PER SCHOLAS

Introduction to Java Programming

Working with Byte Stream programs and Java NIO

Java IO

- Students should be able to understand:
  - Byte Streams
  - Java NIO
  - Writing Objects With Java NIO
  - Reading and Writing with Java NIO
  - Writing Binary Files with Java NIO
  - Reading Files with NIO

Java IO

Byte Streams

- A byte stream access the file byte by byte. Java programs use byte streams to perform input and output of 8-bit bytes. It is suitable for any kind of file, however not quite appropriate for text files. For example, if the file is using a unicode encoding and a character is represented with two bytes, the byte stream will treat these separately and you will need to do the conversion yourself. Byte oriented streams do not use any encoding scheme while Character oriented streams use character encoding scheme(UNICODE). All byte stream classes are descended fromInputStream and OutputStream .

Java IO

Byte Streams

```
 6 public class ByteStreamEx1 {
 7     public static void main(String[] args) {
 8         FileInputStream fis = null;
 9         FileOutputStream fos = null;
10         try{
11 fis = new FileInputStream("C:\\Users\\PSAdmin\\Documents\\sample.txt");
12 fos = new FileOutputStream("C:\\Users\\PSAdmin\\Documents\\sampleTo.txt");
13             int temp;
14             while ((temp = fis.read()) != -1) { // read byte by byte
15                 fos.write((byte) temp); // write byte by byte
16             }
17                 if (fis != null)
18                 fis.close();
19             if (fos != null)
20                 fos.close();
21         }
22         catch (Exception e){
23             System.out.println(e);
24         }
25     }
26 }
```

Java IO

Byte Streams

- Take time to read this article on your own time:
  - http://ecomputernotes.com/java/stream/byte-stream-classes

Java IO

Byte Streams

- Read this article on your own time:
  - http://ecomputernotes.com/java/stream/byte-stream-classes

Java NIO (New I/O)

Java NIO

- Java has provided a second I/O system called NIO (New I/O). Java NIO provides the different way of working with I/O than the standard I/O API's. It is an alternate I/O API for Java (from Java 1.4).
- It supports a buffer-oriented, channel based approach for I/O operations. With the introduction of JDK 7, the NIO system is expanded, providing the enhanced support for file system features and file-handling. Due to the capabilities supported by the NIO file classes, NIO is widely used in file handling.
- NIO was developed to allow Java programmers to implement high-speed I/O without using the custom native code. NIO moves the time-taking I/O activities like filling, namely and draining buffers, etc back into the operating system, thus allows for great increase in operational speed.

Java NIO (New I/O)

Java NIO

- **Channels and Buffers:** In standard I/O API the character streams and byte streams are used. In NIO we work with channels and buffers. Data is always written from a buffer to a channel and read from a channel to a buffer.
- **Selectors:** Java NIO provides the concept of "selectors". It is an object that can be used for monitoring the multiple channels for events like data arrived, connection opened etc. Therefore single thread can monitor the multiple channels for data.
- **Non-blocking I/O:** Java NIO provides the feature of Non-blocking I/O. Here the application returns immediately whatever the data available and application should have pooling mechanism to find out when more data is ready.

Java NIO (New I/O)

Channel and Buffers

- In Java NIO reading and writing are the fundamental process of I/O. Reading from channel: We can create a buffer and then ask a channel to read the data. Writing from channel: We can create a buffer, fill it with data and ask a channel to write the data.
- The core components used in the reading and writing operation are:
  - Channels
  - Buffers
  - Selectors
- In standard I/O API the character streams and byte streams are used. In NIO we work with channels and buffers. All the I/O in NIO is started with a channel. Data is always written from a buffer to a channel and read from a channel to a buffer.

Java NIO (New I/O)

Channel and Buffers

- **Data reading operation:**

- **Data writing operation:**

Java NIO (New I/O)

Channel and Buffers

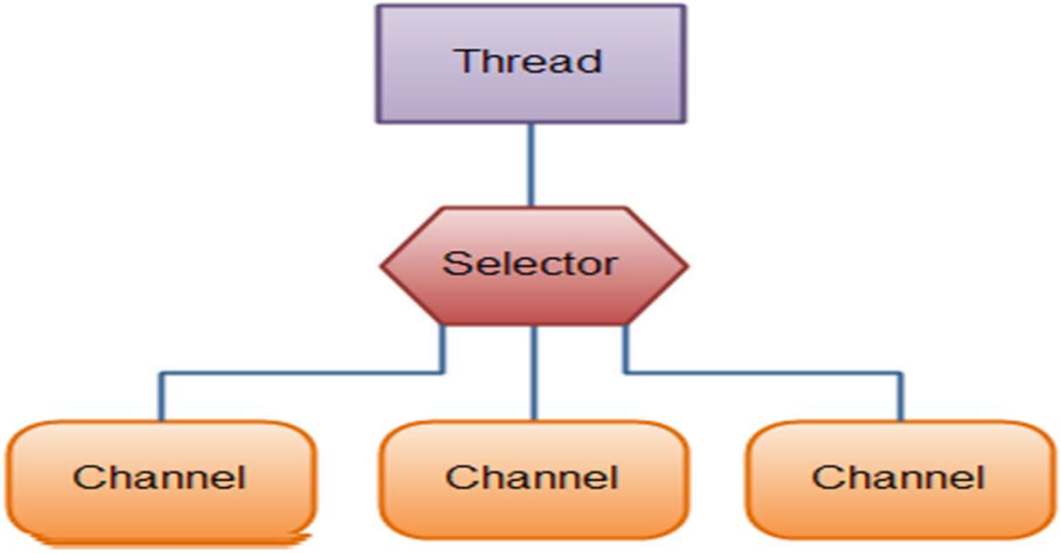
- In Java NIO the primary Channels used are given below:
  - DatagramChannel
  - SocketChannel
  - FileChannel
  - ServerSocketChannel
- In Java NIO the core Buffer used are given below:
  - CharBuffer
  - DoubleBuffer
  - IntBuffer
  - LongBuffer
  - ByteBuffer
  - ShortBuffer
  - FloatBuffer

The above buffer's cover the basic data types that we can send via I/O: characters, double, int, long, byte, short and float.

Java NIO (New I/O)

Channel and Buffers

- Java NIO provides the concept of "selectors". It is an object that can be used for monitoring the multiple channels for events like data arrived, connection opened etc. Therefore single thread can monitor the multiple channels for data.
- It is used if the application has many Channels (connections) open, but has low traffic on each connection. For example: In a chat server.
- Let's see the thread using a Selector to handle the 3 Channel's illustration shown below:



Java NIO (New I/O)

Java NIO Packages

Package	Purpose
java.nio	It is top-level package for NIO system. The various types of buffers are encapsulated by this NIO system.
java.nio.charset	It encapsulates the character sets and also supports encoders and decoders operation that convert characters to bytes and bytes to characters, respectively.
java.nio.charset.spi	It supports the service provider for character sets.
java.nio.channels	It support the channel, which are essentially open the I/O connections.
java.nio.channels.spi	It supports the service providers for channels.
java.nio.file	It provides the support for files.
java.nio.file.spi	It supports the service providers for file system.
java.nio.file.attribute	It provides the support for file attributes.

Java NIO (New I/O)

Channels Implementation

- In Java NIO the primary Channels used are given below:
  - FileChannel: The file channel is used for reading the data from the files. It's object can be created only by calling the getChannel() method. We cannot create FileChannel object directly.
  - Create the object of FileChannel
    - **FileInputStream fis = new FileInputStream("D:\\testin.txt");**
    - **ReadableByteChannel rbc = fis.getChannel();**
- DatagramChannel: The datagram channel can read and write the data over the network via UDP (User Datagram Protocol). It uses the factory methods for creating the new object.
  - The syntax used for opening the DatagramChannel:
    - **DatagramChannel ch = DatagramChannel.open();**
    - The syntax used for closing the DatagramChannel:
    - **DatagramChannel ch = DatagramChannel.close();**

Java NIO (New I/O)

Channels Implementation

```
10 public class NIOEx1 {
11     public static void main(String args[]) throws IOException {
12 FileInputStream input = new FileInputStream("sample.txt");
13         ReadableByteChannel source = input.getChannel();
14   FileOutputStream output = new FileOutputStream("sampleTo.txt");
15         WritableByteChannel destination = output.getChannel();
16         copyData(source, destination);
17         source.close();
18         destination.close();
19     }
20     private static void copyData(ReadableByteChannel src,
21             WritableByteChannel dest) throws IOException {
22         ByteBuffer buffer = ByteBuffer.allocateDirect(20 * 1024);
23         while (src.read(buffer) != -1) {
24             buffer.flip(); // The buffer is used to drained
25             // keep sure that buffer was fully drained
26             while (buffer.hasRemaining()) {
27                 dest.write(buffer);
28             }
29             buffer.clear();
30         }// Now the buffer is empty, ready for the filling
31     }
32 }
```

Java NIO (New I/O)

Buffers

- Buffers are defined inside java.nio package. It defines the core functionality which is common to all buffers: limit, capacity and current position.
- Java NIO buffers are used for interacting with NIO channels. It is the block of memory into which we can write data, which we can later be read again. The memory block is wrapped with a NIO buffer object, which provides easier methods to work with the memory block.
- Allocating a Buffer
  - For obtaining a buffer object we must first allocate a buffer. In every Buffer class an allocate() method is for allocating a buffer.
  - Let's see the example showing the allocation of ByteBuffer, with capacity of 28 bytes:
    - **ByteBuffer buf = ByteBuffer.allocate(28);**
  - allocation of CharBuffer, with space for 2048 characters:
    - **CharBuffer buf = CharBuffer.allocate(2048);**

Java NIO (New I/O)

Buffers

- Reading Data from a Buffer
  - There are two methods for reading the data from a Buffer:
  - Read the data from Buffer by using one of the get() method.
  - Read the data from Buffer into a Channel.
  - Let's see the example that read the data from Buffer using get() method:
    - **byte aByte = buf.get();**
  - Let's see the example that read the data from Buffer into a channel:
    - **int bytesWritten = inChannel.write(buf);**
- Writing Data to a Buffer
  - There are two methods for writing the data into a Buffer:
  - Write the data into Buffer by using one of the put() method.
  - Write the data from Channel into a Buffer.

Java NIO (New I/O)

Buffers

```
11 public class NIOEx2 {
12     public static void main(String[] args) {
13         Path file = null;
14         BufferedReader bufferedReader = null;
15         try {
16         file = Paths.get("sample.txt");
17         InputStream inputStream = Files.newInputStream(file);
18         bufferedReader = new BufferedReader(new InputStreamReader(inputStream));
19         System.out.println("Reading the first Line of testout.txt file: "
20                            + bufferedReader.readLine());
21         } catch (IOException e) {
22             e.printStackTrace();
23         } finally {
24             try {
25                 bufferedReader.close();
26             } catch (IOException ioe) {
27                 ioe.printStackTrace();
28             }
29         }
30     }
31 }
```

Java NIO (New I/O)

Reading Binary Files

- If you want to read a binary file, or a text file containing special characters, you need to use FileInputStream instead of FileReader. Instead of wrapping FileInputStream in a buffer, FileInputStream defines a method called read() that lets you fill a buffer with data, automatically reading just enough bytes to fill the buffer (or less if there aren't that many bytes left to read).

Java NIO (New I/O)

Reading Binary Files Example

```
7  public class BinaryFilesReadEx1 {
8      public static void main(String [] args) {
9          // The name of the file to open.
10         String fileName = "sample.txt";
11         try {
12             // Use this for reading the data.
13             byte[] buffer = new byte[1000];
14          FileInputStream inputStream = new FileInputStream(fileName);
15             // read fills buffer with data and returns
16             // the number of bytes read (which of course
17             // may be less than the buffer size, but
18             // it will never be more).
19             int total = 0;
20             int nRead = 0;
21             while((nRead = inputStream.read(buffer)) != -1) {
22                 // Convert to String so we can display it.
23                 // Of course you wouldn't want to do this with
24                 // a 'real' binary file.
25                 System.out.println(new String(buffer));
26                 total += nRead;
27             }
```

Java NIO (New I/O)

Reading Binary Files Example (Cont.)

```
32              catch(FileNotFoundException ex) {
33                  System.out.println(
34                      "Unable to open file '" +
35                      fileName + "'");
36              }
37              catch(IOException ex) {
38                  System.out.println(
39                      "Error reading file '"
40                      + fileName + "'");
41                  // Or we could just do this:
42                  // ex.printStackTrace();
43              }
44          }
45      }
```

Java NIO (New I/O)

Writing Binary Files Example

- To write a text file in Java, use FileWriter instead of FileReader, and BufferedOutputWriter instead of BufferedOutputReader.

```
 7 public class BinaryFilesWriteEx2 {
 8     public static void main(String [] args) {
 9         // The name of the file to open.
10         String fileName = "sample.txt";
11         try {
12             // Assume default encoding.
13             FileWriter fileWriter =
14                 new FileWriter(fileName);
15             // Always wrap FileWriter in BufferedWriter.
16             BufferedWriter bufferedWriter =
17                 new BufferedWriter(fileWriter);
18             // Note that write() does not automatically
19             // append a newline character.
20             bufferedWriter.write("Hello there,");
21             bufferedWriter.write(" here is some text.");
22             bufferedWriter.newLine();
23             bufferedWriter.write("We are writing");
24             bufferedWriter.write(" the text to the file.");
25
26             // Always close files.
27             bufferedWriter.close();
28         }
```

Java NIO (New I/O)

Writing Binary Files Example (Cont.)

```
29          catch(IOException ex) {
30              System.out.println(
31                  "Error writing to file '"
32                  + fileName + "'");
33              // Or we could just do this:
34              // ex.printStackTrace();
35          }
36      }
37  }
```