Bit Input/Output Stream utility classes for efficient data transfer

Considering the cost and duration of mobile data transmission (GPRS, UMTS, ...), it is cruical to optimize the content of the transmission. Any byte you waste adds to the mobile phone bill and the time the user has to wait while he's working with your application. Especially when transmitting data quite often or if there are large amounts of it, you should try to use only the number of bits that are really required. For example, if the variable you have to send to the server can only be in the range of 0 to 5, it would be sufficient to use 3 bit - if you transmit it using an integer, you'd need several bytes. When sending thousands of those numbers, the difference matters.
The following complete classes behave just like the other high level streams available in Java ME, making them very easy to integrate into your projects.

BitOutputStream

Using the BitOutputStream-class, you can calculate the required amount of bits for the range of numbers you want to transmit. Once you know this, you can use writeBits() - the first parameter has to contain the number you want to transmit, through the second parameter you have to specify how many bits should be used for it. Don't forget to use flush() when finished, as this will write the last byte even if you didn't fill all bits with useful data.

```
import java.io.IOException;
import java.io.OutputStream;

/**
 * The BitOutputStream allows writing individual bits to a
 * general Java OutputStream.
 * Like the various Stream-classes from Java, the BitOutputStream
 * has to be created based on another OutputStream. This class is able
 * to write a single bit to a stream (even though a byte has to be
 * filled until the data is flushed to the underlying output stream).
 * It is also able to write an integer value to the stream using
 * the specified number of bits.
 *
 * @author Andreas Jakl
 */
public class BitOutputStream {
 /**
  * The Java OutputStream that is used to write completed
  * bytes.
  */
 private OutputStream iOs;

 /**
  * The temponary buffer containing the individual bits
  * until a byte has been completed and can be commited
  * to the output stream.
  */
 private int iBuffer;

 /**
  * Counts how many bits have been cached up to now.
  */
 private int iBitCount;

 /**
  * Create a new bit output stream based on an
  * existing Java OutputSTream.
  * @param aOs the output stream this class should use.
  */
 public BitOutputStream(OutputStream aOs)
 {
  iOs = aOs;
 }

 /**
  * Write a single bit to the stream. It will only be flushed
  * to the underlying OutputStream when a byte has been
  * completed or when flush() manually.
  * @param aBit 1 if the bit should be set, 0 if not
  * @throws IOException
  */
 synchronized public void writeBit(int aBit) throws IOException
 {
  if (iOs == null)
   throw new IOException("Already closed");

  if (aBit != 0 && aBit != 1)
  {
   throw new IOException(aBit + " is not a bit");
  }

  iBuffer |= aBit << iBitCount;
  iBitCount++;

  if (iBitCount == 8)
  {
   flush();
  }
 }

 /**
  * Write the current cache to the stream and reset
  * the buffer.
  * @throws IOException
  */
 public void flush() throws IOException
 {
  if (iBitCount > 0)
  {
   iOs.write((byte) iBuffer);
   iBitCount = 0;
   iBuffer = 0;
  }
 }

 /**
```


(C) Copyright Nokia 2013. All rights reserved.

```
   * Flush the data and close the underlying output stream.
   * @throws IOException
   */
  public void close() throws IOException
  {
   flush();
   iOs.close();
   iOs = null;
  }

  /**
   * Write the specified number of bits from the int value
   * to the stream. Correspondig to the InputStream,
   * the bits are written starting at the highest bit
   * ( >> aNumberOfBits ), going down to the lowest bit ( >> 0 ).
   * @param aValue the int containing the bits that should
   * be written to the stream.
   * @param aNumBits how many bits of the integer should
   * be written to the stream.
   * @throws IOException
   */
  synchronized public void writeBits(final int aValue, final short aNumBits)
             throws IOException
  {
   for (int i = aNumBits - 1; i >= 0; i--)
   {
    writeBit((aValue >> i) & 0x01);
   }
  }

  /**
   * Calculate how many bits are needed to store the specified
   * value. Can be used to optimize data transfer.
   * @param aMaxValue the value that you want to test.
   * @return how many bits the specified value needs to be
   * stored.
   */
  public static int getRequiredNumOfBits(final int aMaxValue)
  {
   // 0 still requires 1 bit to write it.
   if (aMaxValue == 0)
    return 1;

   // Go from left to right and search for the first 1
   // 00011010
   //    |---- First 1
   int curBit;
   for (curBit = 31; curBit >= 0; curBit--)
   {
    if ((aMaxValue & (0x01 << curBit)) > 0)
    {
     // Found first bit that is not null - max. value
     break;
    }
   }
   return curBit + 1;

   // Real maximum value is everything filled with 1 from this point on.
   // 00011111
   //int maxVal = powOf2(curBit + 1) - 1;
  }

 }
```

BitInputStream

The BitInputStream-class is the complementary part of the BitOutputStream. It allows to read in a stream that was written by the output stream. Essentially, you will most of the times specify the number of bits to read and will get an integer containing the number that's made out of those numbers.

```
 import java.io.EOFException;
 import java.io.IOException;
 import java.io.InputStream;

 /**
  * The BitInputStream allows reading individual bits from a
  * general Java InputStream.
  * Like the various Stream-classes from Java, the BitInputStream
  * has to be created based on another Input stream. It provides
  * a function to read the next bit from the sream, as well as to read multiple
  * bits at once and write the resulting data into an integer value.
  *
  * @author Andreas Jakl
  */
 public class BitInputStream {
  /**
   * The Java InputStream this class is working on.
   */
  private InputStream iIs;

  /**
   * The buffer containing the currently processed
   * byte of the input stream.
   */
  private int iBuffer;

  /**
   * Next bit of the current byte value that the user will
   * get. If it's 8, the next bit will be read from the
   * next byte of the InputStream.
   */
  private int iNextBit = 8;

  /**
   * Create a new bit input stream based on an existing Java InputStream.
   * @param aIs the input stream this class should read the bits from.
   */
  public BitInputStream(InputStream aIs)
  {
   iIs = aIs;
  }

  /**
   * Read a specified number of bits and return them combined as
```


(C) Copyright Nokia 2013. All rights reserved.

```
   * an integer value. The bits are written to the integer
   * starting at the highest bit ( << aNumberOfBits ), going down
   * to the lowest bit ( << 0 )
   * @param aNumberOfBits defines how many bits to read from the stream.
   * @return integer value containing the bits read from the stream.
   * @throws IOException
   */
  synchronized public int readBits(final short aNumberOfBits)
            throws IOException
  {
   int value = 0;
   for (int i = aNumberOfBits - 1; i >= 0; i--)
   {
    value |= (readBit() << i);
   }
   return value;
  }

  /**
   * Read the next bit from the stream.
   * @return 0 if the bit is 0, 1 if the bit is 1.
   * @throws IOException
   */
  synchronized public int readBit() throws IOException
  {
   if (iIs == null)
    throw new IOException("Already closed");

   if (iNextBit == 8)
   {
    iBuffer = iIs.read();

    if (iBuffer == -1)
     throw new EOFException();

    iNextBit = 0;
   }

   int bit = iBuffer & (1 << iNextBit);
   iNextBit++;

   bit = (bit == 0) ? 0 : 1;

   return bit;
  }

  /**
   * Close the underlying input stream.
   * @throws IOException
   */
  public void close() throws IOException
  {
   iIs.close();
   iIs = null;
  }
 }
```


(C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.