Utility class that provides methods to manipulate stream of data.

```
#region License and Copyright
/* ————————————————————————-
* Dotnet Commons IO
*
*
* This library is free software; you can redistribute it and/or modify it
* under the terms of the GNU Lesser General Public License as published by
* the Free Software Foundation; either version 2.1 of the License, or
* (at your option) any later version.
*
* This library is distributed in the hope that it will be useful, but
* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License
* for more details.
*
* You should have received a copy of the GNU Lesser General Public License
* along with this library; if not, write to the
*
* Free Software Foundation, Inc.,
* 59 Temple Place,
* Suite 330,
* Boston,
* MA 02111-1307
```

```
* USA
*
* ---------------------------
*/
#endregion

using System;
using System.IO;
using System.Text;

namespace Dotnet.Commons.IO
{
///

/// Utility class that provides methods to manipulate stream of data.
///
///
/// This class is ported from Jakarta Commons IO org.apache.commons.io.CopyUtils class.
///
/// This class also contains code taken from an article written by Jon Skeet. The
/// article can be found here:
/// http://www.developerfusion.co.uk/show/4696/
///
public sealed class StreamUtils
{
///

The name says it all.
private const int DEFAULT_BUFFER_SIZE = 1024 * 4;

private StreamUtils(){}

public static void Copy(byte[] input, byte[] output, long outputOffset)
{
if (input.Length == 0) return;
```

```
for (int i=0; i 0)
{
len = input.Read(buffer, offset, DEFAULT_BUFFER_SIZE);
output.Write(buffer, 0, len);
bytesRead += len;
}

return bytesRead;
}

// ————————————————————-
// Reader -> Writer
// ————————————————————-

///

Copy chars from a  to a .
/// the  to read from
///  /// the  to write to
///  ///  the number of characters copied
///
/// if an I/O problem occurs
public  static  int  Copy(StreamReader  inputStreamReader,
StreamWriter outputStreamWriter)
{
char[] buffer = new char[DEFAULT_BUFFER_SIZE];
int count = 0;
int len = DEFAULT_BUFFER_SIZE;

while (len > 0)
{
len = inputStreamReader.Read(buffer, 0, DEFAULT_BUFFER_SIZE);
outputStreamWriter.Write(buffer, 0, len);
count += len;
}
return count;
}

// ————————————————————-
```

```
// InputStream -> Writer
// ----------------------

///

Copy and convert bytes from an Input  to chars on a
/// .
/// The platform's default encoding is used for the byte-to-
char conversion.
///
/// the Input  to read from
///  /// the to write to
///  /// if an I/O problem occurs
public static int Copy(Stream inputStream, StreamWriter
outputStreamWriter)
{
StreamReader inputStreamReader = new StreamReader(inputStream,
System.Text.Encoding.Default);
return Copy(inputStreamReader, outputStreamWriter);
}

///

/// Copy and convert bytes from an Input  to chars on a
/// , using the specified encoding.
///
///  ///  /// The name of a supported character encoding. See
the
/// IANA
/// Charset Registry and MSDN: Encoding class
/// for a list of valid encoding types. /// an I/O problem
occurs
public static int Copy(Stream inputStream, StreamWriter
outputWriter, String encoding)
{
Encoding encode = Encoding.Default;

try
{
```

```
encode = Encoding.GetEncoding(encoding);
}
catch
{
encode = Encoding.Default;
}
StreamReader inputStreamReader = new StreamReader(inputStream,
encode);
return Copy(inputStreamReader, outputWriter);
}

// ————————————————————-
// Reader -> OutputStream
// ————————————————————-

///

/// Serialize chars from a  to bytes on an
/// Output , and flush the Output .
///
/// the  to read from /// the  to write to /// an I/O problem
occurs
public static void Copy(StreamReader inputReader, Stream
output)
{
StreamWriter outputWriter = new StreamWriter(output,
System.Text.Encoding.Default);
Copy(inputReader, outputWriter);

outputWriter.Flush();
}

// ————————————————————-
// String -> OutputStream
// ————————————————————-
///

Serialize chars from a  to bytes on an
/// Output , and
```

```
/// flush the Output .
///
/// the  to read from
///  /// the Output  to write to
///  /// an I/O problem occurs
public static void Copy(String input, Stream output)
{
byte[] inputByteArray = new ASCIIEncoding().GetBytes(input);
StreamWriter  outWriter  =  new  StreamWriter(output,
System.Text.Encoding.Default);
Copy(inputByteArray, outWriter);

outWriter.Flush();
}

// ————————————————————-
// String -> Writer
// ————————————————————-

///

Copy chars from a  to a .
/// the  to read from
///  /// the  to write to
///  /// an I/O problem occurs
public static void Copy(String input, StreamWriter output)
{
output.Write(input);
}

///

/// Copy the exact number of bytes from the source  to a
/// target .
///
/// Source  to copy from /// Target  to copy to /// number of
bytes to copy /// if the source stream does not have enough
data.
public static void CopyExact(Stream source, Stream target, int
```

```
len)
{
byte[] buffer = new byte[DEFAULT_BUFFER_SIZE];
int bytesRead = 0;

while (bytesRead < len) { int sizeNeeded = Math.Min(buffer.Length, len - bytesRead); int readSize = source.Read(buffer, 0, sizeNeeded); if (readSize <= 0) throw new IOException(String.Format("Underlying stream does not have enough data. Read {0} bytes, but {1} needed", readSize, sizeNeeded)); target.Write(buffer, 0, readSize); bytesRead += readSize; } } ///
/// Reads data into a complete array, throwing an EndOfStreamException
/// if the stream runs out of data first, or if an IOException
/// naturally occurs.
///

/// The stream to read data from /// The array to read bytes into. The array
/// will be completely filled from the stream, so an appropriate
/// size must be given. public static void ReadIntoByteArray (Stream stream, byte[] byteArray)
{
int offset=0;
int remaining = byteArray.Length;
stream.Position = 0;
while (remaining > 0)
{
int read = stream.Read(byteArray, offset, remaining);
if (read <= 0) throw new EndOfStreamException (String.Format("End of stream reached with {0} bytes left to read", remaining)); remaining -= read; offset += read; } } ///
/// Reads data from the beginning of a stream until the end is reached. The
/// data is returned as a byte array.
```

```
///

/// The stream to read data from /// thrown if any of the underlying IO calls fail
/// Use this method if you don't know the length of the stream in advance
/// (for instance a network stream) and just want to read the whole lot into a buffer.
///  /// Note:
/// This method of reading the stream is not terribly efficient.
///  ///
public static byte[] GetBytes(Stream stream)
{
if (stream is MemoryStream)
return ((MemoryStream)stream).ToArray();

byte[] byteArray = new byte[DEFAULT_BUFFER_SIZE];
using (MemoryStream ms = new MemoryStream())
{
stream.Position = 0;
while (true)
{
int readLen = stream.Read (byteArray, 0, byteArray.Length);
if (readLen <= 0) return ms.ToArray(); ms.Write (byteArray, 0, readLen); } } } ///
/// Reads data from a stream until the end is reached. The
/// data is returned as a byte array.
///

/// The stream to read data from /// The initial buffer length. If the length is &lt; 1,
/// then the default value of  will be used.
///  /// thrown if any of the underlying IO calls fail
/// Use this method to get the data if you know the expected length of data to start with.
public static byte[] GetBytes (Stream stream, long initialLength)
```

```
{
// If we've been passed an unhelpful initial length, just
// use 32K.
if (initialLength < 1) initialLength = Int16.MaxValue; byte[]
buffer = new byte[initialLength]; int read=0; int chunk; while
( (chunk = stream.Read(buffer, read, buffer.Length-read)) > 0)
{
read += chunk;

// If we've reached the end of our buffer, check to see if
there's
// any more information
if (read == buffer.Length)
{
int nextByte = stream.ReadByte();

// End of stream? If so, we're done
if (nextByte==-1)
{
return buffer;
}

// Nope. Resize the buffer, put in the byte we've just
// read, and continue
byte[] newBuffer = new byte[buffer.Length*2];
Array.Copy(buffer, newBuffer, buffer.Length);
newBuffer[read]=(byte)nextByte;
buffer = newBuffer;
read++;
}
}
// Buffer is now too big. Shrink it.
byte[] ret = new byte[read];
Array.Copy(buffer, ret, read);
return ret;
}

///
```

```
/// Return an ASCII string from a stream of data
///
///  ///
public static string GetAsciiString(Stream stream)
{
ASCIIEncoding encoding = new ASCIIEncoding();
return GetString(stream, encoding);
}

///

/// Return an UTF8 encoded string from a stream of data
///
///  ///
public static string GetString(Stream stream)
{
UTF8Encoding encoding = new UTF8Encoding();
return GetString(stream, encoding);
}

///

/// Return a string from a stream. The string is returned with
/// the encoding provided.
///
///  ///  ///
public static string GetString(Stream stream, Encoding
encoding)
{

if (stream == null)
return string.Empty;

byte[] bytes = new byte[stream.Length];

if (stream is MemoryStream)
bytes = ((MemoryStream)stream).GetBuffer();
else
ReadIntoByteArray(stream, bytes);
```

```
return encoding.GetString(bytes);
}

///

/// Reads the specified number of bytes from any position in a source stream into a
/// specific byte array in a specific start index position. The byte
/// array must have the necessary size to read the portion of the stream required.
///
/// Source stream to read from /// Target byte array to write to /// offset index in the target /// offset position in the stream /// number of bytes to read in the stream /// thrown if the target byte array is too small to stored
/// the required number of bytes read from the stream.
public static void ReadExact(Stream source,
byte[] target,
int sourceOffset,
int targetOffset,
int bytesToRead)
{

if (targetOffset + bytesToRead > target.Length)
throw new ArgumentException(“target array to small”);

int bytesRead = 0;
source.Seek(sourceOffset, SeekOrigin.Begin);
while (bytesRead < bytesToRead) { // need more data int sizeNeeded = Math.Min(DEFAULT_BUFFER_SIZE, bytesToRead - bytesRead); // read either the whole buffer length or // the remaining # of bytes: bytesToRead - sizeNeeded int readSize = source.Read(target, (targetOffset + bytesRead), sizeNeeded); if (readSize <= 0) throw new IOException(String.Format("Underlying stream does not have enough data. Read {0} bytes, but {1} needed", readSize, sizeNeeded)); bytesRead += readSize; } } ///
/// Read a partial segment of a stream, starting from an
```

```
offset position.
///

/// Source stream to read from /// the starting offset position in the stream. Set to 0 if the stream is to be read from the beginning. /// number of bytes to read /// return partial segment as an array of bytes.
public static byte[] ReadPartial(Stream source,
long sourceOffset,
long bytesToRead)
{
long sizeDiff = source.Length – sourceOffset;
if (bytesToRead > sizeDiff)
throw new ArgumentException(“Bytes required exceeds what is available in stream”);

byte[] target = new byte[bytesToRead];

long bytesRead = 0;
source.Seek(sourceOffset, SeekOrigin.Begin);

while (bytesRead < bytesToRead) { // need more data int sizeNeeded = (int)Math.Min((long)DEFAULT_BUFFER_SIZE, bytesToRead - bytesRead); // read either the whole buffer length or // the remaining # of bytes: bytesToRead - sizeNeeded int readSize = source.Read(target, 0, sizeNeeded); if (readSize <= 0) throw new IOException(String.Format("Underlying stream does not have enough data. Read {0} bytes, but {1} needed", readSize, sizeNeeded)); bytesRead += readSize; } return target; } ///
/// Read a stream (like a file or HttpWebRequest) and write to another stream
///

/// the stream to read ///  the stream to write to
[Obsolete(“See StreamUtils.Copy”)]
public static void ReadWriteStream(Stream readStream, Stream writeStream)
```

```
{
Byte[] buffer = new Byte[DEFAULT_BUFFER_SIZE];
int bytesRead = readStream.Read(buffer, 0, DEFAULT_BUFFER_SIZE);
// write the required bytes
while (bytesRead > 0)
{
writeStream.Write(buffer, 0, bytesRead);
bytesRead = readStream.Read(buffer, 0, DEFAULT_BUFFER_SIZE);
}
readStream.Close();
writeStream.Close();
}

///

/// Try to skip bytes in the input stream and return the actual number of bytes skipped.
///
/// Input stream that will be used to skip the bytes /// Number of bytes to be skipped /// Actual number of bytes skipped
public static int Skip(Stream stream, int skipBytes)
{
long oldPosition = stream.Position;
long result = stream.Seek(skipBytes, SeekOrigin.Current) - oldPosition;
return (int)result;
}

///

/// Skips a given number of characters into a given Stream.
///
/// The stream in which the skips are done. /// The number of caracters to skip. /// The number of characters skipped.
public static long Skip(StreamReader stream, long number)
{
long skippedBytes = 0;
```

```
for (long index = 0; index < number; index++) { stream.Read(); skippedBytes++; } return skippedBytes; } ///
/// Skips a given number of characters into a given StringReader.
///

/// The StringReader in which the skips are done. /// The number of caracters to skip. /// The number of characters skipped.
public static long Skip(StringReader strReader, long number)
{
long skippedBytes = 0;
for (long index = 0; index < number; index++) { strReader.Read(); skippedBytes++; } return skippedBytes; } ///
/// Converts a string to an array of bytes
///

/// The string to be converted /// The new array of bytes
public static byte[] ToByteArray(String sourceString)
{
return System.Text.UTF8Encoding.UTF8.GetBytes(sourceString);
}

///

/// Converts a array of object-type instances to a byte-type array.
///
/// Array to convert. /// An array of byte type elements.
public static byte[] ToByteArray(Object[] tempObjectArray)
{
byte[] byteArray = null;
if (tempObjectArray != null)
{
byteArray = new byte[tempObjectArray.Length];
for (int index = 0; index < tempObjectArray.Length; index++) byteArray[index] = (byte)tempObjectArray[index]; } return byteArray; } } } [/csharp]
```