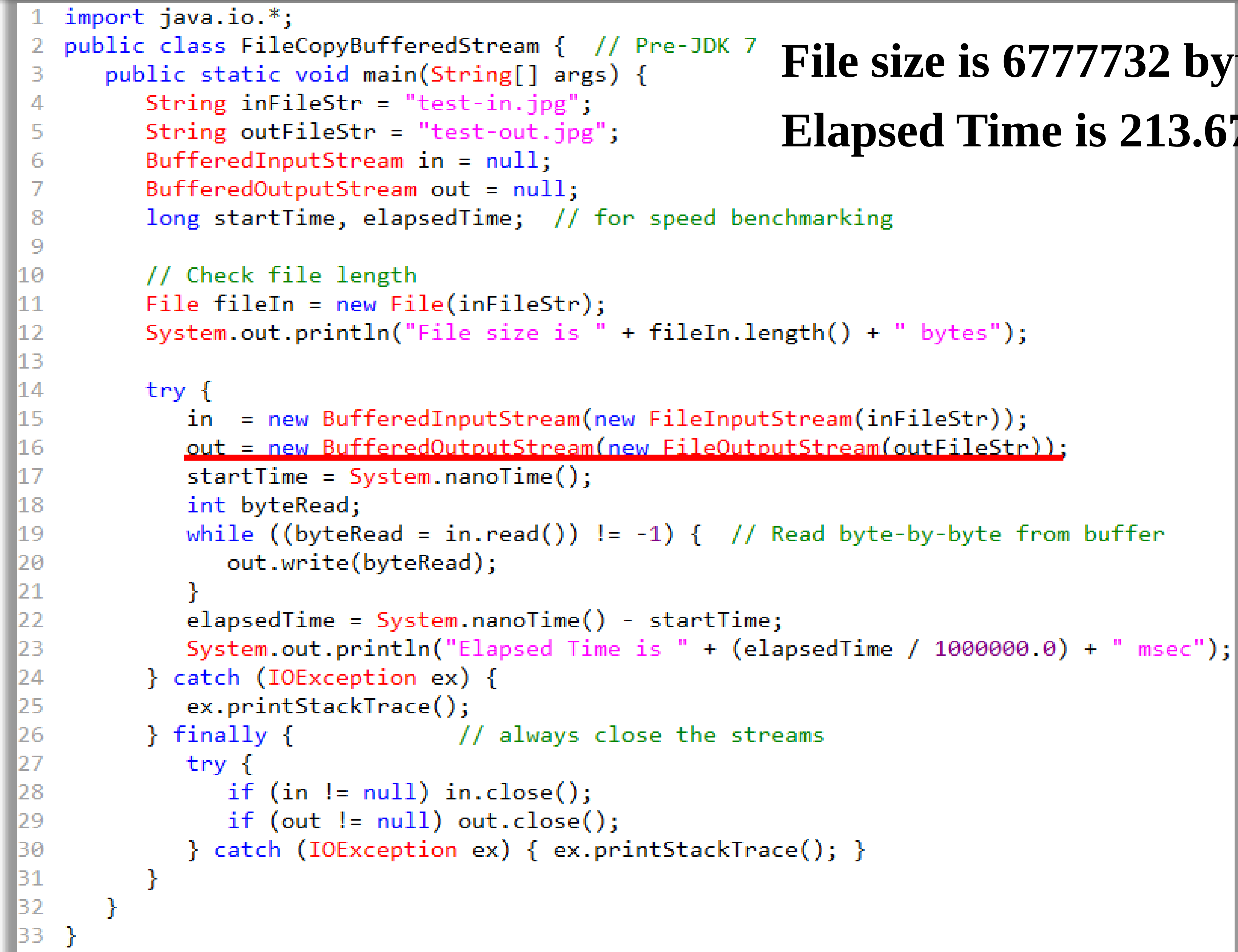


Layered (or Chained) I/O Streams

- The I/O streams are often layered or chained with other I/O streams, for purposes such as buffering, filtering, or data-format conversion (between raw bytes and primitive types)

```
FileInputStream fileIn = new FileInputStream("in.dat");
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);
DataInputStream dataIn = new DataInputStream(bufferIn);
// or
DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                           new FileInputStream("in.dat")));
```

Copying a file with Buffered Streams



```
import java.io.*;
public class FileCopyBufferedStream {  // Pre-JDK 7
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try {
         in  = new BufferedInputStream(new FileInputStream(inFileStr));
         out = new BufferedOutputStream(new FileOutputStream(outFileStr));
         startTime = System.nanoTime();
         int byteRead;
         while ((byteRead = in.read()) != -1) {  // Read byte-by-byte from buffer
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {            // always close the streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) { ex.printStackTrace(); }
      }
   }
}
```

File size is 6777732 bytes

Elapsed Time is 213.673102 msec

Formatted Data-Streams

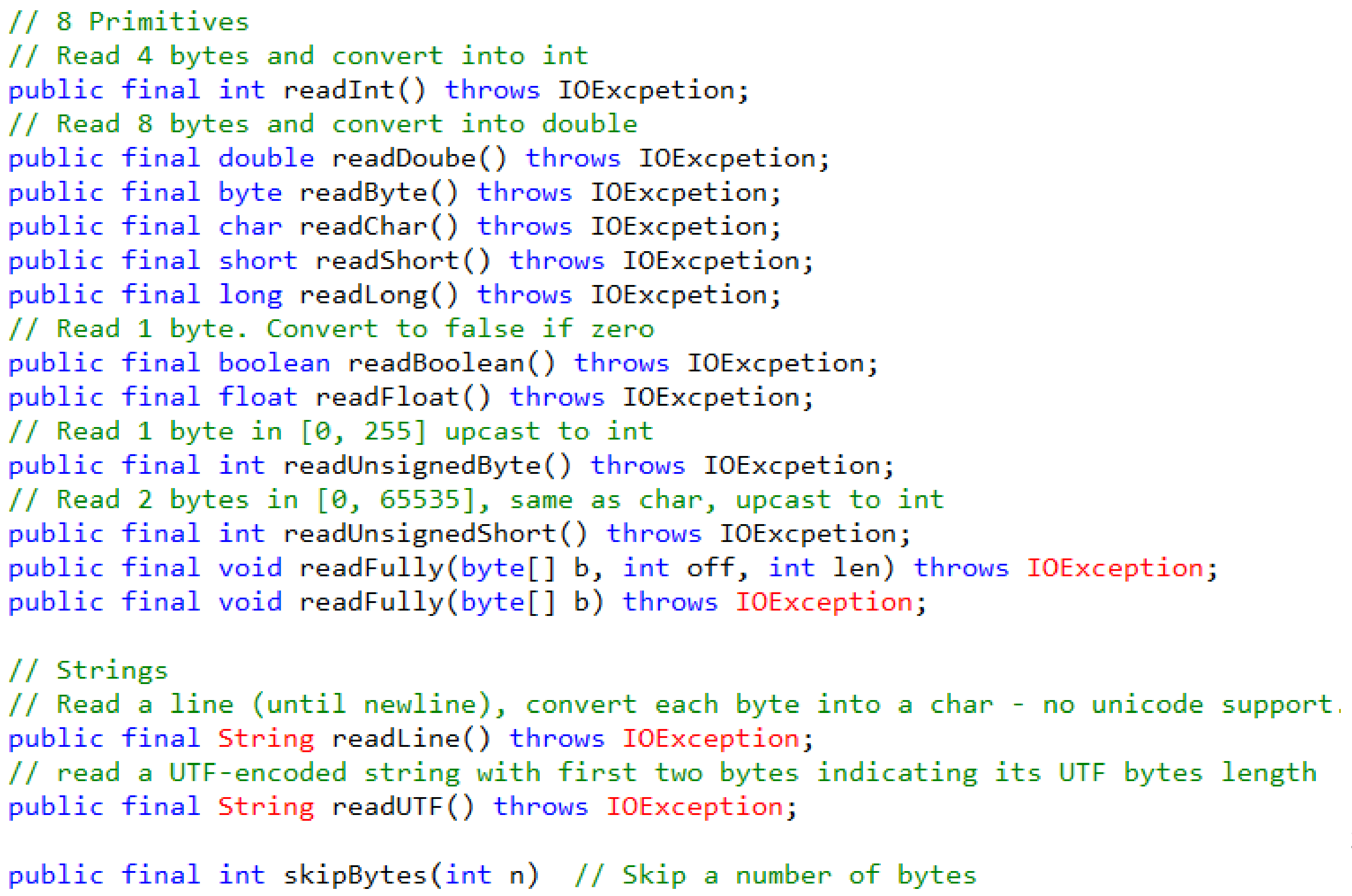
- The *DataInputStream* and *DataOutputStream* can be stacked on top of any *InputStream* and *OutputStream* to parse the raw bytes so as to perform I/O operations in the desired data format, such as *int* and *double*

- To use *DataInputStream* for formatted input, you can chain up the input streams as follows:

```
DataInputStream in = new DataInputStream(
                                new BufferedInputStream(
                                    new FileInputStream("in.dat")));
```

Formatted Data-Streams

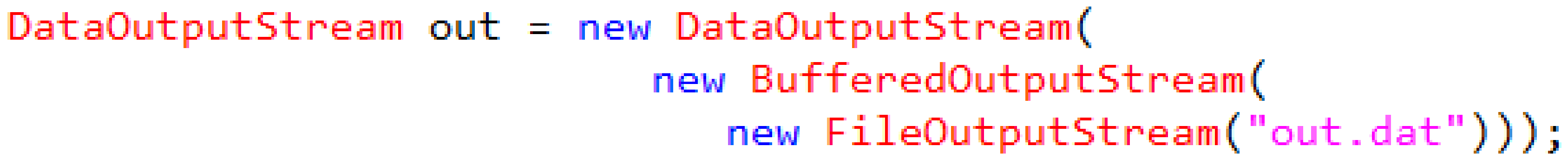
- ***DataInputStream*** **implements** ***DataInput*** **interface, which provides methods to read formatted primitive data and String, such as:**



```
// 8 Primitives
// Read 4 bytes and convert into int
public final int readInt() throws IOExcpetion;
// Read 8 bytes and convert into double
public final double readDoube() throws IOExcpetion;
public final byte readByte() throws IOExcpetion;
public final char readChar() throws IOExcpetion;
public final short readShort() throws IOExcpetion;
public final long readLong() throws IOExcpetion;
// Read 1 byte. Convert to false if zero
public final boolean readBoolean() throws IOExcpetion;
public final float readFloat() throws IOExcpetion;
// Read 1 byte in [0, 255] upcast to int
public final int readUnsignedByte() throws IOExcpetion;
// Read 2 bytes in [0, 65535], same as char, upcast to int
public final int readUnsignedShort() throws IOExcpetion;
public final void readFully(byte[] b, int off, int len) throws IOException;
public final void readFully(byte[] b) throws IOException;

// Strings
// Read a line (until newline), convert each byte into a char - no unicode support
public final String readLine() throws IOException;
// read a UTF-encoded string with first two bytes indicating its UTF bytes length
public final String readUTF() throws IOException;

public final int skipBytes(int n)  // Skip a number of bytes
```

Formatted Data-Streams

- **Similarly, you can stack the *DataOutputStream* as follows:**



```
DataOutputStream out = new DataOutputStream(
                                new BufferedOutputStream(
                                    new FileOutputStream("out.dat")));
```

Formatted Data-Streams

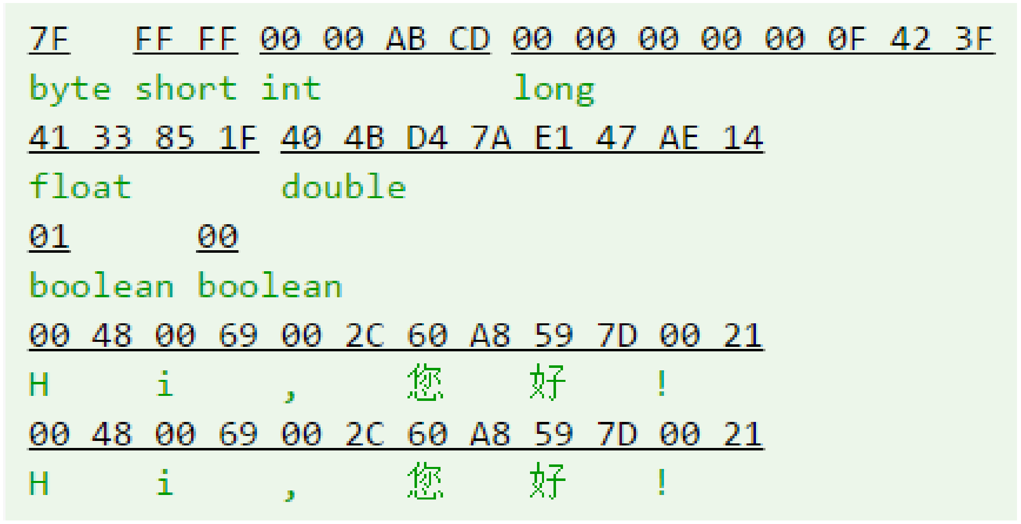
- ***DataOutputStream*** **implements** ***DataOutput*** **interface, which provides methods to write formatted primitive data and String. For examples,**

```
// 8 primitive types
// Write the int as 4 bytes
public final void writeInt(int i) throws IOExcpetion;
public final void writeFloat(float f) throws IOExcpetion;
// Write the double as 8 bytes
public final void writeDoube(double d) throws IOExcpetion;
// least-significant byte
public final void writeByte(int b) throws IOExcpetion;
// two lower bytes
public final void writeShort(int s) throws IOExcpetion;
public final void writeLong(long l) throws IOExcpetion;
public final void writeBoolean(boolean b) throws IOExcpetion;
public final void writeChar(int i) throws IOExcpetion;

// String
// least-significant byte of each char
public final void writeBytes(String str) throws IOExcpetion;
// Write String as UCS-2 16-bit char, Big-endian (big byte first)
public final void writeChars(String str) throws IOExcpetion;
// Write String as UTF, with first two bytes indicating UTF bytes length
public final void writeUTF(String str) throws IOException;

public final void write(byte[] b, int off, int len) throws IOException
public final void write(byte[] b) throws IOException
// Write the least-significant byte
public final void write(int b) throws IOException
```

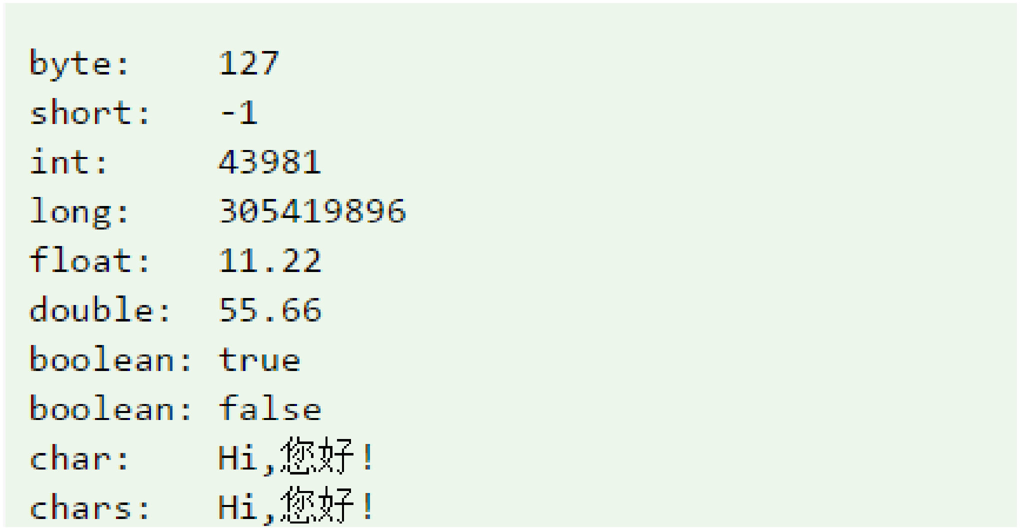
Formatted Data-Streams

```java
import java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-out.dat";
      String message = "Hi,您好!";
      // Write primitives to an output file
      try (DataOutputStream out =
              new DataOutputStream(
                 new BufferedOutputStream(
                    new FileOutputStream(filename)))) {
         out.writeByte(127);
         out.writeShort(0xFFFE);  // -1
         out.writeInt(0xABCD);
         out.writeLong(0xF423F);  // JDK 7 syntax
         out.writeFloat(11.22f);
         out.writeDouble(55.66);
         out.writeBoolean(true);
         out.writeBoolean(false);
         for (int i = 0; i < message.length(); ++i) {
            out.writeChar(message.charAt(i));
         }
         out.writeChars(message);
         out.writeBytes(message);
         out.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
      // Read raw bytes and print in Hex
      try (BufferedInputStream in =
              new BufferedInputStream(
                 new FileInputStream(filename))) {
         int inByte;
         while ((inByte = in.read()) != -1) {
            System.out.printf("%02X ", inByte);
         }     // Print Hex codes
         System.out.printf("%n%n");
      } catch (IOException ex) {
         ex.printStackTrace();
      }
      // Read primitives
      try (DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
         System.out.println("byte:    " + in.readByte());
         System.out.println("short:   " + in.readShort());
         System.out.println("int:     " + in.readInt());
         System.out.println("long:    " + in.readLong());
         System.out.println("float:   " + in.readFloat());
         System.out.println("double:  " + in.readDouble());
         System.out.println("boolean: " + in.readBoolean());
         System.out.println("boolean: " + in.readBoolean());

         System.out.print("char:    ");
         for (int i = 0; i < message.length(); ++i) {
            System.out.print(in.readChar());
         }
         System.out.println();

         System.out.print("chars:   ");
         for (int i = 0; i < message.length(); ++i) {
            System.out.print(in.readChar());
         }
         System.out.println();

         System.out.print("bytes:   ");
         for (int i = 0; i < message.length(); ++i) {
            System.out.print((char)in.readByte());
         }
         System.out.println();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

Formatted Data-Streams



```
7F   FF FF 00 00 AB CD 00 00 00 00 00 0F 42 3F
byte short int         long
41 33 85 1F 40 4B D4 7A E1 47 AE 14
float       double
01      00
boolean boolean
00 48 00 69 00 2C 60 A8 59 7D 00 21
H     i     ,     您    好    !
00 48 00 69 00 2C 60 A8 59 7D 00 21
H     i     ,     您    好    !
```

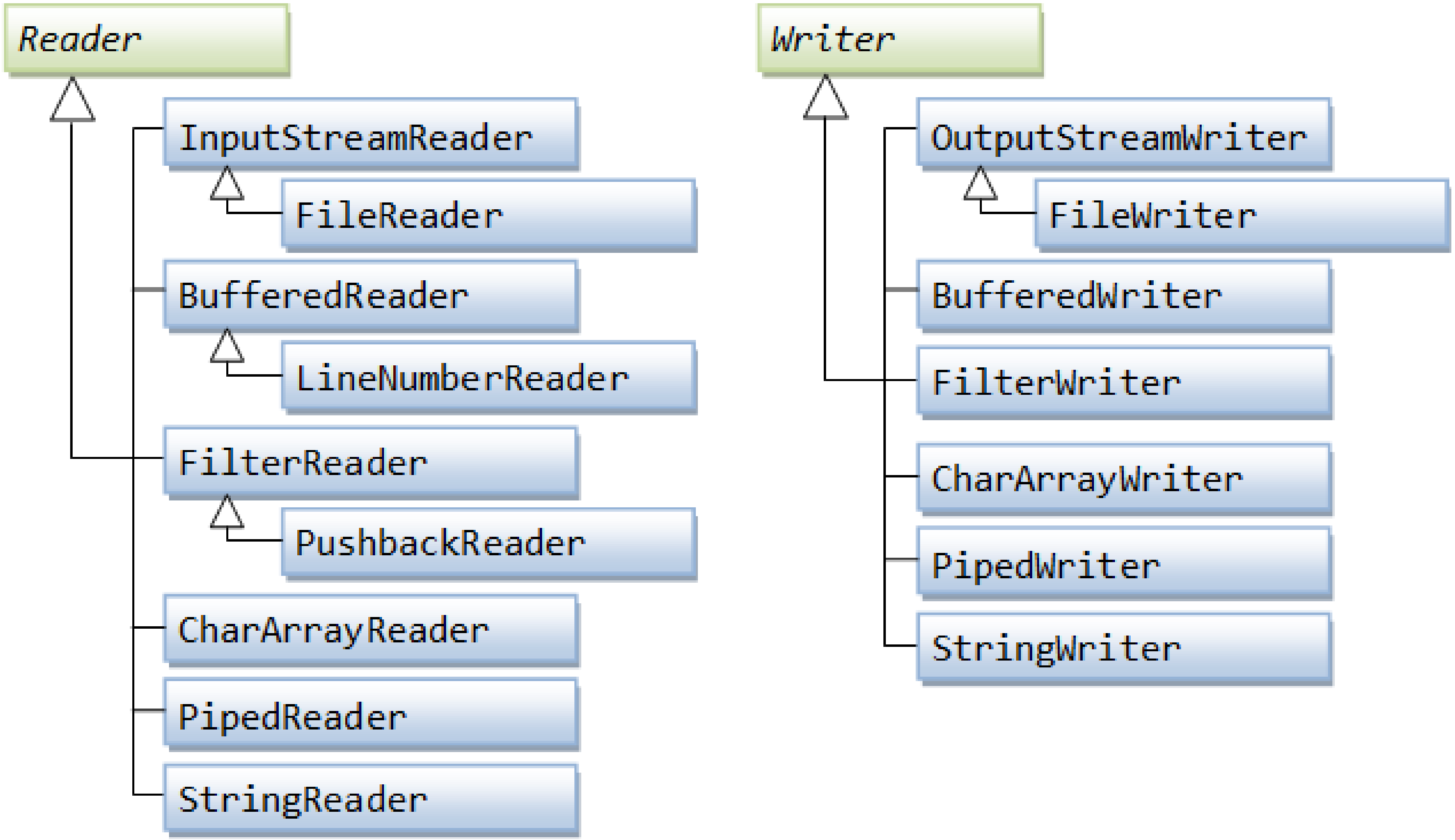


```
byte:     127
short:    -1
int:      43981
long:     305419896
float:    11.22
double:   55.66
boolean:  true
boolean:  false
char:     Hi,您好!
chars:    Hi,您好!
```

Character-Based I/O

- 16 bits, text-based
- Two parent *abstract* classes for characters:
  - ***reader***
  - ***Writer***

Character-Based I/O



Character-Based I/O

- **Java internally stores characters (char type) in 16-bit *UCS-2* character set**

- **The external data source/sink could store characters in other character set**
  - **E.g., *US-ASCII*, *ISO-8859-x*, *UTF-8*, *UTF-16*, and many others, in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes**

- **Java has to differentiate between byte-based I/O for processing 8-bit raw bytes, and character-based I/O for processing texts**

Character-Based I/O

- **The character streams needs to translate between the character set used by external I/O devices and Java internal *UCS-2* format**
  - **For example, the character '您' is stored as**
  - **"60 A8" in *UCS-2* (Java internal)**
  - **"E6 82 A8" in *UTF8***
  - **"C4 FA" in *GBK/GB2312***
  - **"B1 7A" in *BIG5***
  - **If this character is to be written to a file uses *UTF-8*, the character stream needs to translate "60 A8" to "E6 82 A8"**
  - **The reserve takes place in a reading operation**

Character-Based I/O

- **The byte/character streams refer to the unit of operation within the Java programs, which does not necessary correspond to the amount of data transferred from/to the external I/O devices**
  - **This is because some charsets use fixed-length of 8-bit**
    - ***US-ASCII***
    - ***ISO-8859-1***
  - **Variable-length of 1-4 bytes**
    - ***UTF-8***
    - ***UTF-16***
    - ***UTF-16-BE***
    - ***UTF-16-LE***
    - ***GBK***
    - ***BIG5***

Character-Based I/O

- Example: 您好 (Unicode: 60A8H 597DH)
  - The transformation between Unicode and UTF-8

Bits	Unicode	UTF-8 Code	Bytes
7	00000000 0xxxxxxx	0xxxxxxx	1 (ASCII)
11	00000yyy yyxxxxxx	110yyyyy 10xxxxxx	2
16	zzzzyyyy yyxxxxxx	1110zzzz 10yyyyyy 10xxxxxx	3
21	000uuuuu zzzzyyyy yyxxxxxx	11110uuu 10uuzzzz 10yyyyyy 10xxxxxx	4

```
Unicode (UCS-2) is 60A8H = 0110 0000 10 101000B
⇒ UTF-8 is 11100110 10000010 10101000B = E6 82 A8H
Unicode (UCS-2) is 597DH = 0101 1001 01 111101B
⇒ UTF-8 is 11100101 10100101 10111101B = E5 A5 BDH
```

- When a character stream is used to read an 8-bit *ASCII* file, an 8-bit data is read from the file and put into the 16-bit char location of the Java program

Abstract superclass Reader and Writer

- **Other than the unit of operation and charset conversion (which is extremely complex), character-based I/O is almost identical to byte-based I/O**

- **The abstract superclass *Reader* declares an abstract method *read()* to read one character from the input source**

```
public abstract int read() throws IOException
public int read(char[] chars, int offset, int length) throws IOException
public int read(char[] chars) throws IOException
```

- **The abstract superclass *Writer* declares an abstract method *write()* to write one character to the output sink**
  - **The lower 2 bytes of the int argument is written out; while the upper 2 bytes are discarded**

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars, int offset, int length) throws IOException
public void write(char[] chars) throws IOException
```

File I/O Character-Streams

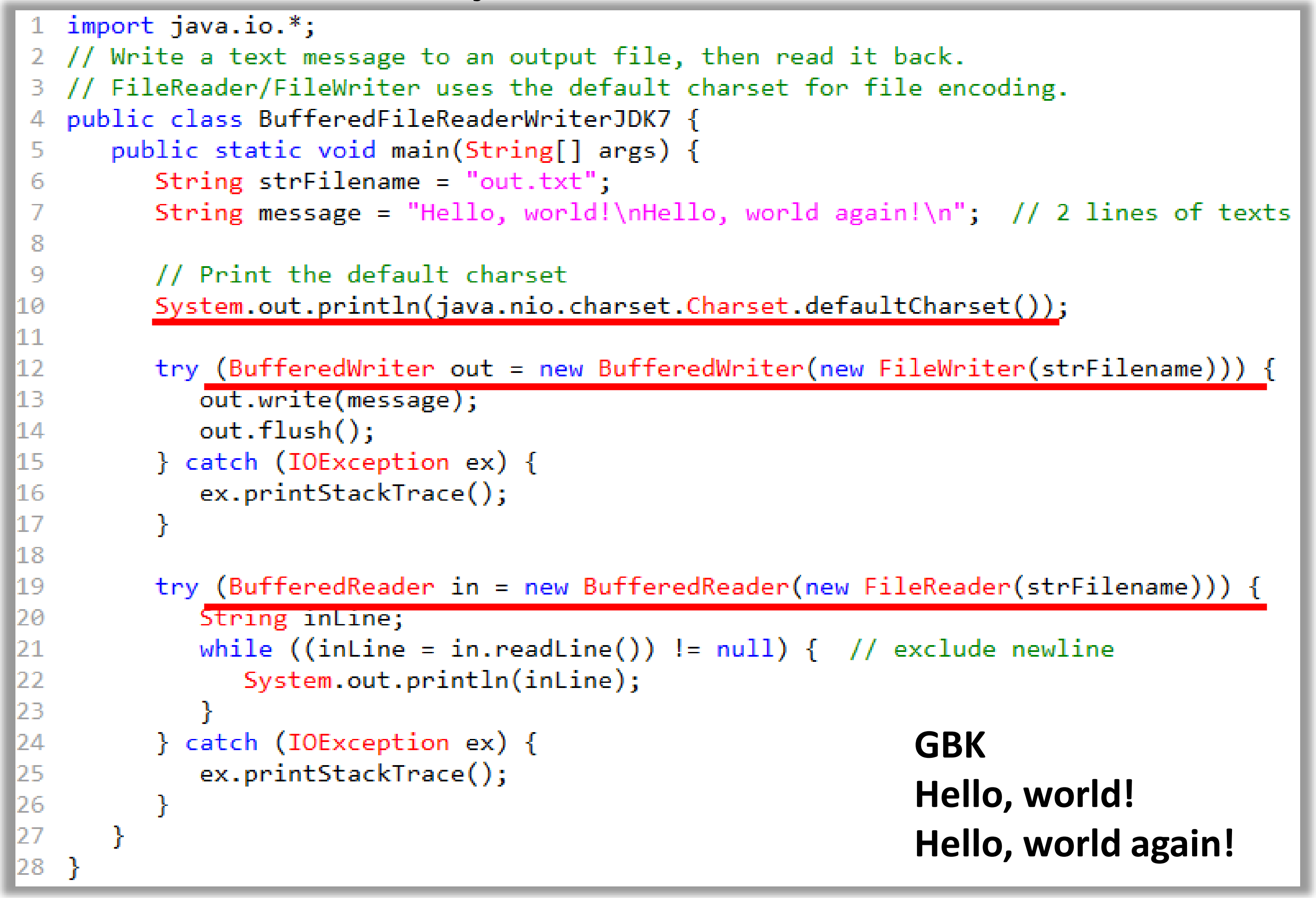
- ***FileReader*** **and** ***FileWriter*** **are concrete implementations to the abstract superclasses** ***Reader*** **and** ***Writer*****, to support I/O from disk files**
- ***FileReader*****/*****FileWriter*** **assumes that the default character encoding (charset) is used for the disk file**
  - **The default charset is kept in the JVM's system property "file.encoding“**
  - **We can get the default charset via static method**

    java.nio.charset.Charset.defaultCharset()

    or ***System.getProperty("file.encoding")***
- **Use of** ***FileReader/FileWriter*** **is NOT recommended as you have no control of the file encoding charset**

Buffered I/O Character-Streams

- ***BufferedReader*** **and** ***BufferedWriter*** **can be stacked on top of** ***FileReader/FileWriter*** **or other character streams to perform buffered I/O, instead of character-by-character**



```java
import java.io.*;
// Write a text message to an output file, then read it back.
// FileReader/FileWriter uses the default charset for file encoding.
public class BufferedFileReaderWriterJDK7 {
   public static void main(String[] args) {
      String strFilename = "out.txt";
      String message = "Hello, world!\nHello, world again!\n";  // 2 lines of texts

      // Print the default charset
      System.out.println(java.nio.charset.Charset.defaultCharset());

      try (BufferedWriter out = new BufferedWriter(new FileWriter(strFilename))) {
         out.write(message);
         out.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      try (BufferedReader in = new BufferedReader(new FileReader(strFilename))) {
         String inLine;
         while ((inLine = in.readLine()) != null) {  // exclude newline
            System.out.println(inLine);
         }
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

```
GBK
Hello, world!
Hello, world again!
```

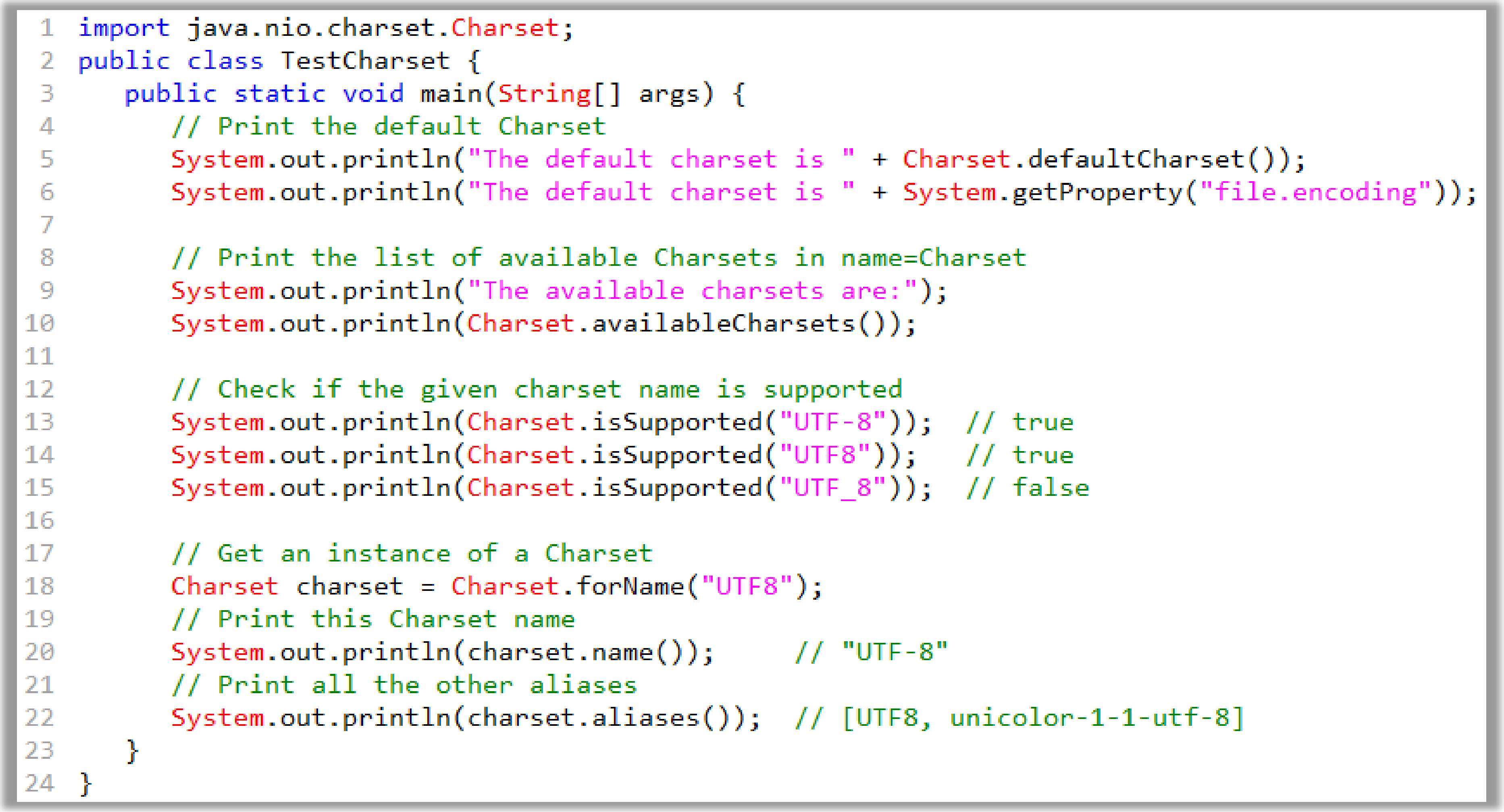
Character Set (or Charset)

- JDK 1.4 provides a new package *java.nio.charset* as part of NIO (New IO) to support character translation between the Unicode (UCS-2) used internally in Java program and external devices
  - which could be encoded in any other format (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, UTF-16BE, UTF-16LE, and etc.)
- The main class *java.nio.charset.Charset* provides static methods for testing whether a particular charset is supported, locating charset instances by name, and listing all the available charsets and the default charset

```
// lists all the available charsets
public static SortedMap<String,Charset> availableCharsets()
// Returns the default charset
public static Charset defaultCharset()
// Returns a Charset instance for the given charset name (in String)
public static Charset forName(String charsetName)
// Tests if this charset name is supported
public static boolean isSupported(String charsetName)
```

Character Set (or Charset)

❑ Example



```java
import java.nio.charset.Charset;
public class TestCharset {
    public static void main(String[] args) {
        // Print the default Charset
        System.out.println("The default charset is " + Charset.defaultCharset());
        System.out.println("The default charset is " + System.getProperty("file.encoding"));

        // Print the list of available Charsets in name=Charset
        System.out.println("The available charsets are:");
        System.out.println(Charset.availableCharsets());

        // Check if the given charset name is supported
        System.out.println(Charset.isSupported("UTF-8"));  // true
        System.out.println(Charset.isSupported("UTF8"));   // true
        System.out.println(Charset.isSupported("UTF_8"));  // false

        // Get an instance of a Charset
        Charset charset = Charset.forName("UTF8");
        // Print this Charset name
        System.out.println(charset.name());     // "UTF-8"
        // Print all the other aliases
        System.out.println(charset.aliases());  // [UTF8, unicolor-1-1-utf-8]
    }
}
```

Character Set (or Charset)

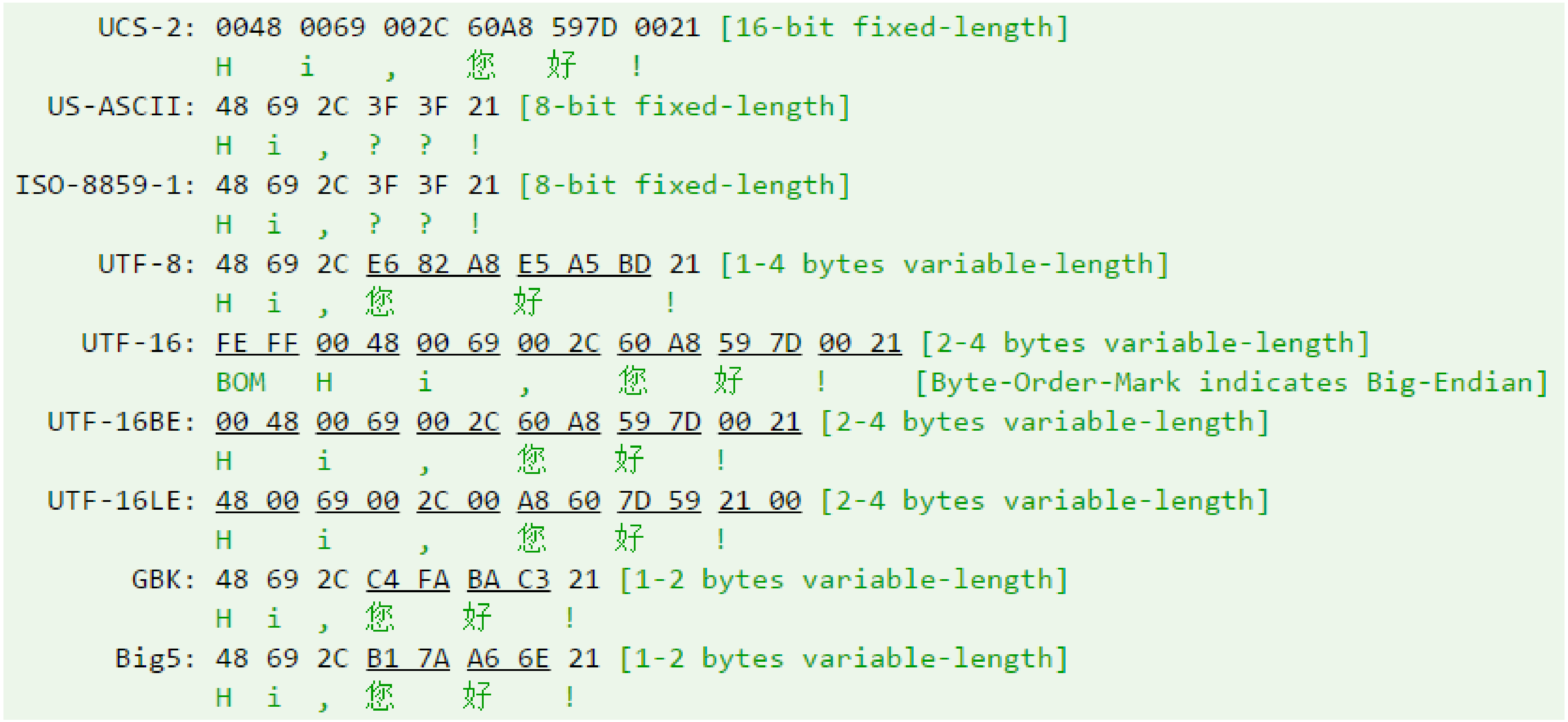
- The ***Charset*** class provides methods to ***encode/decode*** characters from *UCS-2* used in Java program and the specific charset used in the external devices (such as *UTF-8*)
- The ***encode()/decode()*** methods (New I/O methods) operate on ***ByteBuffer*** and ***CharBuffer***

```
public final ByteBuffer encode(String s)
public final ByteBuffer encode(CharBuffer cb)
// Encodes Unicode UCS-2 characters in the CharBuffer/String
//  into a "byte sequence" using this charset, and returns a ByteBuffer.

public final CharBuffer decode(ByteBuffer bb)
// Decode the byte sequence encoded using this charset in the ByteBuffer
//  to Unicode UCS-2, and return a charBuffer.
```

Character Set (or Charset)

```java
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;

public class TestCharsetEncodeDecode {
   public static void main(String[] args) {
      // Try these charsets for encoding
      String[] charsetNames = {"US-ASCII", "ISO-8859-1", "UTF-8", "UTF-16",
                               "UTF-16BE", "UTF-16LE", "GBK", "BIG5"};

      String message = "Hi,您好!";  // Unicode message to be encoded
      // Print UCS-2 in hex codes
      System.out.printf("%10s: ", "UCS-2");
      for (int i = 0; i < message.length(); ++i) {
         System.out.printf("%04X ", (int)message.charAt(i));
      }
      System.out.println();

      for (String charsetName: charsetNames) {
         // Get a Charset instance given the charset name string
         Charset charset = Charset.forName(charsetName);
         System.out.printf("%10s: ", charset.name());

         // Encode the Unicode UCS-2 characters into a byte sequence in this charset.
         ByteBuffer bb = charset.encode(message);
         while (bb.hasRemaining()) {
            System.out.printf("%02X ", bb.get());  // Print hex code
         }
         System.out.println();
         bb.rewind();
      }
   }
}
```

java.io.PrintStream & java.io.PrintWriter



```
     UCS-2: 0048 0069 002C 60A8 597D 0021 [16-bit fixed-length]
            H    i    ,    您   好   !
  US-ASCII: 48 69 2C 3F 3F 21 [8-bit fixed-length]
            H  i  ,  ?  ?  !
ISO-8859-1: 48 69 2C 3F 3F 21 [8-bit fixed-length]
            H  i  ,  ?  ?  !
     UTF-8: 48 69 2C E6 82 A8 E5 A5 BD 21 [1-4 bytes variable-length]
            H  i  ,  您       好       !
    UTF-16: FE FF 00 48 00 69 00 2C 60 A8 59 7D 00 21 [2-4 bytes variable-length]
            BOM   H     i     ,     您    好    !     [Byte-Order-Mark indicates Big-Endian]
  UTF-16BE: 00 48 00 69 00 2C 60 A8 59 7D 00 21 [2-4 bytes variable-length]
            H     i     ,     您    好    !
  UTF-16LE: 48 00 69 00 2C 00 A8 60 7D 59 21 00 [2-4 bytes variable-length]
            H     i     ,     您    好    !
       GBK: 48 69 2C C4 FA BA C3 21 [1-2 bytes variable-length]
            H  i  ,  您    好    !
      Big5: 48 69 2C B1 7A A6 6E 21 [1-2 bytes variable-length]
            H  i  ,  您    好    !
```

Text File I/O

- **Java internally stores characters (char type) in 16-bit *UCS-2* character set**
- **But the external data source/sink could store characters in other character set (e.g., *US-ASCII*, *ISO-8859-x*, *UTF-8*, *UTF-16*, and many others), in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes**
- **The *FileReader/FileWriter* introduced earlier uses the default charset for decoding/encoding, resulted in non-portable programs**

Text File I/O

- **To choose the charset, we use *InputStreamReader* and *OutputStreamWriter***
  - ***InputStreamReader* and *OutputStreamWriter* are considered to be byte-to-character "bridge" streams**
- **We choose the character set in the *InputStreamReader*'s constructor:**

```
public InputStreamReader(InputStream in)   // Use default charset
public InputStreamReader(InputStream in, String charsetName) throws UnsupportedEncodingException
public InputStreamReader(InputStream in, Charset cs)
```

- **We choose the character set in the *OutputStreamWriter*'s constructor:**

```
public OutputStreamWriter(OutputStream out)
public OutputStreamWriter(OutputStream out, String charsetName) throws UnsupportedEncodingException
public OutputStreamWriter(OutputStream out, Charset cs)
```

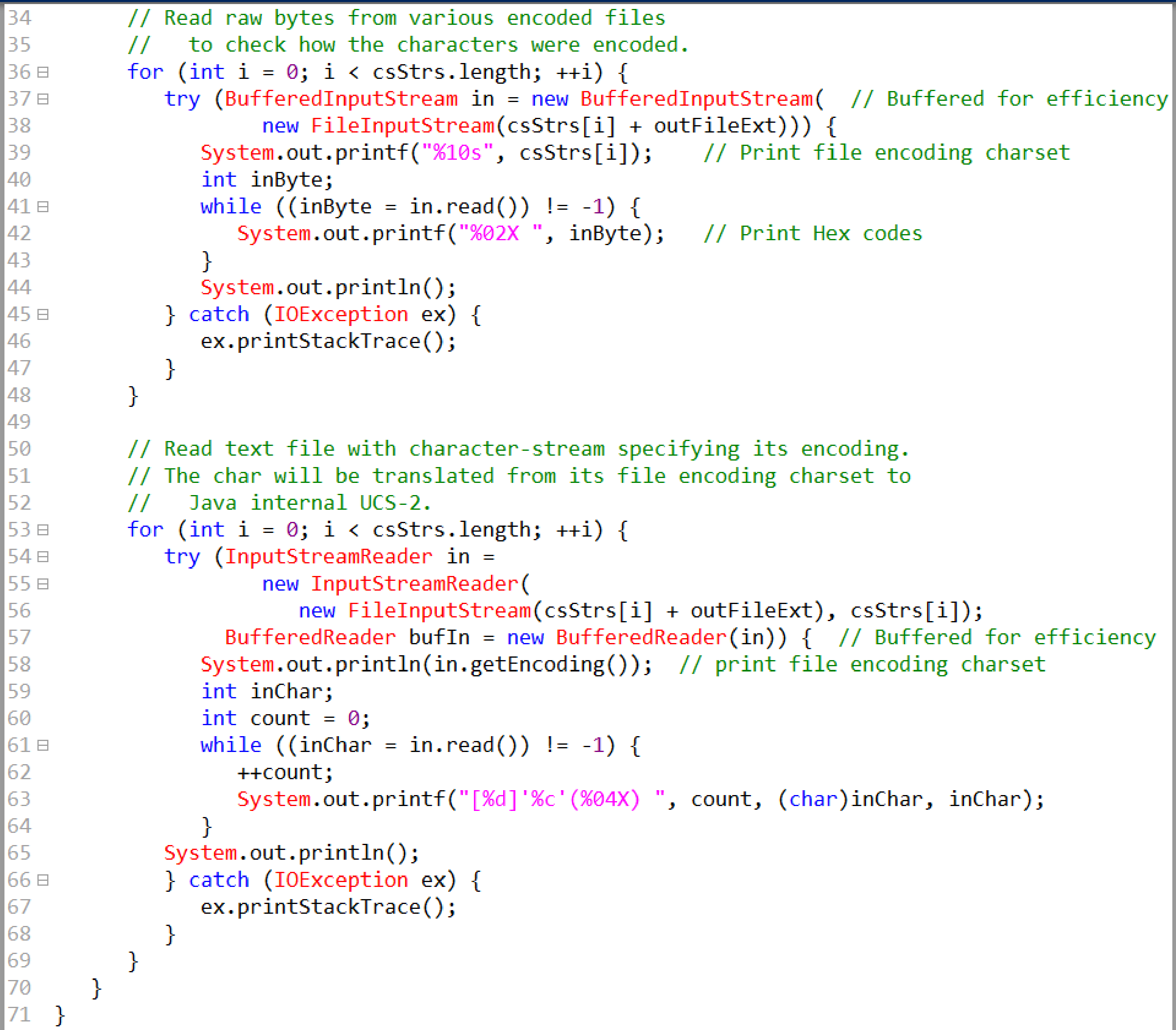
Text File I/O

- **You can list the available charsets via static method *java.nio.charset.Charset.availableCharsets()***
- **The commonly-used Charset names supported by Java are:**
  - **"US-ASCII": 7-bit ASCII (aka ISO646-US)**
  - **"ISO-8859-1": Latin-1**
  - **"UTF-8": Most commonly-used encoding scheme for Unicode**
  - **"UTF-16BE": Big-endian (big byte first) (big-endian is usually the default)**
  - **"UTF-16LE": Little-endian (little byte first)**
  - **"UTF-16": with a 2-byte BOM (Byte-Order-Mark) to specify the byte order. FE FF indicates big-endian, FF FE indicates little-endian.**
- **As the *InputStreamReader/OutputStreamWriter* often needs to read/write in multiple bytes, it is best to wrap it with a *BufferedReader/BufferedWriter***

Text File I/O

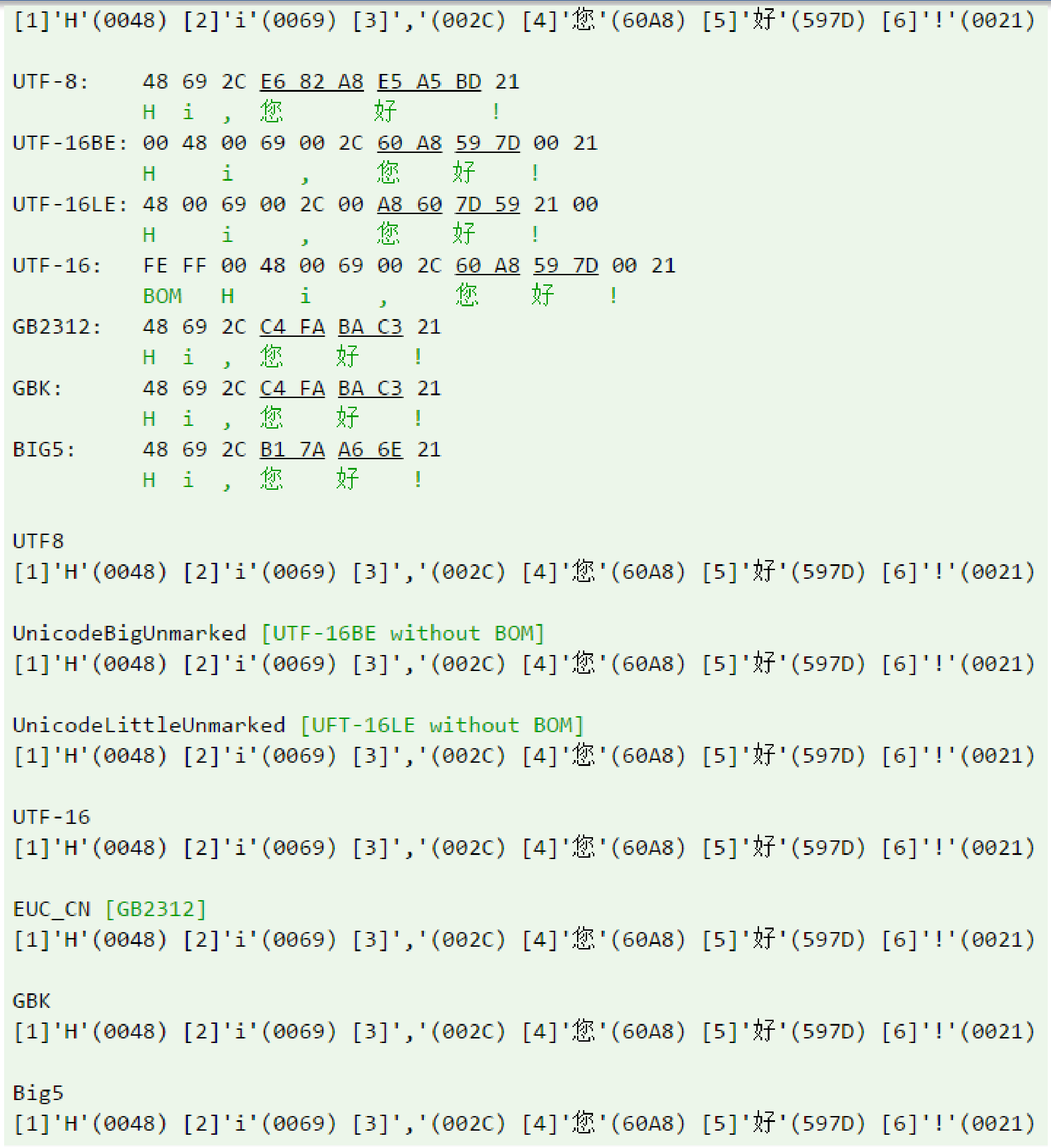
```java
import java.io.*;
// Write texts to file using OutputStreamWriter specifying its charset encoding.
// Read byte-by-byte using FileInputStream.
// Read char-by-char using InputStreamReader specifying its charset encoding.
public class TextFileEncodingJDK7 {
   public static void main(String[] args) {
      String message = "Hi,您好!";   // with non-ASCII chars
      // Java internally stores char in UCS-2/UTF-16
      // Print the characters stored with Hex codes
      for (int i = 0; i < message.length(); ++i) {
         char aChar = message.charAt(i);
         System.out.printf("[%d]'%c'(%04X) ", (i+1), aChar, (int)aChar);
      }
      System.out.println();

      // Try these charsets for encoding text file
      String[] csStrs = {"UTF-8", "UTF-16BE", "UTF-16LE", "UTF-16", "GB2312", "GBK", "BIG5"};
      String outFileExt = "-out.txt";   // Output filenames are "charset-out.txt"

      // Write text file in the specified file encoding charset
      for (int i = 0; i < csStrs.length; ++i) {
         try (OutputStreamWriter out =
                 new OutputStreamWriter(
                    new FileOutputStream(csStrs[i] + outFileExt), csStrs[i]);
              BufferedWriter bufOut = new BufferedWriter(out)) {  // Buffered for efficiency
            System.out.println(out.getEncoding());  // Print file encoding charset
            bufOut.write(message);
            bufOut.flush();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
```

Text File I/O



```java
      // Read raw bytes from various encoded files
      //   to check how the characters were encoded.
      for (int i = 0; i < csStrs.length; ++i) {
         try (BufferedInputStream in = new BufferedInputStream(  // Buffered for efficiency
                 new FileInputStream(csStrs[i] + outFileExt))) {
            System.out.printf("%10s", csStrs[i]);    // Print file encoding charset
            int inByte;
            while ((inByte = in.read()) != -1) {
               System.out.printf("%02X ", inByte);   // Print Hex codes
            }
            System.out.println();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }

      // Read text file with character-stream specifying its encoding.
      // The char will be translated from its file encoding charset to
      //   Java internal UCS-2.
      for (int i = 0; i < csStrs.length; ++i) {
         try (InputStreamReader in =
                 new InputStreamReader(
                    new FileInputStream(csStrs[i] + outFileExt), csStrs[i]);
              BufferedReader bufIn = new BufferedReader(in)) {  // Buffered for efficiency
            System.out.println(in.getEncoding());  // print file encoding charset
            int inChar;
            int count = 0;
            while ((inChar = in.read()) != -1) {
               ++count;
               System.out.printf("[%d]'%c'(%04X) ", count, (char)inChar, inChar);
            }
         System.out.println();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

Text File I/O



```
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UTF-8:    48 69 2C E6 82 A8 E5 A5 BD 21
          H  i  ,  您        好        !
UTF-16BE: 00 48 00 69 00 2C 60 A8 59 7D 00 21
          H     i     ,     您     好     !
UTF-16LE: 48 00 69 00 2C 00 A8 60 7D 59 21 00
          H     i     ,     您     好     !
UTF-16:   FE FF 00 48 00 69 00 2C 60 A8 59 7D 00 21
          BOM   H     i     ,     您     好     !
GB2312:   48 69 2C C4 FA BA C3 21
          H  i  ,  您     好     !
GBK:      48 69 2C C4 FA BA C3 21
          H  i  ,  您     好     !
BIG5:     48 69 2C B1 7A A6 6E 21
          H  i  ,  您     好     !

UTF8
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UnicodeBigUnmarked [UTF-16BE without BOM]
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UnicodeLittleUnmarked [UFT-16LE without BOM]
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UTF-16
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

EUC_CN [GB2312]
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

GBK
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

Big5
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)
```

Object Serialization and Object Streams

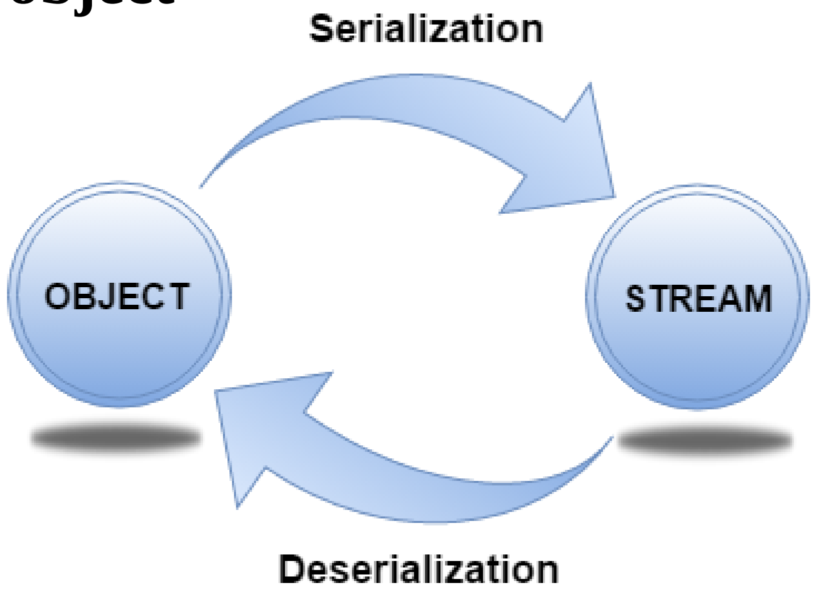
- Data streams (*DataInputStream* & *DataOutputStream*) allow you to read and write primitive data (such as int, double) and String, rather than individual bytes

- Object streams (*ObjectInputStream* & *ObjectOutputStream*) go one step further to allow you to read and write entire objects (such as Date, ArrayList or any custom objects)

Object Serialization and Object Streams

- Object serialization is the process of representing a "particular state of an object" in a serialized bit-stream, so that the bit stream can be written out to an external device (such as a disk file or network)

- The bit-stream can later be re-constructed to recover the state of that object



Object Serialization and Object Streams

- **Object serialization is necessary to save a state of an object into a disk file for persistence or sent the object across the network for applications such as Web Services, Distributed-object applications, and Remote Method Invocation (RMI)**

Object Serialization and Object Streams

- **In Java, object that requires to be serialized must implement *java.io.Serializable* or *java.io.Externalizable* interface.**
- ***Serializable* interface is an empty interface with nothing declared. Its purpose is simply to declare that particular object is serializable**

ObjectInputStream & ObjectOutputStream

- The ***ObjectInputStream*** **and** ***ObjectOutputStream*** **can be used to serialize an object into a bit-stream and transfer it to/from an I/O streams, via these methods:**

```
public final Object readObject() throws IOException, ClassNotFoundException;
public final void writeObject(Object obj) throws IOException;
```

- ***ObjectInputStream*** **and** ***ObjectOutputStream*** **must be stacked on top of a concrete implementation of** ***InputStream*** **or** ***OutputStream*****, such as** ***FileInputStream*** **or** ***FileOutputStream***

ObjectInputStream & ObjectOutputStream

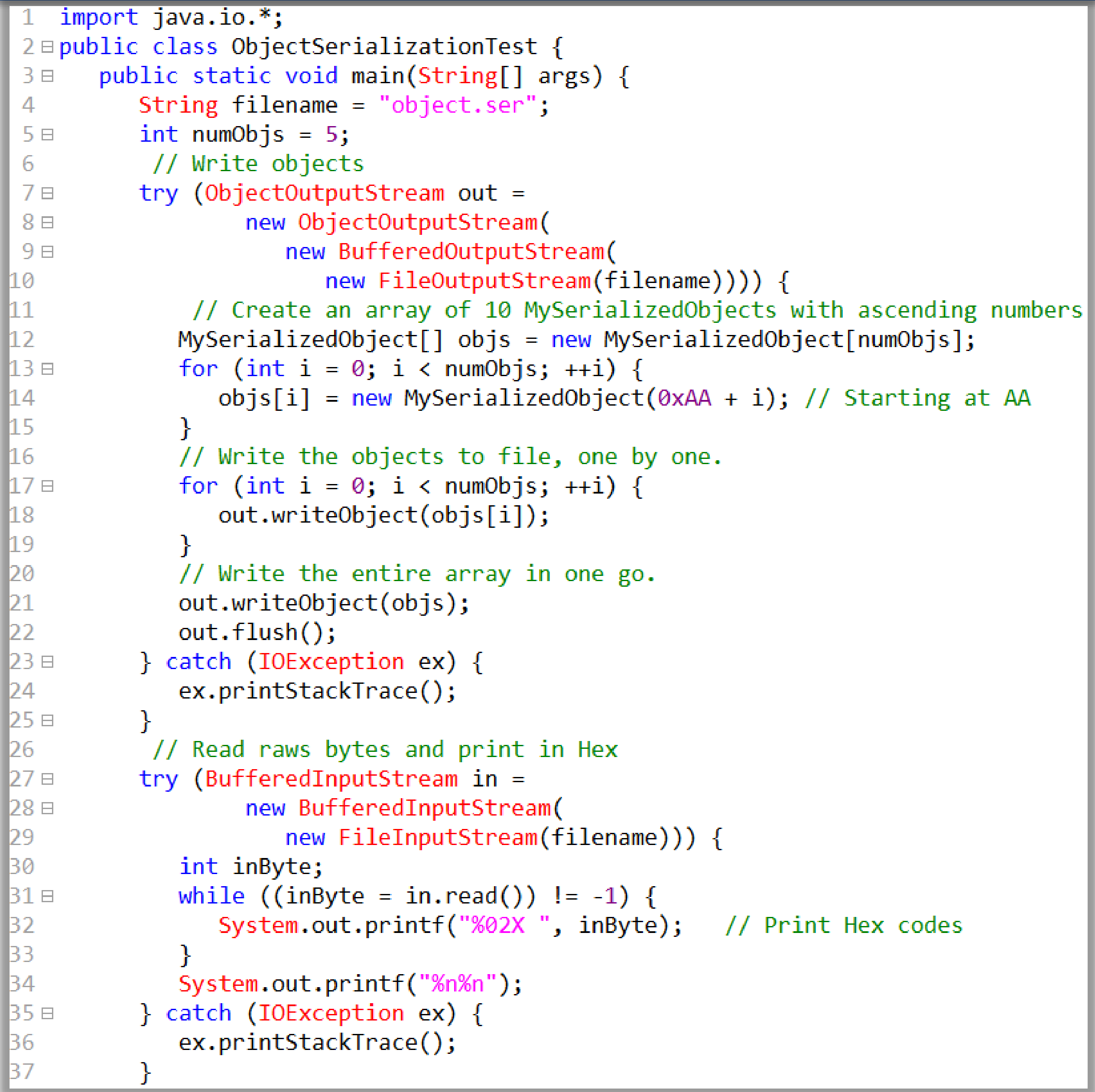
- writes objects to a disk file

```
ObjectOutputStream out =
   new ObjectOutputStream(
      new BufferedOutputStream(
         new FileOutputStream("object.ser")));
out.writeObject("The current Date and Time is "); // write a String object
out.writeObject(new Date());                      // write a Date object
out.flush();
out.close();
```

- To read and re-construct the object back in a program, use the method readObject() that returns an java.lang.Object
  - Downcast the Object back to its original type

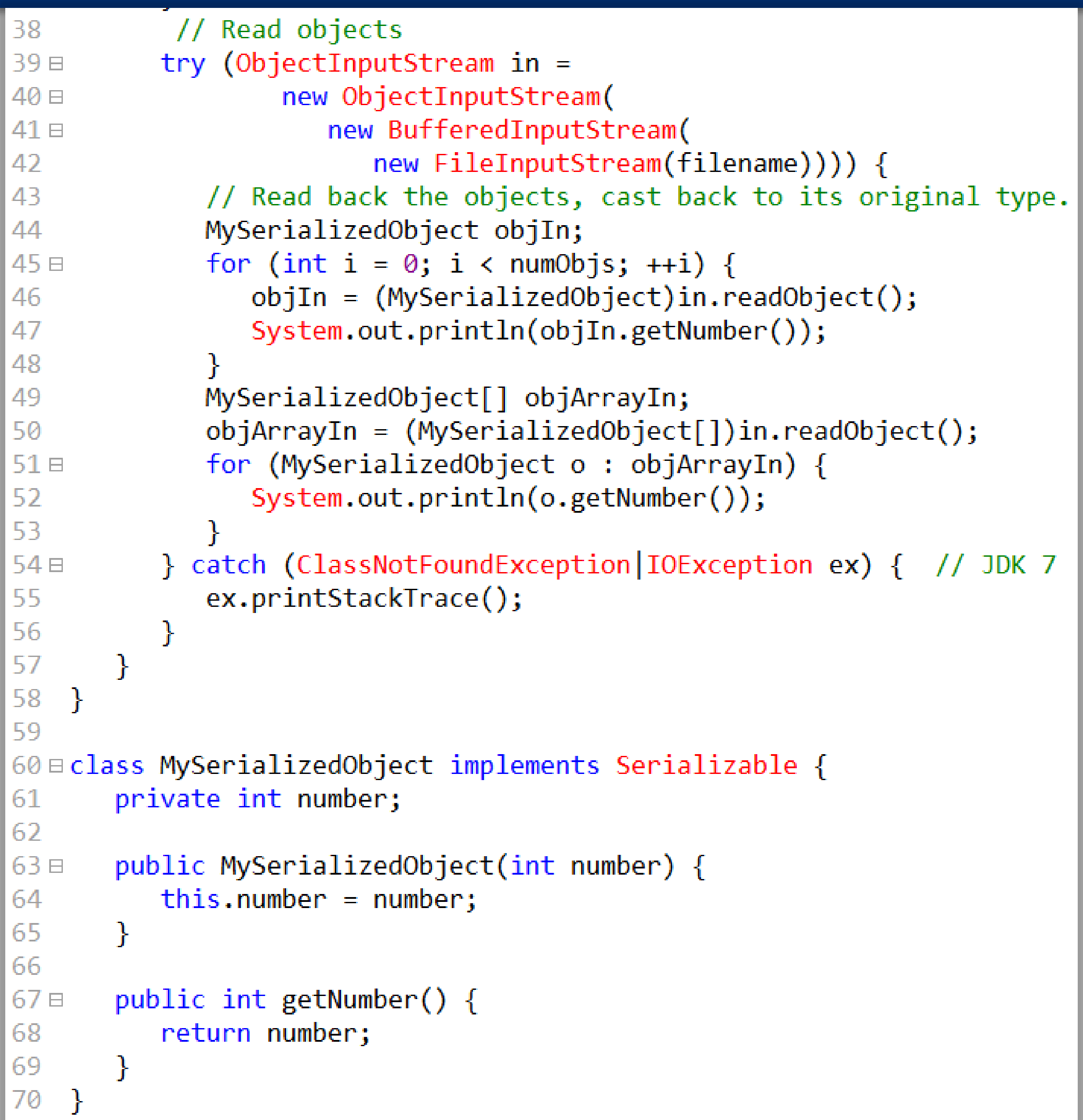
```
ObjectInputStream in =
   new ObjectInputStream(
      new BufferedInputStream(
         new FileInputStream("object.ser")));
String str = (String)in.readObject();
Date d = (Date)in.readObject(new Date());  // downcast
in.close();
```

ObjectInputStream & ObjectOutputStream



```java
import java.io.*;
public class ObjectSerializationTest {
   public static void main(String[] args) {
      String filename = "object.ser";
      int numObjs = 5;
       // Write objects
      try (ObjectOutputStream out =
              new ObjectOutputStream(
                 new BufferedOutputStream(
                    new FileOutputStream(filename)))) {
          // Create an array of 10 MySerializedObjects with ascending numbers
         MySerializedObject[] objs = new MySerializedObject[numObjs];
         for (int i = 0; i < numObjs; ++i) {
            objs[i] = new MySerializedObject(0xAA + i); // Starting at AA
         }
         // Write the objects to file, one by one.
         for (int i = 0; i < numObjs; ++i) {
            out.writeObject(objs[i]);
         }
         // Write the entire array in one go.
         out.writeObject(objs);
         out.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
       // Read raws bytes and print in Hex
      try (BufferedInputStream in =
              new BufferedInputStream(
                 new FileInputStream(filename))) {
         int inByte;
         while ((inByte = in.read()) != -1) {
            System.out.printf("%02X ", inByte);   // Print Hex codes
         }
         System.out.printf("%n%n");
      } catch (IOException ex) {
         ex.printStackTrace();
      }
```

ObjectInputStream & ObjectOutputStream



```
       // Read objects
      try (ObjectInputStream in =
              new ObjectInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
         // Read back the objects, cast back to its original type.
         MySerializedObject objIn;
         for (int i = 0; i < numObjs; ++i) {
            objIn = (MySerializedObject)in.readObject();
            System.out.println(objIn.getNumber());
         }
         MySerializedObject[] objArrayIn;
         objArrayIn = (MySerializedObject[])in.readObject();
         for (MySerializedObject o : objArrayIn) {
            System.out.println(o.getNumber());
         }
      } catch (ClassNotFoundException|IOException ex) {  // JDK 7
         ex.printStackTrace();
      }
   }
}

class MySerializedObject implements Serializable {
   private int number;

   public MySerializedObject(int number) {
      this.number = number;
   }

   public int getNumber() {
      return number;
   }
}
```

ObjectInputStream & ObjectOutputStream

```
AC ED 00 05 73 72 00 12 4D 79 53 65 72 69 61 6C 69 7A 65 64 4F 62 6A 65
63 74 1F 7B 91 BD 02 1C DC 30 02 00 01 49 00 06 6E 75 6D 62 65 72 78 70
00 00 00 AA 73 71 00 7E 00 00 00 00 00 AB 73 71 00 7E 00 00
00 00 00 AC 73 71 00 7E 00 00 00 00 00 AD 73 71 00 7E 00 00
00 00 00 AE 75 72 00 15 5B 4C 4D 79 53 65 72 69 61 6C 69 7A 65 64 4F 62
6A 65 63 74 3B 13 95 A0 51 BC 86 75 38 02 00 00 78 70 00 00 00 05 71 00
7E 00 01 71 00 7E 00 02 71 00 7E 00 03 71 00 7E 00 04 71 00 7E 00 05
```

Formatted Text I/O

- JDK 1.5 introduces *java.util.Scanner* class, which greatly simplifies formatted text input from input source (e.g., files, keyboard, network)
- *Scanner* is a simple text scanner which can parse the input text into primitive types and strings using regular expressions.
- It first breaks the text input into tokens using a delimiter pattern, which is by default the white spaces (blank, tab and newline).
  - The tokens may then be converted into primitive values of different types using the various *nextXxx()* methods (*nextInt()*, *nextByte()*, *nextShort()*, *nextLong()*, *nextFloat()*, *nextDouble()*, *nextBoolean()*, *next()* for String, and *nextLine()* for an input line). You can also use the *hasNextXxx()* methods to check for the availability of a desired input.

Formatted Text I/O

- **The commonly-used constructors are as follows. You can construct a Scanner to parse a byte-based InputStream (e.g., System.in), a disk file, or a given String**

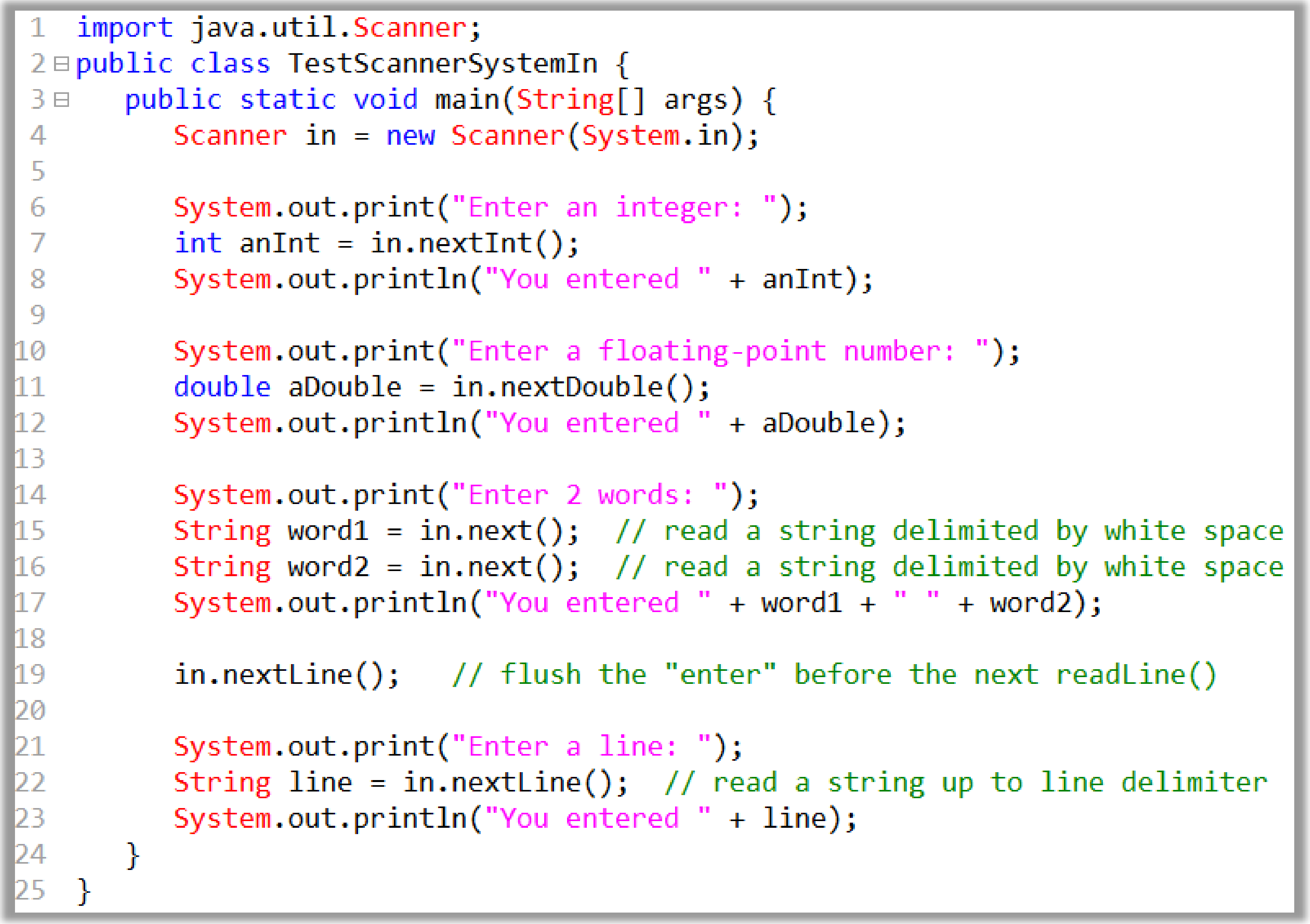
```
// Scanner piped from a disk File
public Scanner(File source) throws FileNotFoundException
public Scanner(File source, String charsetName) throws FileNotFoundException
// Scanner piped from a byte-based InputStream, e.g., System.in
public Scanner(InputStream source)
public Scanner(InputStream source, String charsetName)
// Scanner piped from the given source string (NOT filename string)
public Scanner(String source)
```

Formatted Text I/O

❑ Examples

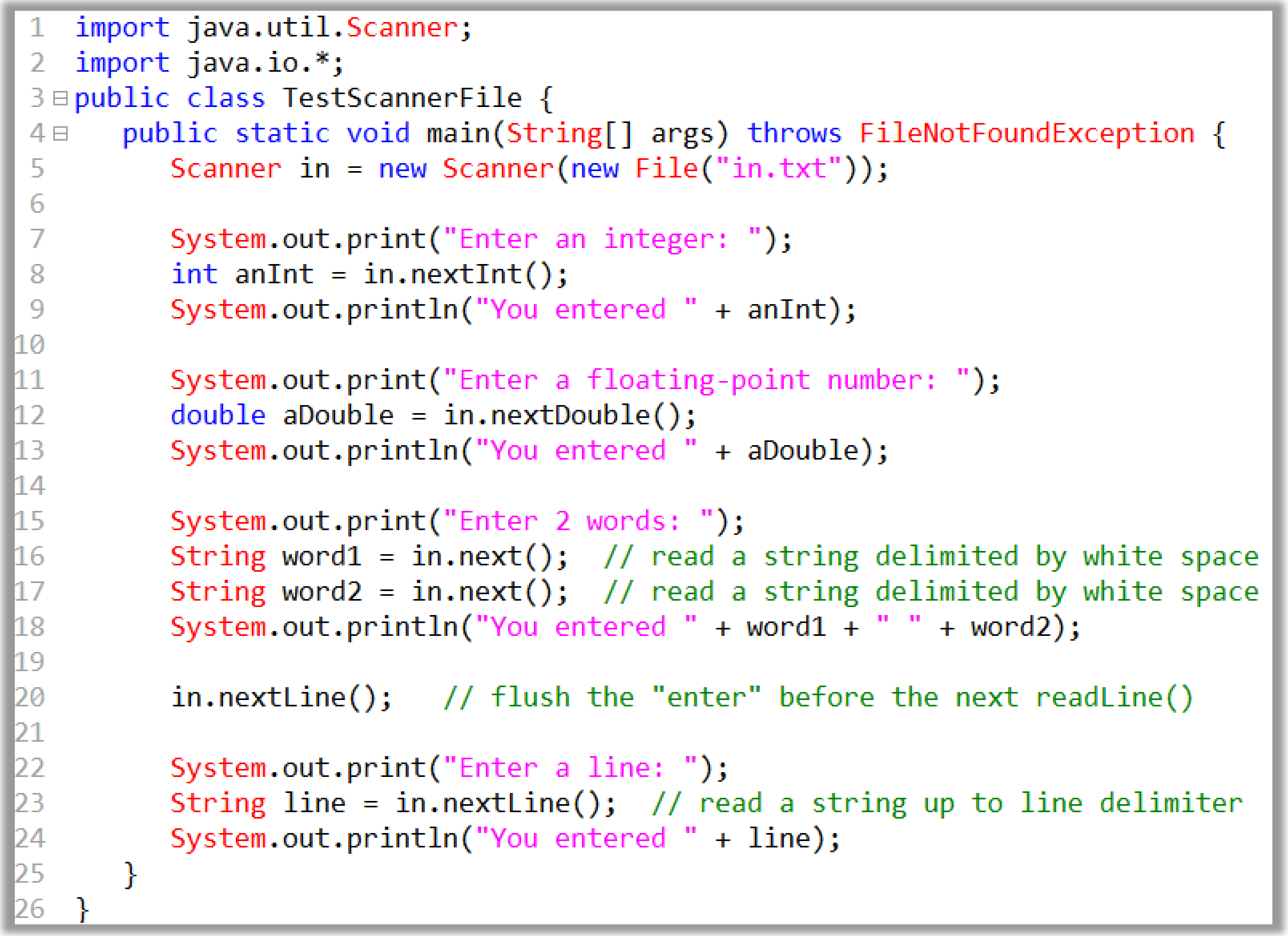
```
// Construct a Scanner to parse an int from keyboard
Scanner in1 = new Scanner(System.in);
int i = in1.nextInt();

// Construct a Scanner to parse all doubles from a disk file
Scanner in2 = new Scanner(new File("in.txt"));  // need to handle FileNotFoundException
while (in2.hasNextDouble()) {
   double d = in.nextDouble();
}

// Construct a Scanner to parse a given text string
Scanner in3 = new Scanner("This is the input text String");
while (in3.hasNext()) {
   String s = in.next();
}
```

Formatted Text I/O



```java
import java.util.Scanner;
public class TestScannerSystemIn {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int anInt = in.nextInt();
        System.out.println("You entered " + anInt);

        System.out.print("Enter a floating-point number: ");
        double aDouble = in.nextDouble();
        System.out.println("You entered " + aDouble);

        System.out.print("Enter 2 words: ");
        String word1 = in.next();  // read a string delimited by white space
        String word2 = in.next();  // read a string delimited by white space
        System.out.println("You entered " + word1 + " " + word2);

        in.nextLine();   // flush the "enter" before the next readLine()

        System.out.print("Enter a line: ");
        String line = in.nextLine();  // read a string up to line delimiter
        System.out.println("You entered " + line);
    }
}
```

Formatted Text I/O



```java
import java.util.Scanner;
import java.io.*;
public class TestScannerFile {
   public static void main(String[] args) throws FileNotFoundException {
      Scanner in = new Scanner(new File("in.txt"));

      System.out.print("Enter an integer: ");
      int anInt = in.nextInt();
      System.out.println("You entered " + anInt);

      System.out.print("Enter a floating-point number: ");
      double aDouble = in.nextDouble();
      System.out.println("You entered " + aDouble);

      System.out.print("Enter 2 words: ");
      String word1 = in.next();  // read a string delimited by white space
      String word2 = in.next();  // read a string delimited by white space
      System.out.println("You entered " + word1 + " " + word2);

      in.nextLine();   // flush the "enter" before the next readLine()

      System.out.print("Enter a line: ");
      String line = in.nextLine();  // read a string up to line delimiter
      System.out.println("You entered " + line);
   }
}
```

Thank you

zhenling@seu.edu.cn