Java Programming

Input & Output (I/O)

JDK has two sets of IO packages: the Standard IO (`java.io`) (since JDK 1.0) and the New IO (`java.nio`) (introduced in JDK 1.4). In addition, JDK 1.5 introduced the formatted text-IO via new classes `Scanner`/`Formatter` and `printf()`.

Class `java.io.File`

The class `java.io.File` can represent either a *file* or a *directory*.

It also maintains two system-dependent properties, for you to write programs that are portable:

- Directory Separator: Windows systems use backslash `'\'` (e.g., `"c:\jdk\bin\java.exe"`), while Unixes use forward slash `'/'` (e.g., `"/usr/jdk/bin/java.exe"`). This system-dependent property is maintained in the static field `File.separator` (as `String`) or `File.separatorChar`. (They failed to follow the naming convention for constants, which was adopted in JDK 1.2.)
- Path Separator: Windows use semi-colon `';'` to separate paths (or directories) white Unixes use colon `':'`. This system-dependent value can be retrieved from static field `File.pathSeparator` (as `String`) or `File.pathSeparatorChar`.

The commonly used constructor in `java.io.File` is (not throwing `FileNotFoundException`?!):

```
public File(String fileOrDirName)
```

For example,

```
File file = new File("in.txt");
```

For applications that you intend to distribute as Jar files, you should use URL class to reference resources, as it can reference disk file as well as Jar'ed file , for example,

```
java.net.URL url = this.getClass().getResource("icon.png");
```

The commonly-used methods are:

```
public boolean exists()       // tests if the file/directory exists.
public boolean isDirectory()  // tests if this instance is a directory.
public boolean isFile()       // tests if this instance is a file.
public boolean canRead()      // tests if the file readable.
public boolean canWrite()     // tests if the file writeable.
public long length()          // returns the length of the file.
public boolean delete()       // deletes the file or directory.
public void deleteOnExit()    // deletes the file or directory when the program terminates.
public boolean renameTo(File dest) // renames the file
public boolean mkdir()        // create this directory.
```

For a directory, the following methods can be used to list its contents:

```
// List the contents of a directory
public String[] list()
public File[] listFiles()
// List with a filename filter
public String[] list(FilenameFilter filter)
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
```

Example 1: Recursively list the contents of a directory (similar to Unix's `"ls -r"` command).

```
// Recursively list the contents of a directory
import java.io.File;
public class ListDirectoryRecusive {
   public static void main(String[] args) {
      File dir = new File("d:\\bin\\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);
         }
      }
   }
}
```

Example 2: Filename filter

```
// List all the files end with ".java"
import java.io.File;
import java.io.FilenameFilter;
public class ListDirectoryWFilter {
   public static void main(String[] args) {
      File dir = new File(".");     // current working directory
      if (dir.isDirectory()) {
         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);
         }
      }
   }
}
```

The interface `java.io.FilenameFilter` declares one abstract method:

```
public boolean accept(File dirname, String filename)
```

where *`dirname`* and *`filename`* are the directory name and filename respectively. The `list()` method

does a *callback* to `accept()` for each of the file/sub-directory produced. You can specify your filtering criteria to the *`dirname`* and *`filename`* in `accept()`. Those files/sub-directories that result in a `false` return will be excluded.

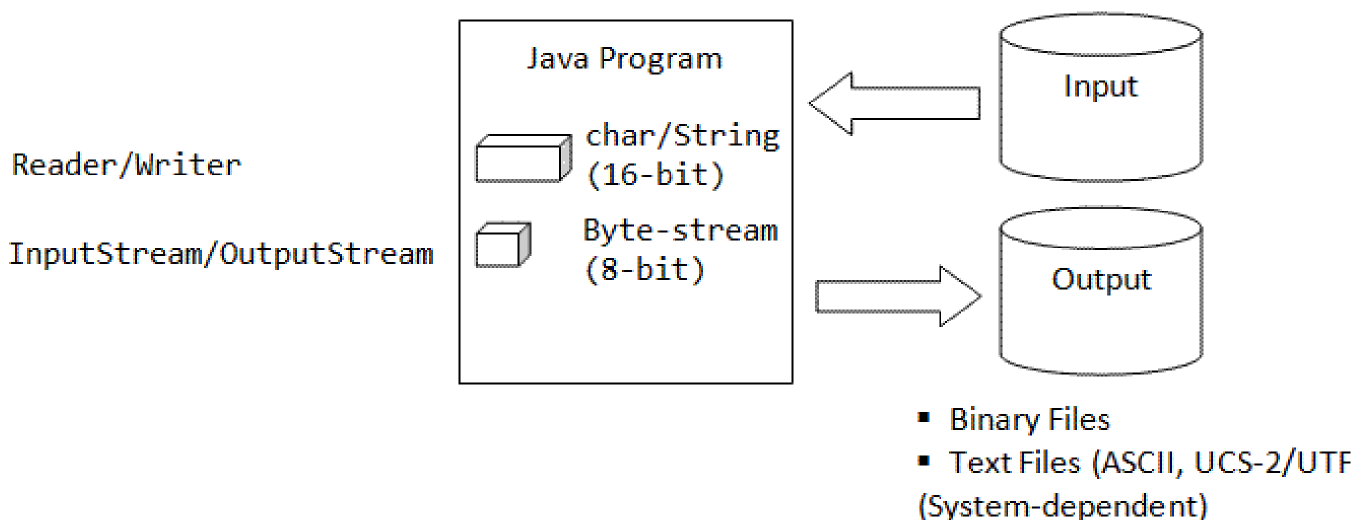
IO Operations in Standard IO package (java.io)

Programs read inputs from data sources (e.g., keyboard, file, network, or another program) and write outputs to data sinks (e.g., console, file, network, or another program). Inputs and outputs in Java are handled by the so-called *stream*. A *stream* is a sequential and continuous one-way flow of information (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). They are all treated as a sequential flow of data. The 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 data source by opening an input stream, and sends data to data sink by opening an output stream. All Java I/O streams are one-way (except the RandomAccessFile, which will be covered later). If your program needs to perform both input and output, you have to open two separate streams - an input stream and an output stream.

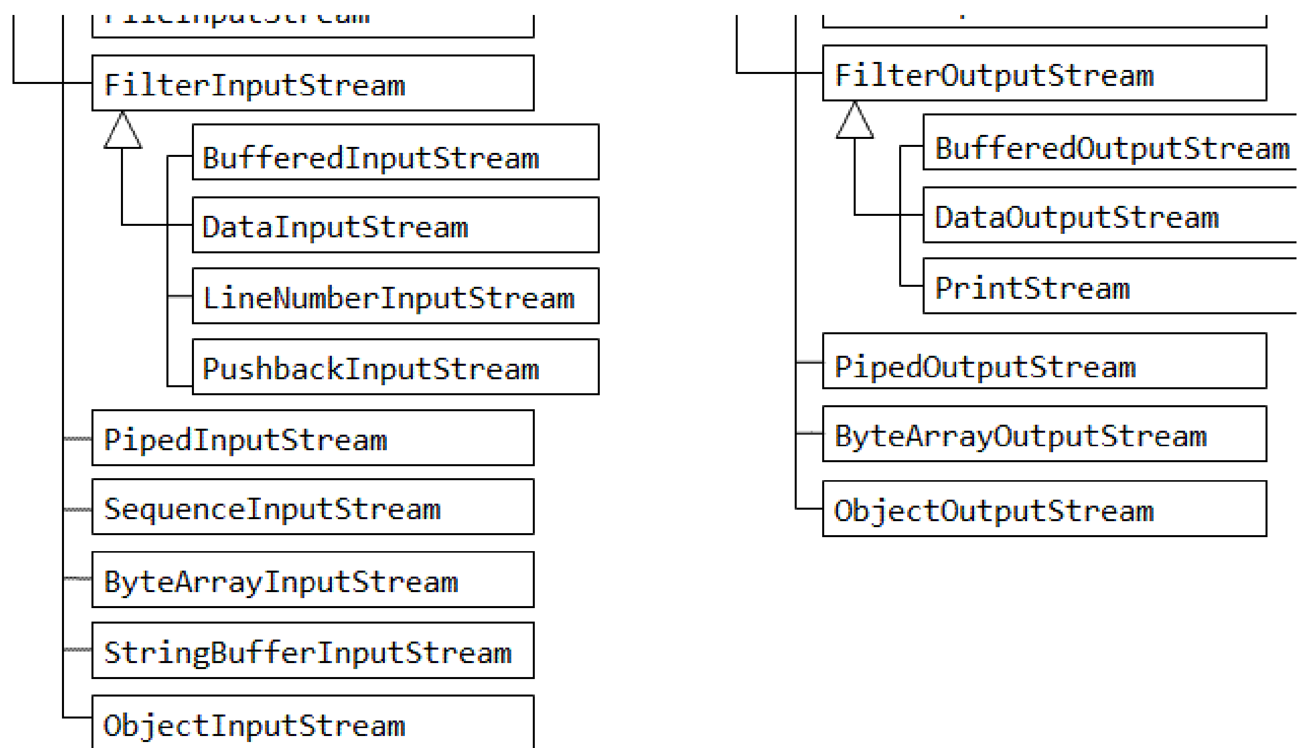
IO operations involved three steps:

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

Java's IO operations is more complicated than C/C++, as Java uses 16-bit character set (instead of 8-bit character set). As a consequence, it needs to differentiate between byte-based IO and character-based IO.



Byte-Based IO & Byte Streams



Byte streams can be used to read or write bytes serially from an external device. All the byte streams are derived from the abstract superclass `InputStream` and `OutputStream`, as illustrated in the above class diagram.

The abstract superclass `InputStream` declares an abstract method `read()` to read one data-byte from the input source, and convert the unsigned byte value (of 0 to 255) to an `int`. The `read()` method will *block* until a byte is available, an I/O error occurs, or the "end of stream" is reached. It returns -1 if for "end of stream", and throws an `IOException` if it encounters an I/O error.

```
public abstract int read() throws IOException  // read one data-byte from the input stream
```

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

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

Similar to the input counterpart, the abstract superclass `OutputStream` declares an abstract method `write()` to write a data-byte to the output sink. The least significant byte of the `int` argument is written out; while the upper 3 bytes are discarded.

```
public void abstract void write(int unsignedByte) throws IOException  // write one data-byte
```

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

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

Both the `InputStream` and the `OutputStream` provides a `close()` method to close the stream, which performs the necessary clean-up operations as well as frees the system resources. (Although it is not absolutely necessary to close the IO streams explicitly, as the garbage collector will do the job, it is probably a good practice to do it to free up the system resources immediately when the streams are no longer needed.)

```
public void close() throws IOException  // close the opened Stream
```

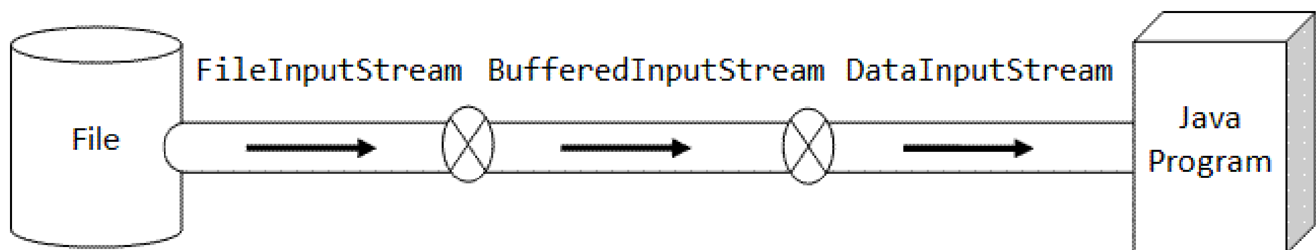
In addition, the `OutputStream` provides a `flush()` method to flush the remaining bytes from the output buffer.

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

`InputStream` and `OutputStream` are abstract classes that cannot be instantiated. You need to choose an appropriate concrete implementation subclass (such as `FileInputStream` and `FileOutputStream`) to establish a connection to a physical device (such as file, network, keyboard/console).

Layered (or Chained) IO Streams

The IO streams are often layered or chained with other IO steams, for purposes such as buffering, filtering or format conversion. For example, we can layer a `BufferedInputStream` to a `FileInputStream` for buffered input, and stack a `DataInputStream` in front for formatted data input, as illustrated in the following diagram.



File IO Streams - `FileInputStream` & `FileOutputStream`

`FileInputStream` and `FileOutputStream` are concrete implementation to the abstract class of `InputStream` and `OutputStream`, to support File IO.

Buffered IO Streams - `BufferedInputStream` & `BufferedOutputStream`

The `InputStream` and `OutputStream` is designed to read/write a single byte of data, which is grossly inefficient. Buffering is commonly applied to speed up the IO operations.

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 fin = new FileInputStream("in.dat");
BufferedInputStream bin = new BufferedInputStream(fin);
DataInputStream din = new DataInputStream(bin);
// or
DataInputStream in = new DataInputStream(
```

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

Example 1: Copying a file without Buffering

```
import java.io.*;
public class FileCopyNoBuffer {
   public static void main(String[] args) {
      File fileIn;
      FileInputStream in = null;
      FileOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      try {
         fileIn = new File("test-in.jpg");
         System.out.println("File size is " + fileIn.length() + " bytes");
         in = new FileInputStream(fileIn);
         out = new FileOutputStream("test-out.jpg");

         startTime = System.nanoTime();
         int byteRead;
         // Read a unsigned byte (0-255) and padded to 32-bit int
         while ((byteRead = in.read()) != -1) {
            // Write the least significant byte, 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 streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) { ex.printStackTrace(); }
      }
   }
}
```

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

The first example copies a file by reading a byte from an input file and writing it to the output file. It uses `FileInputStream` and `FileOutputStream` directly without buffer. Notice that most the IO methods "throws" `IOException`, which must be caught by the program or declared to be thrown to the calling method. The method `close()` is often called inside the "`finally`" clause. 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).

I used `System.nanoTime()`, which is new in JDK 1.5, for a more accurate measure of the elapsed time, instead of the legacy and not-so-precise `System.currentTimeMillis()`.

The output shows that it took about 4 seconds to copy a 400KB file.

Example 2: Copying a file with a Programmer-Managed Buffer

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

```
public class FileCopyUserBuffer {
   public static void main(String[] args) {
      File fileIn;
      FileInputStream in = null;
      FileOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      try {
         fileIn = new File("test-in.jpg");
         System.out.println("File size is " + fileIn.length() + " bytes");
         in = new FileInputStream(fileIn);
         out = new FileOutputStream("test-out.jpg");
         startTime = System.nanoTime();
         byte[] byteBuf = new byte[4096];    // 4K 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 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.

Below are the elapsed times for the various buffer size (Warning: The timing depends on many factors, your processor speed, OS, RAM size, etc.):

- 0.5 KB: 13.8 msec
- 1 KB: 6.9 msec
- 2 KB: 4.6 msec
- 4 KB: 2.9 msec
- 8 KB: 2.0 msec
- 16 KB: 1.76 msec
- 32 KB: 1.58 msec
- 64 KB: 1.33 msec
- 128 KB: 2.03 msec
- 256 KB: 1.74 msec
- 512 KB (larger than file size): 6.13 msec

Larger buffer size, up to a certain limit, generally improves the IO 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 waste the memory.

Example 3: Copying a file with Buffered Streams

```
import java.io.*;
public class FileCopyBuffered {
   public static void main(String[] args) {
      File fileIn;
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      try {
         fileIn = new File("test-in.jpg");
         System.out.println("File size is " + fileIn.length() + " bytes");
         in  = new BufferedInputStream(new FileInputStream(fileIn));
         out = new BufferedOutputStream(new FileOutputStream("test-out.jpg"));
         startTime = System.nanoTime();
         int byteRead;
         while ((byteRead = in.read()) != -1) {
            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 417455 bytes
Elapsed Time is 61.834954 msec
```

In this example, I chained the `FileInputStream` with `BufferedInputStream`, `FileOutputStream` with `BufferedOutputStream`, and read/write byte-by-byte. The JRE decides on the buffer size. The program took 62 milliseconds, about 60 times speed-up compared with example 1, but slower than the programmer-managed buffer.

DataInputStream & DataOutputStream

The `DataInputStream` and `DataOutputStream` can be stacked on top of any `InputStream` and `OutputStream` to filter the streams 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")));
```

`DataInputStream` implements `DataInput` interface, which provides methods to read formatted primitive-type inputs, such as:

```
public final int readInt() throws IOExcpetion;
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;
public final boolean readBoolean() throws IOExcpetion;
public final float readFloat() throws IOExcpetion;
public final int readUnsignedByte() throws IOExcpetion;   // [0, 255] upcast to int
public final int readUnsignedShort() throws IOExcpetion;  // [0, 65535], same as char, upcast
```

Similarly, you can stack the `DataOutputStream` as follows:

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

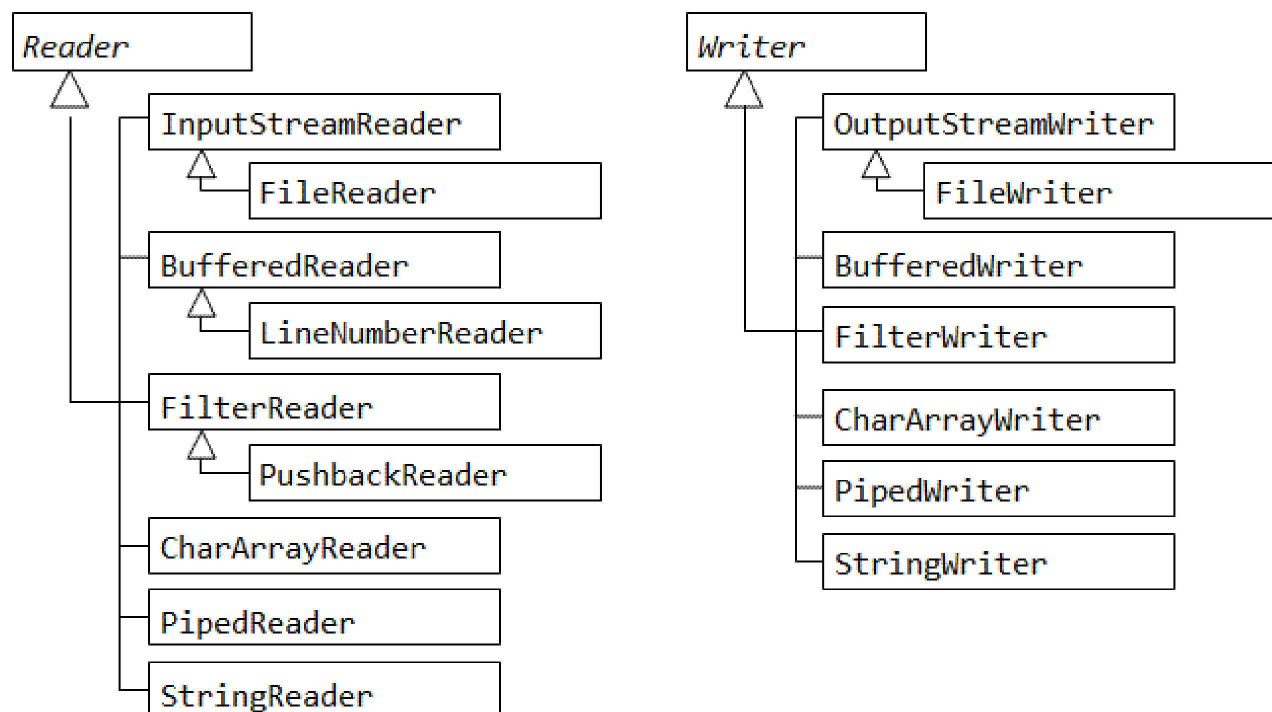
`DataOutputStream` implements `DataOutput` interface, which provides methods to write formatted primitive-type input. For examples,

```
public final void writeInt() throws IOExcpetion;
public final void writeDoube() throws IOExcpetion;
// others similar to DataInput interface
```

Network IO

[PENDING]

Character-Based IO & Character Streams



Java uses 16-bit character set internally, instead of 8-bit character set. Hence, it has to differentiate between byte-based IO (for binary data), and character-based text IO. Other than the unit of operation, character-based IO is almost identical to byte-based IO.

Instead of `InputStream` and `OutputStream`, `Reader` and `Writer` shall be used for character-based IO.

The abstract superclass `Reader` operates on characters (primitive `char` or unsigned 16-bit integer from 0 to 65535), it similarly declares an abstract method `read()` to read one character from the input source, and convert the `char` to an `int`; and variations of `read()` to read a block of characters.

```
public abstract int read() throws IOException
public int read(byte[] bytesBuffer, int offset, int length) throws IOException
public int read(byte[] bytesBuffer) throws IOException
```

The abstract superclass `Writer` similarly declares an abstract method `write()`, to write a 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(byte[] bytesBuffer, int offset, int length) throws IOException
public void write(byte[] bytesBuffer) throws IOException
```

Example:

```
import java.io.*;
public class FileCopyCharBased {
   public static void main(String[] args) {
      BufferedReader in = null;
      BufferedWriter out = null;
      try {
         in  = new BufferedReader(new FileReader("in.txt"));
         out = new BufferedWriter(new FileWriter("out.txt"));
         int charRead;
         while ((charRead = in.read()) != -1) {
            System.out.printf("%c ", (char)charRead);
            out.write(charRead);
         }
      } 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(); }
      }
   }
}
```

Java Internal Character vs. External Devices

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 between the program and the external IO devices. This is because the extenal IO device may store a character in 8-bit format (e.g., ASCII file) or 16-bit format (e.g., UCS-16) or encoded (e.g., UTF-8 or UTF-16). When 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. You can run the above example on a 8-bit ASCII text file and a 16-bit Unicode text file and

observe the result.

Unicode Files (UCS-2, UTF-8, UTF-16)

Java programs use 16-bit raw Unicode (or UCS-2) internally. But external Unicode files could be unencoded (i.e., UCS-2) or encoded in UTF-8 or UTF-16 format. Read "Data Representation: Character Encoding" for details about Unicode character set and Unicode encoding.

Two classes `InputStreamReader` and `OutputStreamWriter` can be used to convert unencoded UCS-2 used internally in the Java program to UTF-8 byte-stream for the external file. [Can `FileReader` and `FileWriter` handle it automatically??]

You can specify the character set in the `InputStreamReader`'s contructor:

```
public InputStreamReader(InputStream in)   // default charset
public InputStreamReader(InputStream in, String charsetName) throws UnsupportedEncodingExcept
public InputStreamReader(InputStream in, Charset cs)
public InputStreamReader(InputStream in, CharsetDecoder dec)
```

The supported and commonly-used charset are:

- "`US-ASCII`": 7-bit ASCII (ISO646-US)
- "`ISO-8859-1`"
- "`UTF-8`"
- "`UTF-16`", "`UTF-16BE`" (big-endian), "`UTF-16LE`" (little-endian)

Example of writing Unicode texts to a UTF-8 file,

```
FileOutputStream fos = new FileOutputStream("test.txt");   // byte-based stream
Writer out = new OutputStreamWriter(fos, "UTF8");   // convert UCS-2 character to UTF-8 byte-
out.write(str);
out.close();
```

Example of reading Unicode texts from a UTF-8 file:

```
FileInputStream fis = new FileInputStream("test.txt");       // byte-based stream
InputStreamReader isr = new InputStreamReader(fis, "UTF8");  // convert UTF-8 byte-stream to
Reader in = new BufferedReader(isr);
int charRead;
while ((charRead = in.read()) != -1) { ... }
in.close();
```

Example: The following program writes an UTF-8 file, and reads it back via character-based input and byte-based input.

```
import java.io.*;
public class UnicodeFileIO {
   public static void main(String[] args) {
      try {
         FileOutputStream fos = new FileOutputStream("TestUTF8.txt");
         Writer out = new OutputStreamWriter(fos, "UTF8");   // Output file UTF-8 encoded
         out.write("Hello, 您好\n");
         // UCS-2: 您(60A8H), 好(597DH)
         // UTF-8: 您(E6 82 A8), 好(E5 A5 BD)
         out.close();
```

```
         // Character-based (or text-based) input
         FileInputStream fis = new FileInputStream("TestUTF8.txt");
         Reader in = new InputStreamReader(fis, "UTF8");
         int charRead;
         int charCount = 0;
         while ((charRead = in.read()) != -1) {
            charCount++;
            System.out.printf("%c(%02XH) ", (char)charRead, charRead);
         }
         System.out.println("\nNumber of characters read = " + charCount);
         in.close();

         // Byte-based (or stream-based) input
         fis = new FileInputStream("TestUTF8.txt");
         int byteRead;
         int byteCount = 0;
         while ((byteRead = fis.read()) != -1) {
            byteCount++;
            System.out.printf("%02XH ", (byte)byteRead);
         }
         System.out.println("\nNumber of bytes read = " + byteCount);
         fis.close();
      } catch (FileNotFoundException ex) {
         ex.printStackTrace();
      } catch (UnsupportedEncodingException ex) {
         ex.printStackTrace();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

```
H(48H) e(65H) l(6CH) l(6CH) o(6FH) ,(2CH)  (20H) 您(60A8H) 好(597DH)
(0AH)
Number of characters read = 10

48H 65H 6CH 6CH 6FH 2CH 20H E6H 82H A8H E5H A5H BDH 0AH
Number of bytes read = 14
```

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 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).

In Java, object that requires to be serialized must implement `java.io.Serializable` or `java.io.Externalizable` interface. `Serializable` interface is an *empty* interface (or *tagged* 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 o) throws IOException;
```

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

For example, to write objects to a file:

```
ObjectOutputStream out =
   new ObjectOutputStream(
      new BufferedOutputStream(
         new FileOutputStream("object.dat")));
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()` which returns an `Object` and downcast it back to its original type.

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

Example: Object serialization

```
import java.io.*;

public class ObjectSerializationTest {
   public static void main(String[] args) {
      ObjectInputStream in = null;
      ObjectOutputStream out = null;
      try {
         out = new ObjectOutputStream(new BufferedOutputStream(
                  new FileOutputStream("object.dat")));

         // Create an array of 10 SerializedObjects with ascending numbers
         SerializedObject[] objs = new SerializedObject[10];
         for (int i = 0; i < objs.length; i++) {
            objs[i] = new SerializedObject(i);
         }
         // Write the 10 objects to file, one by one.
         for (int i = 0; i < objs.length; i++) {
            out.writeObject(objs[i]);
         }
         // Write the entire array in one go.
         out.writeObject(objs);
         out.close();

         in = new ObjectInputStream(new BufferedInputStream(
               new FileInputStream("object.dat")));
         // Read back the objects, cast back to its original type.
         SerializedObject objIn;
```

```
         for (int i = 0; i < objs.length; i++) {
            objIn = (SerializedObject)in.readObject();
            System.out.println(objIn.getNumber());
         }
         SerializedObject[] objArrayIn;
         objArrayIn = (SerializedObject[])in.readObject();
         for (SerializedObject o : objArrayIn) {
            System.out.println(o.getNumber());
         }
         in.close();
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}

class SerializedObject implements Serializable {
   private int number;

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

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

Primitive types and array are, by default, serializable.

The `ObjectInputStream` and `ObjectOutputStream` implement `DataInput` and `DataOutput` interface respectively. You can used methods such as `readInt()`, `readDouble()`, `writeInt()`, `writeDouble()` for reading and writing primitive types.

Notes on Serialization:

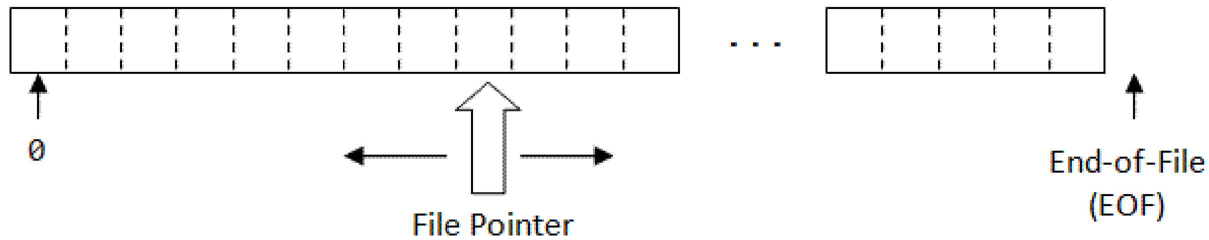
- `static` fields are not serialized.
- To prevent certain fields from being serialized, mark them using the keyword `transient`. This could cut down the amount of data traffic.
- The `writeObject()` method writes out the class of the object, the class signature, and values of non-`static` and non-`transient` fields.

Random Access Files

All the I/O streams covered so far are one-way streams. That is, they are either read-only input stream or write-only output stream. Furthermore, they are all sequential access streams, meant for reading and writing data sequentially. It is sometimes necessary to read a file record directly as well as modifying existing records or inserting new records. The class `RandomAccessFile` provides supports for non-sequential, direct (or random) access to a disk file. `RandomAccessFile` is a two-way stream, supporting both input and output operations within the same stream.

`RandomAccessFile` can be treated as a huge byte array. You can use a file pointer (of type `long`), similar to array index, to access individual byte or group of bytes in primitive types (such as int and double). The file pointer is located at 0 when the file is opened. It advances automatically for every read and write

operation by the number of bytes processed.



In constructing a `RandomAccessFile`, you can use flags `'r'` or `'rw'` to indicate whether the file is "read-only" or "read-write" access, e.g.,

```
RandomAccessFile f1 = new RandomAccessFile("filename", "r");
RandomAccessFile f2 = new RandomAccessFile("filename", "rw");
```

The following methods are available:

```
// Position the file pointer for subsequent read/write operation.
public void seek(long pos) throws IOException;
// Move the file pointer forward by the specified number of bytes.
public int skipBytes(int bytes) throws IOException;
// Obtain the position of the current file pointer, in bytes, from the beginning of the file.
public long getFilePointer() throws IOException;
// Returns the length of the random access file.
public long length() throws IOException;
```

`RandomAccessFile` does not inherit from `InputStream` or `OutputStream`. However, it implements `DataInput` and `DataOutput` interfaces (similar to `DataInputStream` and `DataOutputStream`). Therefore, you can use various methods to read/write primitive types to the file, e.g.,

```
public int readInt() throws IOException;
public double readDouble() throws IOException;
public void writeInt(int i) throws IOException;
public void writeDouble(double d) throws IOException;
```

Example: Read and write records from a `RandamAccessFile`. (A student file consists of student record of name (String) and id (`int`)).

[PENDING]

Compressed I/O Streams

The classes `ZipInputStream` and `ZipOutputStream` (in package `java.util`) support reading and writing of compressed data in ZIP format. The classes `GZIPInputStream` and `GZIPOutputStream` (in package `java.util`) support reading and writing of compressed data in GZIP format.

Example: Reading and writing ZIP files

[@PENDING]

Example: Reading and writing JAR files

[@PENDING]

Formatted Text IO (JDK 1.5)

Formatted Text Input via Scanner

JDK 1.5 introduced `java.util.Scanner` class, which greatly simplifies formatted text input from text files or keyboard (or `System.in`). `Scanner`, as the name implied, can scan for texts from an input source. It can also scan and parse text into primitive type. It first breaks the text inputs into *tokens* using a delimiter pattern, which is by default whitespace (blank, tab and newline). The tokens may then be converted into primitive values of different types using the various `nextXxx()` methods (such as `nextInt()` and `nextDouble()`). Regular expression can optionally be applied for matching.

The commonly-used constructors are:

```
// 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
public Scanner(InputStream source)
public Scanner(InputStream source, String charsetName)
// Scanner piped from a New IO's channel
public Scanner(ReadableByteChannel source)
public Scanner(ReadableByteChannel source, String charsetName)
// Scanner piped from the given source string (NOT filename string)
public Scanner(String source)
```

The last constructor pipes the text input from the `String` given in the argument. The string is not used as a filename. For example,

```

// Construct a Scanner to scan a given text string.
java.util.Scanner in1 = new Scanner("This is the input");
// Construct a Scanner piped from a file.
java.util.Scanner in2 = new Scanner(new File("in.txt"));  // need to handle FileNotFoundExcep
```

Example 1: The most common usage of `Scanner` is to read texts and values of primitive types form the keyboard (`System.in`), as follows:

```
java.util.Scanner in = new Scanner(System.in);  // piped from standard input device, default
// Use the default delimiter of whitespace (blank, newline and tab) to break up tokens
String str = in.next();           // reads the next token as String
int anInt = in.nextInt();         // scans and parses the next token as int
Double aDouble = in.nextDouble(); // scans and parses the next token as double
```

The `nextXxx()` methods throw `InputMismatchException` if the next token does not match the type to be parsed.

Example 2: Reading from a text file.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class FileScanner {
   public static void main(String[] args) throws FileNotFoundException {
      Scanner in = new Scanner(new File("input.txt"));
```

```
      double d;
      while (in.hasNextDouble()) {   // iterate thru the input source
         d = in.nextDouble();
         System.out.println(d);
      }
   }
}
// input.txt
// 1.1 2.2 3.3
// 4.4
```

The `Scanner` class implements `iterator<String>` interface. You can use `hasNext()` coupled with `next()` to iterate through all the `String` tokens. You can also directly iterate through the primitive types via methods `hasNextXxx()` and `nextXxx()`. `Xxx` includes all primitive types (`byte`, `short`, `int`, `long`, `float`, `double` and `boolean`), `BigInteger` and `BigNumber`. `char` is not included but can be retrieved from `String` via `charAt()`.

Instead of the default whitespace, you can set the delimiter to a chosen string or a regular expression via these methods:

```
public Pattern delimiter()
public Scanner useDelimiter(Pattern pattern)
public Scanner useDelimiter(String pattern)
```

Example 3: Customized token delimiter

```
import java.util.Scanner;
public class ScannerWithDelimiter {
   public static void main(String[] args) {
      Scanner in = new Scanner("one apple 2 apple red apple big apple 5.5 apple");
      // Zero or more whitespace, followed by 'apple', followed by zero or more whitespace.
      in.useDelimiter("\\s*apple\\s*");
      // The delimiter breaks the input into tokens {"one", "2", "red", "big", "5.5"}.
      System.out.println(in.next());
      System.out.println(in.nextInt());      // parses text into int
      System.out.println(in.next());
      System.out.println(in.next());
      System.out.println(in.nextDouble());   // parses text into double
   }
}
```

You can use the following methods to find the next occurrence of the specified pattern using regular expressions:

```
public String findInLine(Pattern pattern)
public String findInLine(String pattern)
public String findWithinHorizon(Pattern pattern, int horizon)
public String findWithinHorizon(String pattern, int horizon)
public Scanner skip(Pattern pattern)
public Scanner skip(String pattern)
```

Example 4: [PENDING]

Formatted Text Printing with `printf()`

JDK 1.5 introduced C-like `printf()` method (in classes `java.io.PrintSteam` and

java.io.PrintWriter) for formatted-text printing. printf() takes this syntax:

```
public PrintStream|PrintWriter printf(String format, Object... args)
public PrintStream|PrintWriter printf(Locale l, String format, Object... args)
```

printf() takes a variable number of arguments (or varargs). Varargs was introduced in JDK 1.5 (that is the reason Java cannot support printf() earlier).

printf() can be called from System.out, as System.out is a PrintStream. For example,

```
System.out.printf("Hello %4d %6.2f %s, and\n   Hello again\n", 123, 5.5, "Hello");
```

```
Hello  123   5.50 Hello, and
   Hello again
```

Format Specifier: A format specifier begins with '%' and ends with a conversion-type character (e.g. "%d" for integer, "%f" for float and double), with optional parameters in between, as follows:

```
%[argument_position$][flag(s)][width][.precision]conversion-type-character
```

- The optional *argument_position* specifies the position of the argument in the argument list. The first argument is "1$", second argument is "2$", and so on.
- The optional *width* indicates the minimum number of characters to be output.
- The optional *precision* restricts the number of characters (or number of decimal places for float-point numbers).
- The mandatory *conversion-type-character* indicates how the argument should be formatted. For examples: 'b', 'B' (boolean), 'h', 'H' (hex string), 's', 'S' (string), 'c', 'C' (character), 'd' (decimal integer), 'o' (octal integer), 'x', 'X' (hexadecimal integer), 'e', 'E' (float-point number in scientific notation), 'f' (floating-point number), '%' (percent sign). The uppercase conversion code (e.g., 'S') formats the texts in uppercase.
- Flag: '-' (left-justified), '+' (include sign), ' ' (include leading space), '0' (zero-padded), ',' (include grouping separator). '(' (negative value in parentheses), '#' (alternative form).

Examples:

```
System.out.printf("%2$2d %3$2d %1$2d\n", 1, 12, 123, 1234);
```

```
12 123  1
```

```
System.out.printf(Locale.FRANCE, "e = %+10.4f\n", Math.PI);
```

```
e =    +3,1416
```

```
System.out.printf("Revenue: $ %(,.2f, Profit: $ %(,.2f\n", 12345.6, -1234.5678);
```

```
Revenue: $ 12,345.60, Profit: $ (1,234.57)
```

Formatted-Text Output via Formatter

JDK 1.5 introduced Scanner for formatted text input. It also introduced java.util.Formatter for formatted text output.

The Formatter has the following constructors:

```
public Formatter(File file)
public Formatter(String filename)
```

```
public Formatter(OutputStream os)
public Formatter(PrintStream ps)
```

The `format()` method can be used to write formatted text output:

```
public Formatter format(String format, Object... args)
public Formatter format(Locale l, String format, Object... args)
```

Notice that the method `format()` has the same syntax as the method `printf()`, using the same set of format specifier as `printf()`.

Other methods are:

```
public void flush()   // flush out all the buffered output
public void close()
```

Example:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Formatter;
import java.util.Scanner;

public class FormatterTest {
   public static void main(String[] args) {
      try {
         Formatter out = new Formatter("out.txt");  // override existing file
         out.format("%3d %s\n", 12, "Hello-world");
         out.format("%3d %s\n", 345, "Hello");
         out.format("%3d %s\n", 6789, "again");
         out.close();

         // Set up text file input using a scanner and read records
         Scanner in = new Scanner(new File("out.txt"));
         while (in.hasNext()) {
            int id = in.nextInt();
            String name = in.next();
            System.out.printf("%4d %s\n", id, name);
         }
      } catch (FileNotFoundException ex) { // for Formatter and Scanner
         ex.printStackTrace();
      }
   }
}
```

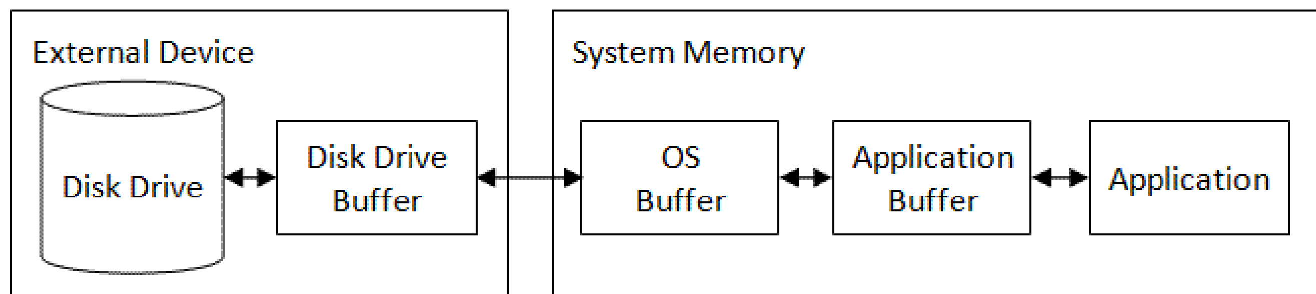
Formatter vs. PrintWriter/PrintStream

[PENDING]

New IO (java.nio) (JDK 1.4)

New IO package was introduced in JDK 1.4 to support:

- High-performance file IO
- Scalable network IO
- Characters sets



Physical IO operation is thousands times slower than memory access. Hence, a chunk of data is often cache or buffer to improve the throughtput. As illustrated from the above diagram, many layers of cache exist between you Java application and physical disk.

- Disk Buffer is RAM that is built into the disk drive to store a block of data from the disk. The cost of transferring data from the disk surface to the disk buffer is by far the slowest and the most expensive operation, bacuase it involves physical movement of the disk.
- OS Buffer: OS does its own buffering as it can cache more data and manage it more elegantly. This buffer can also be shared among the applications.
- Application Buffer: Application may optionally buffer its own data.

Channels (`java.nio.channels.Channel`)

Channel represents a connection to a physical IO device, such as file, network socket, or even another program. It is similar to Standard IO's stream, but a more platform-dependent version of stream. Because channels have a closer ties to the underlying platform, they can achieve better IO throughput. The types of channel include:

- `FileChannel`
- `SocketChannel`: support non-blocking connection.
- `DatagramChannel`: Datagram-oriented socket (i.e., UDP).

A `Channel` object can be obtained by calling the `getChannel()` methods of classes such as `java.io.FileInputStream`, `java.io.FileOutputStream`, `java.io.RandomAccessFile`, `java.net.Socket`, `java.net.ServerSocket`, `java.net.DatagramSocket`, and `java.net.MulticastSocket`.

For example, you can obtained a `FileChannel` as follows:

```
FileInputStream fis = new FileInputStream("in.dat");
FileChannel fc = fis.getChannel();
```

A `FileChannel` obtained from a `FileInputStream` is read-only; while a FileChannel obtained from a `FileOutputStream` is write-only. The Standard IO's stream is read/write into a byte-array buffer; while `FileChannel` uses a `ByteBuffer` object.

Example: Copying a file using `FileChannel`

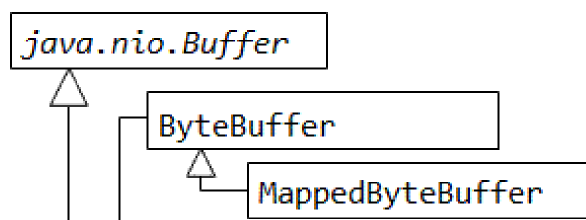
```
import java.io.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class FileChannelCopy {
```

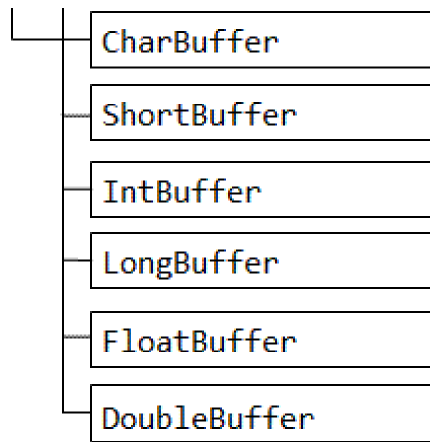
```
   public static void main(String[] args) {
      File fileIn;
      FileInputStream fis;
      FileOutputStream fos;
      FileChannel fic = null;
      FileChannel foc = null;
      ByteBuffer bytes;
      long startTime, elapsedTime;  // for speed benchmarking

      try {
         fileIn = new File("test-in.jpg");
         System.out.println("File size is " + fileIn.length() + " bytes");
         fis = new FileInputStream(fileIn);
         fos = new FileOutputStream("test-out.jpg");
         // Construct channels from streams
         fic = fis.getChannel();
         foc = fos.getChannel();
         bytes = ByteBuffer.allocate(4096);

         startTime = System.nanoTime();
         while (true) {
            bytes.clear();
            int bytesRead = fic.read(bytes);
            if (bytesRead <= 0) break;
            // flip the buffer which set the limit to current position, and position to 0.
            bytes.flip();
            foc.write(bytes);
         }
         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 (fic != null) fic.close();
            if (foc != null) foc.close();
         } catch (IOException ex) { ex.printStackTrace(); }
      }
   }
}
```

```
File size is 417455 bytes
Elapsed Time is 6.955283 msec
```

Remark: It is slower than the 3 msec reported using IO streams with a programmer-managed buffer of the same size (4 KB)?! [May need to look into it!]

Buffers (java.nio.Buffer)





These is a buffer for each primitive type. Buffer is similar an byte array, which represents a contiguous chunk of data with a fixed capcaity. The fundamental difference between an byte array used in Standard IO and Buffer is that array are always *non-direct*, whereas Buffer could be *direct*. This will be explained later.

A `Buffer` is allocated once and reused. A `Buffer` has a capacity, position, limit, and an optional mark:

- The capacity must be specified when the `Buffer` is constructed and cannot be changed. You can retrieve it via method `capacity()`.
- The position indicates where the next piece of data is to be read or written. You can retrieve the current position via method `position()` or change the current position via method `position(int newPosition)`.
- The `limit` specifies the current occupancy, i.e., the uppermost position that contains vaid data. You can retrieve the current limit via method `limit()` or set the `limit` via method `limit(int newLimit)`.
- The mark provide a positional marker. You can mark the current position via the method `mark()`.

Several methods are available to manipulate the `Buffer`:

- `clear()`: set the `position` to 0, `limit` to the `capacity`, and discard `mark`.
- `flip()`: set the `limit` to the current `position`, then `position` to 0, and discard `mark`.
- `reset()`: set the position to the previously-marked position.
- `rewind()`: set the position to 0, and discard mark.

`java.nio.MappedByteBuffer`

`MappedByteBuffer` is a so-called *direct* buffer that is managed by the OS, instead of the Java application. In other words, `MappedByteBuffer` can be used to wrap a region of OS buffer. Application can allocate different direct buffer to *view* the diffeent portion of the OS buffer.

[PENDING] New IO to be continued...

[PENDING] Network IO

REFERENCES & RESOURCES

- PENDING

Latest version tested: JDK 1.6
Last modified: August 29, 2008

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME