BufferedInputStream for Java ME

```
/**
* A <code>BufferedInputStream</code> adds
* functionality to another input stream-namely, the ability to buffer the input and to
* support the mark and reset methods. When the BufferedInputStream is created, an internal buffer array is
* created. As bytes from the stream are read or skipped, the internal buffer is refilled
* as necessary from the contained input stream, many bytes at a time.
*The mark operation remembers a point in the input stream and the reset operation
* causes all the bytes read since the most
* recent <code>mark</code> operation to be
* reread before new bytes are taken from
* the contained input stream.
*
*/
import java.io.*;
public class BuffereadoInputStream extends InputStream{ //extends FilterInputStream {
//here resides the problem with inheritance in Java ME
//FilterInputStream is'nt a Java ME class, but we don't need it (I hope)
private static int defaultBufferSize = 2048;



/**
* The internal buffer array where the data is stored. When necessary,
* it may be replaced by another array of
* a different size.
*/
protected byte buf[];

/**
* The index one greater than the index of the last valid byte in
* the buffer.
* This value is always
* in the range <code>0</code> through <code>buf.length</code>;
* elements <code>buf[0]</code> through <code>buf[count-1]
* </code>contain buffered input data obtained
* from the underlying input stream.
*/
protected int count;

/**
* The current position in the buffer. This is the index of the next
* character to be read from the <code>buf</code> array.

*/
protected int pos;

/**
* The value of the <code>pos</code> field at the time the last
* <code>mark</code> method was called.
*/
protected int markpos = -1;

/**
* The maximum read ahead allowed after a call to the
* <code>mark</code> method before subsequent calls to the
* <code>reset</code> method fail.
*/
protected int marklimit;
//************************//
private InputStream padre; //this is new variable that we need;
//where we find "in", we put "padre", that's all...
//***********************//

private void ensureOpen() throws IOException {
//if (in == null)
if(padre==null)
throw new IOException("Stream closed");
}


public BuffereadoInputStream(InputStream in) {
this(in, defaultBufferSize);
}
```


(C) Copyright Nokia 2013. All rights reserved.

```
public BuffereadoInputStream(InputStream in, int size) {
//super(in);
padre=in;
if (size <= 0) {
throw new IllegalArgumentException("Buffer size <= 0");
}
buf = new byte[size];
}

private void fill() throws IOException {
if (markpos < 0)
pos = 0; /* no mark: throw away the buffer */
else if (pos >= buf.length) /* no room left in buffer */
if (markpos > 0) { /* can throw away early part of the buffer */
int sz = pos - markpos;
System.arraycopy(buf, markpos, buf, 0, sz);
pos = sz;
markpos = 0;
} else if (buf.length >= marklimit) {
markpos = -1; /* buffer got too big, invalidate mark */
pos = 0; /* drop buffer contents */
} else { /* grow buffer */
int nsz = pos * 2;
if (nsz > marklimit)
nsz = marklimit;
byte nbuf[] = new byte[nsz];
System.arraycopy(buf, 0, nbuf, 0, pos);
buf = nbuf;
}
count = pos;
//int n = in.read(buf, pos, buf.length - pos);
int n = padre.read(buf, pos, buf.length - pos);
if (n > 0)
count = n + pos;
}

public synchronized int read() throws IOException {
ensureOpen();
if (pos >= count) {
fill();
if (pos >= count)
return -1;
}
return buf[pos++] & 0xff;
}

/**
* Read characters into a portion of an array, reading from the underlying
* stream at most once if necessary.
*/
private int read1(byte[] b, int off, int len) throws IOException {
int avail = count - pos;
if (avail <= 0) {

if (len >= buf.length && markpos < 0) {
//return in.read(b, off, len);
return padre.read(b, off, len);
}
fill();
avail = count - pos;
if (avail <= 0) return -1;
}
int cnt = (avail < len) ? avail : len;
System.arraycopy(buf, pos, b, off, cnt);
pos += cnt;
return cnt;
}


public synchronized int read(byte b[], int off, int len)
throws IOException
{
ensureOpen();
if ((off | len | (off + len) | (b.length - (off + len))) < 0) {
throw new IndexOutOfBoundsException();
} else if (len == 0) {
return 0;
}
```


(C) Copyright Nokia 2013. All rights reserved.

```
int n = read1(b, off, len);
if (n <= 0) return n;
//while ((n < len) && (in.available() > 0)) {
while ((n < len) && (padre.available() > 0)) {
int n1 = read1(b, off + n, len - n);
if (n1 <= 0) break;
n += n1;
}
return n;
}

public synchronized long skip(long n) throws IOException {
ensureOpen();
if (n <= 0) {
return 0;
}
long avail = count - pos;

if (avail <= 0) {
// If no mark position set then don't keep in buffer
if (markpos <0)
//return in.skip(n);
return padre.skip(n);

// Fill in buffer to save bytes for reset
fill();
avail = count - pos;
if (avail <= 0)
return 0;
}

long skipped = (avail < n) ? avail : n;
pos += skipped;
return skipped;
}


public synchronized int available() throws IOException {
ensureOpen();
//return (count - pos) + in.available();
return (count - pos) + padre.available();
}


public synchronized void mark(int readlimit) {
marklimit = readlimit;
markpos = pos;
}


public synchronized void reset() throws IOException {
ensureOpen();
if (markpos < 0)
throw new IOException("Resetting to invalid mark");
pos = markpos;
}

public boolean markSupported() {
return true;
}


public void close() throws IOException {
//if (in == null)
if (padre == null)
return;
//in.close();
padre.close();
//in = null;
padre=null;
buf = null;
}
}
```


(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.