Lab 5. I/O Stream and Character Encoding

Authors: Yida Tao, Yao Zhao

1. I/O Stream Overview

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O and handles translation between characters and bytes. `FileReader`, for example, uses `FileInputStream`, while `FileWriter` uses `FileOutputStream`. There are two general-purpose byte-to-character "bridge" streams: `InputStreamReader` and `OutputStreamWriter`. Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

1.1. FileInputStream

`FileInputStream` obtains input bytes from a file in a file system. Run `ByteReader.java` with a GB2312 encoded `sample.txt` and observe the result.

```
public class ByteReader {

    public static void main(String[] args) {

        try (FileInputStream fis = new FileInputStream("sample.txt")){

            byte[] buffer = new byte[65535];
            int byteNum = fis.read(buffer);
            System.out.println("Total number of bytes being read: " + byteNum);
            System.out.print("File content in unsigned int: ");

            for(int i = 0; i < byteNum; i++){
                // byte to unsigned int (rather than signed)
                System.out.printf("%d ", Byte.toUnsignedInt(buffer[i]));
            }
            System.out.println();
            System.out.print("File content in hexdecimal: ");
            for(int i = 0; i < byteNum; i++){
                // lowercase hex
                System.out.printf("%02x ",buffer[i]);
            }

        } catch (FileNotFoundException e) {
            System.out.println("The pathname does not exist.");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Failed or interrupted when doing the I/O
operations");
            e.printStackTrace();
        }
    }
}
```

1.2. InputStreamReader and BufferedReader

`InputStreamReader` is a bridge between a byte stream and a character stream that converts a byte stream into a character stream.

If there is no buffer, each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full. Simply put, reading/writing from/to memory is faster than reading/writing from/to a disk.

For instance, if you read 1000 bytes in unbuffered mode, you need to make 1000 system calls. In buffered mode, however, the `BufferedInputStream` reads a block of data (usually 1024 bytes) using single system call.

There are four buffered stream classes used to wrap unbuffered streams: `BufferedInputStream` and `BufferedOutputStream` create buffered byte streams, while `BufferedReader` and `BufferedWriter` create buffered character streams. Run `BufferReader.java` with the GB2312 encoded `sample.txt` and observe the result.

```
public class BufferReader {
    public static void main(String[] args) {

        try (FileInputStream fis = new FileInputStream("sample.txt");
             InputStreamReader isr = new InputStreamReader(fis, "gb2312");
             BufferedReader bReader = new BufferedReader(isr);){

            char[] cbuf = new char[16];
            int file_len = bReader.read(cbuf);

            System.out.println(file_len);
            System.out.println(cbuf);

        } catch (FileNotFoundException e) {
            System.out.println("The pathname does not exist.");
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            System.out.println("The Character Encoding is not supported.");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Failed or interrupted when doing the I/O 
operations");
            e.printStackTrace();
        }
    }
}
```

1.3. FileOutputStream

`FileOutputStream` is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using `FileWriter`.

Run `ByteWriter.java` and then open the output file `bytewriter_output.txt` and observe its encoding, which is the same as the Java platform default encoding. The reason is that we used `String.getBytes()`, which encodes a string into a sequence of bytes using the platform's default charset.

```
public class ByteWriter {

    public static void main(String[] args) {

        try (FileOutputStream fos = new FileOutputStream("bytewriter_output.txt"))
{

            String s = "计算机系统";
            byte[] bytes = s.getBytes();

            fos.write(bytes);
            fos.flush();//fos.close();
        } catch (FileNotFoundException e) {
            System.out.println("The pathname does not exist.");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Failed or interrupted when doing the I/O
operations");
            e.printStackTrace();
        }
    }
}
```

1.4. OutputStreamWriter and BufferedWriter

Run `BufferWriter.java`; then open `bufferwriter_output.txt` and observe its encoding. Next, modify "100" to 100; run the program again and observe the output. Note that for `bWriter.write(100)`, it actually writes a single character specified by the input int. Check the method documentation for details.

```
public class BufferWriter {

    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream(new
File("bufferwriter_output.txt"));
             OutputStreamWriter osw = new OutputStreamWriter(fos, "gbk");
             BufferedWriter bWriter = new BufferedWriter(osw);){
            bWriter.write("你好！\n");
//          bWriter.write(100);
            bWriter.write("100");
            bWriter.write(" 分 \n");
            bWriter.write("送给你！\n");
            bWriter.flush();//bWriter.close();

        } catch (FileNotFoundException e) {
            System.out.println("The pathname does not exist.");
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            System.out.println("The Character Encoding is not supported.");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Failed or interrupted when doing the I/O
operations");
            e.printStackTrace();
        }
    }
}
```

2. Character Encoding

There are various ways for characters to be encoded as binary data. A particular encoding is known as a character set or charset. The encoding for charsets are specified by international standards organizations and have names such as "UTF-16", "UTF-8", and "ISO-8859-1".

In `UTF-16`, characters are encoded as 16-bit UNICODE values; this is the character set that is used internally by Java. `UTF-8` is another way of encoding UNICODE characters using 8 bits for common ASCII characters and longer codes for other characters. Both `UTF-16` and `UTF-8` use variable length encodings: `UTF-16` uses either 2 or 4 bytes (instead of 1, 2, 3, or 4 bytes in `UTF-8`).

Run the following code and observe the result.

```
char c = '计';
int value = c;
System.out.printf("%s\n", c);
System.out.printf("Unicode for 计: %X\n", value);
```

As mentioned before, `String.getBytes()` encodes a string into a sequence of bytes using the platform's default charset. We could also used overloaded `String.getBytes()` by specifying the charset. Run the following code and observe the result.

```
String str = "计算机";

System.out.printf("Java platform default: ");
byte[] bytes0 = str.getBytes();
for (byte b : bytes0) {
    System.out.printf("%2X ", b);
}
System.out.println();
```

```
    System.out.printf("GBK: ");
    byte[] bytes1 = str.getBytes("GBK");
    for (byte b : bytes1) {
        System.out.printf("%2X ", b);
    }
    System.out.println();

    System.out.printf("UTF_16: ");
    byte[] bytes2 = str.getBytes(StandardCharsets.UTF_16);
    for (byte b : bytes2) {
        System.out.printf("%2X ", b);
    }
    System.out.println();

    System.out.printf("UTF_16BE: ");
    byte[] bytes3 = str.getBytes(StandardCharsets.UTF_16BE);
    for (byte b : bytes3) {
        System.out.printf("%2X ", b);
    }
    System.out.println();

    System.out.printf("UTF_16LE: ");
    byte[] bytes4 = str.getBytes(StandardCharsets.UTF_16LE);
    for (byte b : bytes4) {
        System.out.printf("%2X ", b);
    }
    System.out.println();
```