NMK30703: Programming for Networks

Lab module 2:

Streams

Mohamed Elshaikh

Faculty of Electronics Engineering Technology – UniMAP (FTKEN-UniMAP)

Objectives

- Utilize FileInputStream class which is InputStream type and FileOutputStream class which is OutputStream type to read from and write to a file.
- Use FileReaderclass which is Reader type and FileWriter class which is Writer type to read from and write to a file.

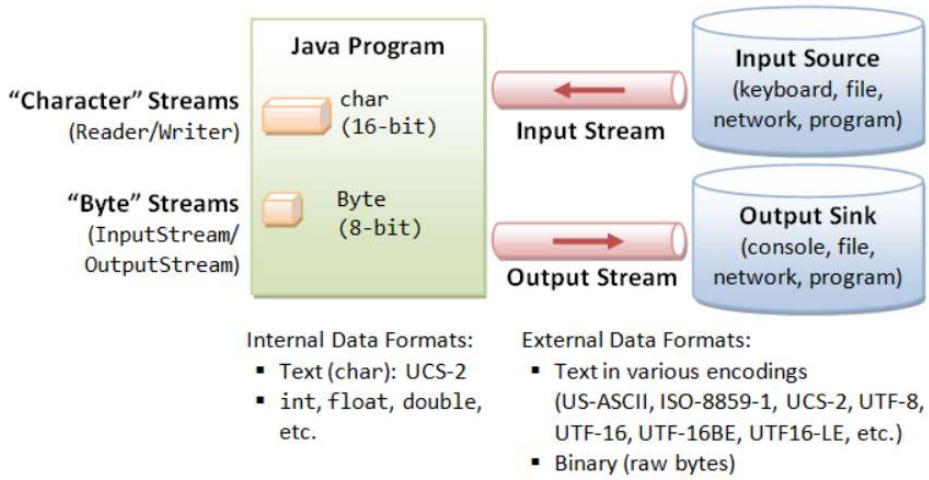
Introduction

Programs read inputs from data sources (e.g., keyboard, file, network, memory buffer, or another program) and write outputs to data sinks (e.g., display console, file, network, memory buffer, or another program). In Java standard I/O, inputs and outputs are handled by the so-called *streams*. A *stream* is a sequential and contiguous one-way flow of data (just like water or oil flows through the pipe). It is important to mention that Java does not differentiate between the various types of data sources or sinks (e.g., file or network) in stream I/O. They are all treated as a sequential flow of data. 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. All Java I/O streams are one-way (except the `RandomAccessFile`, which will be discussed later). If your program needs to perform both input and output, you have to open two streams - an input stream and an output stream.

Stream I/O operations involve three steps:

1. *Open* an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
2. *Read* from the opened input stream until "end-of-stream" encountered, or *write* to the opened output stream (and optionally flush the buffered output).
3. *Close* the input/output stream.

Java's I/O operations is more complicated than C/C++ to support internationalization (i18n). 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. As a consequence, Java needs to differentiate between byte-based I/O for processing *raw bytes* or *binary data*, and character-based I/O for processing *texts* made up of characters.



a) Reading from an InputStream

The `abstract` superclass `InputStream` declares an `abstract` method `read()` to read one data-byte from the input source:

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

The `read()` method:

- returns the input byte read as an `int` in the range of 0 to 255, or
- returns -1 if "end of stream" condition is detected, or
- throws an `IOException` if it encounters an I/O error.

The `read()` method returns an `int` instead of a `byte`, because it uses -1 to indicate end-of-stream.

The `read()` method *blocks* until a byte is available, an I/O error occurs, or the "end-of-stream" is detected. The term "*block*" means that the method (and the program) will be suspended. The program will resume only when the method returns.

Two variations of `read()` methods are implemented in the `InputStream` for reading a block of bytes into a byte-array. It returns the number of bytes read, or -1 if "end-of-stream" encounters.

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

b) Writing to an OutputStream

Similar to the input counterpart, the `abstract` superclass `OutputStream` declares an abstract method `write()` to write a data-byte to the output sink. `write()` takes an `int`. The least-significant byte of the `int` argument is written out; the upper 3 bytes are discarded. It throws an `IOException` if I/O error occurs (e.g., output stream has been closed).

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

Similar to the read(), two variations of the write() method to write a block of bytes from a byte-array are implemented:

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

c) Opening & Closing I/O Streams

You *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, which performs the necessary clean-up operations to free up the system resources.

```
public void close() throws IOException  // close this Stream
```

It is a good practice to explicitly close the I/O stream, by running `close()` in the `finally` clause of `try-catch-finally` to free up the system resources immediately when the stream is no longer needed. This could prevent serious resource leaks. Unfortunately, the `close()` method also throws a `IOException`, and needs to be enclosed in a nested `try-catch` statement, as follows. This makes the codes somehow ugly.

```
FileInputStream in = null;
......
try {
   in = new FileInputStream(...);  // Open stream
   ......
   ......
} catch (IOException ex) {
   ex.printStackTrace();
} finally {  // always close the I/O streams
   try {
      if (in != null) in.close();
   } catch (IOException ex) {
      ex.printStackTrace();
   }
}
```

JDK 1.7 introduces a new try-with-resources syntax, which automatically closes all the opened resources after try or catch, as follows. This produces much neater codes.

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

d) Flushing the OutputStream

In addition, the `OutputStream` provides a `flush()` method to flush the remaining bytes from the output buffer.

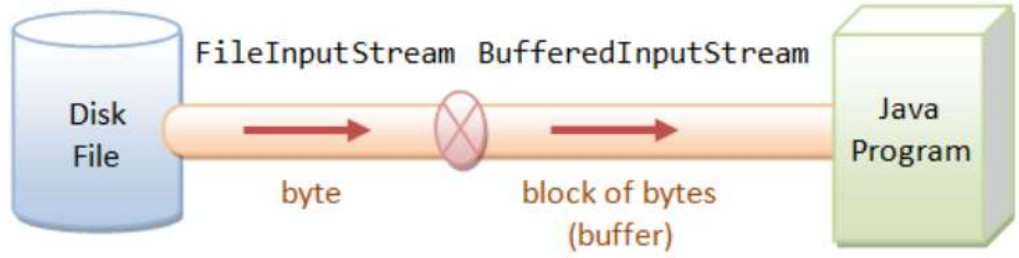
```
public void flush() throws IOException  // Flush the output
```

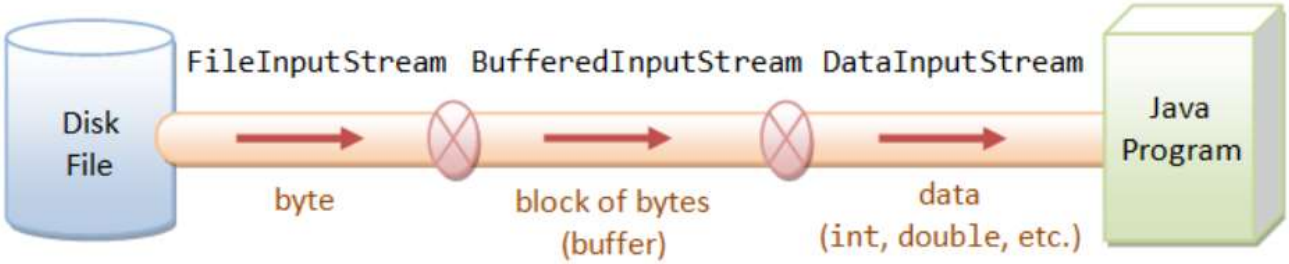
e) Implementations of abstract InputStream/OutputStream

`InputStream` and `OutputStream` are `abstract` classes that cannot be instantiated. You need to choose an appropriate concrete subclass to establish a connection to a physical device. For example, you can instantiate a `FileInputStream` or `FileOutputStream` to establish a stream to a physical disk file.

f) 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). For example, we can layer a `BufferedInputStream` to a `FileInputStream` for buffered input, and stack a `DataInputStream` in front for formatted data input (using primitives such as `int`, `double`), as illustrated in the following diagrams.





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.

1. Buffered I/O Byte-Streams

The read()/write() method in InputStream/OutputStream are designed to read/write a single byte of data on each call. This is grossly inefficient, as each call is handled by the underlying operating system (which may trigger a disk access, or other expensive operations). 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.

FileInputStream/FileOutputStream is not buffered. It is often chained to a BufferedInputStream or BufferedOutputStream, which provides the buffering. To chain the streams together, simply pass an instance of one stream into the constructor of another stream. For example, the following codes chain a FileInputStream to a BufferedInputStream, and finally, a DataInputStream:

```
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")));
```

Example 1

Copying a file byte-by-byte without Buffering.

```
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();
         }
      }
   }
}
```

Output

```
File size is 417455 bytes
Elapsed Time is 3781.500581 msec
```

This example copies a file by reading a byte from the input file and writing it to the output file. It uses FileInputStream and FileOutputStream directly without buffering. Notice that most the I/O methods "throws" IOException, which must be caught or declared to be thrown. The method close() is programmed inside the finally clause. It is guaranteed to be run after try or catch. However, method close() also throws an IOException, and therefore must be enclosed inside a nested try-catch block, which makes the codes a little ugly.

We used System.nanoTime(), which was introduced in JDK 1.5, for a more accurate measure of the elapsed time, instead of the legacy not-so-precise System.currentTimeMillis(). The output shows that it took about 4 seconds to copy a 400KB file.

As mentioned, JDK 1.7 introduces a new try-with-resources syntax, which automatically closes all the resources opened, after try or catch. For example, the above example can be re-written in a much neater manner as follow:

```
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();
      }
   }
}
```

Example 2

Copying a file with a Programmer-Managed Buffer.

```
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(); }
      }
   }
}
```

Output

```
File size is 417455 bytes
Elapsed Time is 2.938921 msec
```

This example again uses `FileInputStream` and `FileOutputStream` directly. However, instead of reading/writing one byte at a time, it reads/writes a 4KB block. This program took only 3 millisecond - a more than 1000 times speed-up compared with the previous example.

Larger buffer size, up to a certain limit, generally improves the I/O performance. However, there is a trade-off between speed-up the the memory usage. For file copying, a large buffer is certainly recommended. But for reading just a few bytes from a file, large buffer simply wastes the memory.

I re-write the program using JDK 1.7, and try on various buffer size on a much bigger file of 26MB.

```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();
         }
      }
   }
}
```

Output

```
File size is 26246026 bytes [26 MB]
   1KB: 573.54msec
   2KB: 316.43msec
   4KB: 178.47msec
   8KB: 116.32msec
  16KB:  85.61msec
  32KB:  65.92msec
  64KB:  57.81msec
 256KB:  63.38msec
1024KB:  98.87msec
```

Increasing buffer size helps only up to a certain point?!

Example 3

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(); }
      }
   }
}
```

Output

```
File size is 417455 bytes
Elapsed Time is 61.834954 msec
```

1. 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`". You can get the default charset via `static` method `java.nio.charset.Charset.defaultCharset()` or `System.getProperty("file.encoding")`. It is probable safe to use `FileReader`/`FileWriter` for ASCII texts, provided that the default charset is compatible to ASCII (such as US-ASCII, ISO-8859-x, UTF-8, and many others, but NOT UTF-16, UTF-16BE, UTF-16LE and many others). Use of `FileReader/FileWriter` is NOT recommended as you have no control of the file encoding charset.

2. 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. `BufferedReader` provides a new method `readLine()`, which reads a line and returns a `String` (without the line delimiter). Lines could be delimited by "`\n`" (Unix), "`\r\n`" (Windows), or "`\r`" (Mac).

Example 4

```
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();
      }
   }
}
```

3. 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.

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

Example 5

```
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]
   }
}
```

The default charset for file encoding is kept in the system property "`file.encoding`". To change the JVM's default charset for file encoding, you can use command-line VM option "`-Dfile.encoding`". For example, the following command run the program with default charset of UTF-8.

```
> java -Dfile.encoding=UTF-8 TestCharset
```

Most importantly, 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).

```
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.
```

The `encode()`/`decode()` methods operate on `ByteBuffer` and `CharBuffer` introduced also in JDK 1.4, which will be explain in the New I/O section.

Example 6

The following example encodes some Unicode texts in various encoding scheme, and display the Hex codes of the encoded byte sequences.

```
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();
      }
   }
}
```

Output

```
     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    !
```

Example 7

The following example tries out the encoding/decoding on `CharBuffer` and `ByteBuffer`. Buffers will be discussed later under New I/O.

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

public class TestCharsetEncodeByteBuffer {
   public static void main(String[] args) {
      byte[] bytes = {0x00, 0x48, 0x00, 0x69, 0x00, 0x2C,
                      0x60, (byte)0xA8, 0x59, 0x7D, 0x00, 0x21};  // "Hi,您好!"
      // Print UCS-2 in hex codes
      System.out.printf("%10s: ", "UCS-2");
      for (int i = 0; i < bytes.length; ++i) {
         System.out.printf("%02X ", bytes[i]);
      }
      System.out.println();

      Charset charset = Charset.forName("UTF-8");
      // Encode from UCS-2 to UTF-8
      // Create a ByteBuffer by wrapping a byte array
      ByteBuffer bb = ByteBuffer.wrap(bytes);
      // Create a CharBuffer from a view of this ByteBuffer
      CharBuffer cb = bb.asCharBuffer();
      ByteBuffer bbOut = charset.encode(cb);
      // Print hex code
      System.out.printf("%10s: ", charset.name());
      while (bbOut.hasRemaining()) {
         System.out.printf("%02X ", bbOut.get());
      }
      System.out.println();

      // Decode from UTF-8 to UCS-2
      bbOut.rewind();
      CharBuffer cbOut = charset.decode(bbOut);
      System.out.printf("%10s: ", "UCS-2");
      while (cbOut.hasRemaining()) {
         char aChar = cbOut.get();
         System.out.printf("'%c'[%04X] ", aChar, (int)aChar);  // Print char & hex code
      }
      System.out.println();
   }
}
```

Output

```
UCS-2: 00 48 00 69 00 2C 60 A8 59 7D 00 21
UTF-8: 48 69 2C E6 82 A8 E5 A5 BD 21
UCS-2: 'H'[0048] 'i'[0069] ','[002C] '您'[60A8] '好'[597D] '!'[0021]
```

4. Text File I/O

As mentioned, 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.

To choose the charset, you need to use `InputStreamReader` and `OutputStreamWriter`. `InputStreamReader` and `OutputStreamWriter` are considered to be byte-to-character "bridge" streams. You can 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)
```

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`.

Example 8

The following program writes Unicode texts to a disk file using various charsets for file encoding. It then reads the file byte-by-byte (via a byte-based input stream) to check the encoded characters in the various charsets. Finally, it reads the file using the character-based reader.

```
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();
         }
      }

      // 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();
         }
      }
   }
}
```

Output

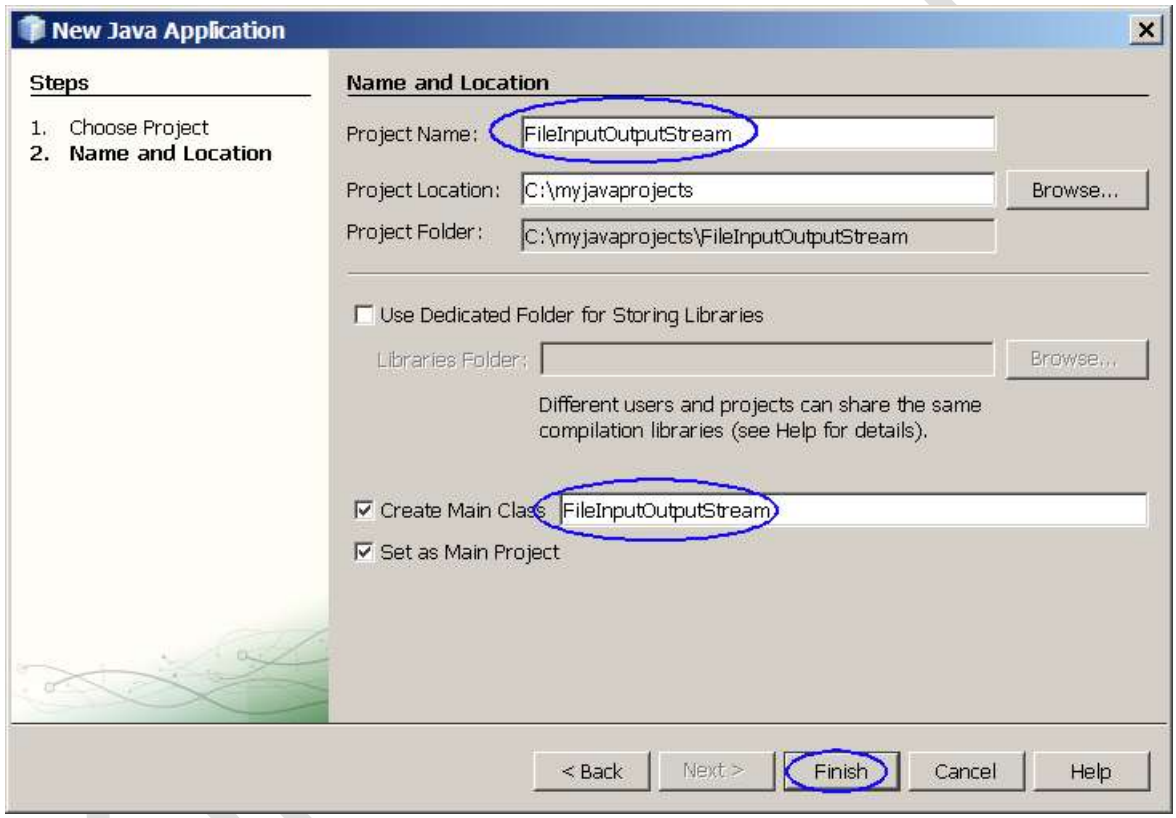
```
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

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)
```

As seen from the output, the characters 您好 is encoded differently in different charsets. Nonetheless, the `InputStreamReader` is able to translate the characters into the same UCS-2 used in Java program.

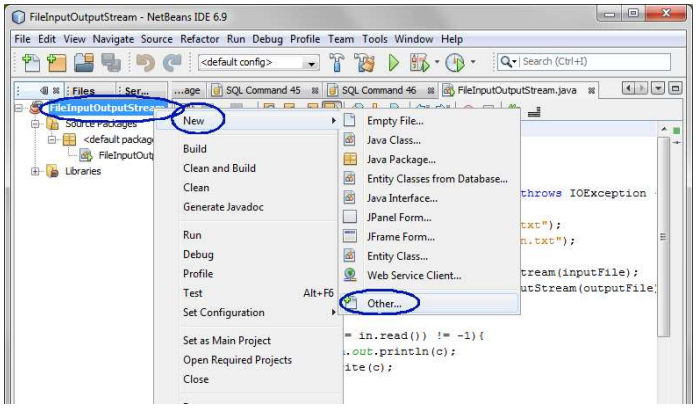
TASK: Byte Stream

Write an application that uses FileInputStream and FileOutputStream:

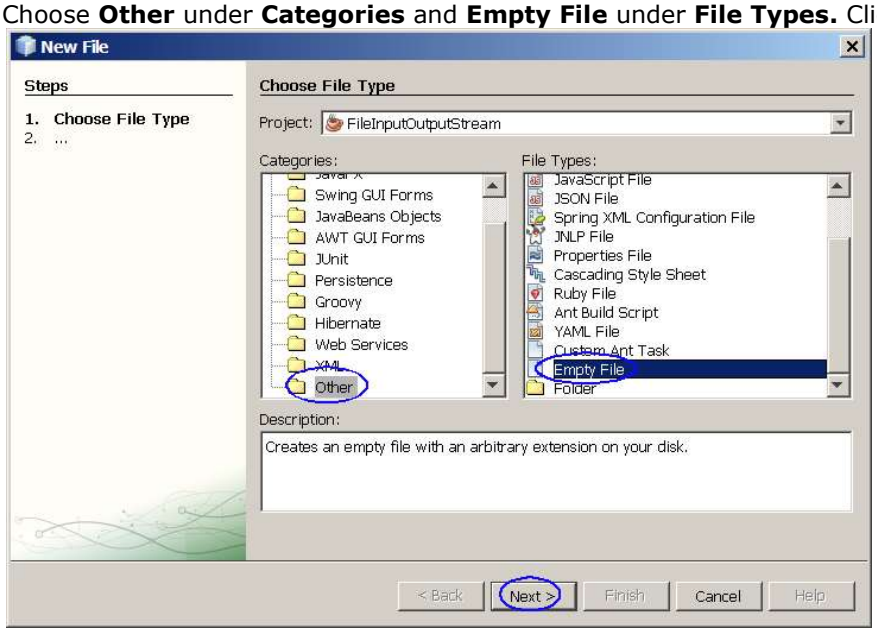
1. Create a new NetBeans project
    a. Select **File**->**New Project (Ctrl+Shift+N)**.
    b. Under **Choose Project** pane, select **Java** under **Categories** and **Java Application** under **Projects**.
    c. Click **Next**.
    d. Under **Name and Location** pane, for the **Project Name** field, type in **FileInputOutputStream** as project name. (Your Project Location may differ from the figure below).
    e. For **Create Main Class** field, type in **FileInputOutputStream**.
    f. Click **Finish**.



2. Provide **farrago.txt** as an input file. You may create this text file using any text editor program such as Notepad or using Netbeans IDE as follows:
    a. Right click **FileInputOutputStream** project and select **New->Other**.



b. Choose **Other** under **Categories** and **Empty File** under **File Types.** Click Next.



c. Observe that the **New Empty File** dialog box appears.

d. For the **File Name** field, type in **farrago.txt**. Click Finish.

e. Observe that the empty **farrago.txt** appears in the editor window. Write the contents below to the empty file:

> Subclasses of OutputStream use these methods to write data onto particular media.
>
> For instance, a FileOutputStream uses these methods to write data into a file.
>
> TelnetOutputStream uses these methods to write data onto a network connection.
>
> ByteArrayOutputStream uses these methods to write data into an expandable byte array.

3. Modify the IDE generated **FileInputOutputStream.java** as shown below.

```
   import java.io.*;

public class FileInputOutputStream {

    public static void main(String[] args) throws IOException {

        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");

        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);
        int c;

        while ((c = in.read()) != -1){
            System.out.println(c);
            out.write(c);
        }

        System.out.println("FileInputStream is used to read a file and
FileOutPutStream is used for writing.");

        in.close();
        out.close();
    }
}
```

4. Build and run the project and observe the result in the **Output** window. Explain why do you get such output?

5. Open the project directory and find the file **outagain.txt**. What is the content of the file?

6. **Re-read the codes that you have written previously in FileInputOutputStream.java and write a line of //comment at the end of each line to explain what that particular line of codes does to the program.**

7. Modify the codes so that it can display proper characters into the Output window.

8. Modify the codes to include a try-catch block.