This is the Java 8 version of BufferedReader which extends Reader. You will note the decorator pattern, i.e., BufferedReader is-a Reader and BufferedReader has-a Reader (in the case, the Reader that it has is a FileReader).

BufferedReader uses its FileReader in three ways:

- **in.read()   to use the FileReader to read a character**
- **in.ready()   to check to see whether this stream is ready to be read.**
- **in.close()   to close the FileReader**

BufferedReader has, as an instance variable, "cb" (character buffer) which is a char array.

```
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Spliterator;
import java.util.Spliterators;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;

/**
 * Reads text from a character-input stream, buffering characters so as to
 * provide for the efficient reading of characters, arrays, and lines.
 *
 * <p> The buffer size may be specified, or the default size may be used.  The
 * default is large enough for most purposes.
 *
 * <p> In general, each read request made of a Reader causes a corresponding
 * read request to be made of the underlying character or byte stream.  It is
 * therefore advisable to wrap a BufferedReader around any Reader whose read()
 * operations may be costly, such as FileReaders and InputStreamReaders.  For
 * example,
 *
 * <pre>
 * BufferedReader in
 *   = new BufferedReader(new FileReader("foo.in"));
 * </pre>
 *
 * will buffer the input from the specified file.  Without buffering, each
 * invocation of read() or readLine() could cause bytes to be read from the
 * file, converted into characters, and then returned, which can be very
 * inefficient.
 *
 * <p> Programs that use DataInputStreams for textual input can be localized by
 * replacing each DataInputStream with an appropriate BufferedReader.
 *
 * @see FileReader
 * @see InputStreamReader
 * @see java.nio.file.Files#newBufferedReader
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class BufferedReader extends Reader {

    private Reader in;

    private char cb[];
    private int nChars, nextChar;

    private static final int INVALIDATED = -2;
    private static final int UNMARKED = -1;
    private int markedChar = UNMARKED;
    private int readAheadLimit = 0; /* Valid only when markedChar > 0 */

    /** If the next character is a line feed, skip it */
    private boolean skipLF = false;

    /** The skipLF flag when the mark was set */
    private boolean markedSkipLF = false;

    private static int defaultCharBufferSize = 8192;
    private static int defaultExpectedLineLength = 80;

    /**
     * Creates a buffering character-input stream that uses an input buffer of
     * the specified size.
     *
     * @param  in   A Reader
     * @param  sz   Input-buffer size
     *
     * @exception  IllegalArgumentException  If {@code sz <= 0}
     */
    public BufferedReader(Reader in, int sz) {
        super(in);
        if (sz <= 0)
```

```
            throw new IllegalArgumentException("Buffer size <= 0");
        this.in = in;
        cb = new char[sz];
        nextChar = nChars = 0;
    }

    /**
     * Creates a buffering character-input stream that uses a default-sized
     * input buffer.
     *
     * @param  in   A Reader
     */
    public BufferedReader(Reader in) {
        this(in, defaultCharBufferSize);
    }

    /** Checks to make sure that the stream has not been closed */
    private void ensureOpen() throws IOException {
        if (in == null)
            throw new IOException("Stream closed");
    }

    /**
     * Fills the input buffer, taking the mark into account if it is valid.
     */
    private void fill() throws IOException {
        int dst;
        if (markedChar <= UNMARKED) {
            /* No mark */
            dst = 0;
        } else {
            /* Marked */
            int delta = nextChar - markedChar;
            if (delta >= readAheadLimit) {
                /* Gone past read-ahead limit: Invalidate mark */
                markedChar = INVALIDATED;
                readAheadLimit = 0;
                dst = 0;
            } else {
                if (readAheadLimit <= cb.length) {
                    /* Shuffle in the current buffer */
                    System.arraycopy(cb, markedChar, cb, 0, delta);
                    markedChar = 0;
                    dst = delta;
                } else {
                    /* Reallocate buffer to accommodate read-ahead limit */
                    char ncb[] = new char[readAheadLimit];
                    System.arraycopy(cb, markedChar, ncb, 0, delta);
                    cb = ncb;
                    markedChar = 0;
                    dst = delta;
                }
                nextChar = nChars = delta;
            }
        }

        int n;
        do {
            n = in.read(cb, dst, cb.length - dst);
        } while (n == 0);
        if (n > 0) {
            nChars = dst + n;
            nextChar = dst;
        }
    }

    /**
     * Reads a single character.
     *
     * @return The character read, as an integer in the range
     *         0 to 65535 (<tt>0x00-0xffff</tt>), or -1 if the
     *         end of the stream has been reached
     * @exception  IOException  If an I/O error occurs
     */
    public int read() throws IOException {
        synchronized (lock) {
            ensureOpen();
            for (;;) {
                if (nextChar >= nChars) {
                    fill();
                    if (nextChar >= nChars)
                        return -1;
                }
                if (skipLF) {
                    skipLF = false;
                    if (cb[nextChar] == '\n') {
                        nextChar++;
                        continue;
                    }
                }
                return cb[nextChar++];
            }
        }
    }

    /**
     * Reads characters into a portion of an array, reading from the underlying
     * stream if necessary.
     */
    private int read1(char[] cbuf, int off, int len) throws IOException {
        if (nextChar >= nChars) {
            /* If the requested length is at least as large as the buffer, and
               if there is no mark/reset activity, and if line feeds are not
               being skipped, do not bother to copy the characters into the
               local buffer.  In this way buffered streams will cascade
               harmlessly. */
            if (len >= cb.length && markedChar <= UNMARKED && !skipLF) {
                return in.read(cbuf, off, len);
            }
            fill();
        }
        if (nextChar >= nChars) return -1;
        if (skipLF) {
            skipLF = false;
            if (cb[nextChar] == '\n') {
                nextChar++;
                if (nextChar >= nChars)
                    fill();
                if (nextChar >= nChars)
                    return -1;
            }
        }
        int n = Math.min(len, nChars - nextChar);
        System.arraycopy(cb, nextChar, cbuf, off, n);
        nextChar += n;
        return n;
    }

    /**
     * Reads characters into a portion of an array.
     *
     * <p> This method implements the general contract of the corresponding
     * <code>{@link Reader#read(char[], int, int) read}</code> method of the
     * <code>{@link Reader}</code> class.  As an additional convenience, it
     * attempts to read as many characters as possible by repeatedly invoking
     * the <code>read</code> method of the underlying stream.  This iterated
     * <code>read</code> continues until one of the following conditions becomes
     * true: <ul>
     *
     *   <li> The specified number of characters have been read,
     *
     *   <li> The <code>read</code> method of the underlying stream returns
     *   <code>-1</code>, indicating end-of-file, or
     *
     *   <li> The <code>ready</code> method of the underlying stream
     *   returns <code>false</code>, indicating that further input requests
     *   would block.
     *
     * </ul> If the first <code>read</code> on the underlying stream returns
     * <code>-1</code> to indicate end-of-file then this method returns
     * <code>-1</code>.  Otherwise this method returns the number of characters
     * actually read.
     *
     * <p> Subclasses of this class are encouraged, but not required, to
     * attempt to read as many characters as possible in the same fashion.
     *
     * <p> Ordinarily this method takes characters from this stream's character
     * buffer, filling it from the underlying stream as necessary.  If,
     * however, the buffer is empty, the mark is not valid, and the requested
     * length is at least as large as the buffer, then this method will read
     * characters directly from the underlying stream into the given array.
     * Thus redundant <code>BufferedReader</code>s will not copy data
     * unnecessarily.
     *
     * @param      cbuf  Destination buffer
     * @param      off   Offset at which to start storing characters
     * @param      len   Maximum number of characters to read
     *
     * @return     The number of characters read, or -1 if the end of the
     *             stream has been reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read(char cbuf[], int off, int len) throws IOException {
        synchronized (lock) {
            ensureOpen();
            if ((off < 0) || (off > cbuf.length) || (len < 0) ||
                ((off + len) > cbuf.length) || ((off + len) < 0)) {
                throw new IndexOutOfBoundsException();
            } else if (len == 0) {
                return 0;
            }

            int n = read1(cbuf, off, len);
            if (n <= 0) return n;
            while ((n < len) && in.ready()) {
                int n1 = read1(cbuf, off + n, len - n);
                if (n1 <= 0) break;
                n += n1;
            }
            return n;
        }
    }

    /**
     * Reads a line of text.  A line is considered to be terminated by any one
     * of a line feed ('\n'), a carriage return ('\r'), or a carriage return
     * followed immediately by a linefeed.
     *
     * @param      ignoreLF  If true, the next '\n' will be skipped
     *
     * @return     A String containing the contents of the line, not including
     *             any line-termination characters, or null if the end of the
     *             stream has been reached
     *
     * @see        java.io.LineNumberReader#readLine()
     *
     * @exception  IOException  If an I/O error occurs
     */
    String readLine(boolean ignoreLF) throws IOException {
        StringBuffer s = null;
        int startChar;

        synchronized (lock) {
            ensureOpen();
            boolean omitLF = ignoreLF || skipLF;

        bufferLoop:
            for (;;) {

                if (nextChar >= nChars)
                    fill();
                if (nextChar >= nChars) { /* EOF */
                    if (s != null && s.length() > 0)
                        return s.toString();
                    else
                        return null;
                }
                boolean eol = false;
                char c = 0;
                int i;

                /* Skip a leftover '\n', if necessary */
                if (omitLF && (cb[nextChar] == '\n'))
                    nextChar++;
                skipLF = false;
                omitLF = false;

            charLoop:
                for (i = nextChar; i < nChars; i++) {
                    c = cb[i];
                    if ((c == '\n') || (c == '\r')) {
                        eol = true;
                        break charLoop;
                    }
                }

                startChar = nextChar;
                nextChar = i;

                if (eol) {
                    String str;
                    if (s == null) {
                        str = new String(cb, startChar, i - startChar);
                    } else {
                        s.append(cb, startChar, i - startChar);
                        str = s.toString();
                    }
                    nextChar++;
                    if (c == '\r') {
                        skipLF = true;
                    }
                    return str;
                }

                if (s == null)
                    s = new StringBuffer(defaultExpectedLineLength);
                s.append(cb, startChar, i - startChar);
            }
        }
    }

    /**
     * Reads a line of text.  A line is considered to be terminated by any one
     * of a line feed ('\n'), a carriage return ('\r'), or a carriage return
     * followed immediately by a linefeed.
     *
     * @return     A String containing the contents of the line, not including
     *             any line-termination characters, or null if the end of the
     *             stream has been reached
     *
     * @exception  IOException  If an I/O error occurs
     *
     * @see java.nio.file.Files#readAllLines
     */
```

```
    public String readLine() throws IOException {
        return readLine(false);
    }

    /**
     * Skips characters.
     *
     * @param  n  The number of characters to skip
     *
     * @return    The number of characters actually skipped
     *
     * @exception  IllegalArgumentException  If <code>n</code> is negative.
     * @exception  IOException  If an I/O error occurs
     */
    public long skip(long n) throws IOException {
        if (n < 0L) {
            throw new IllegalArgumentException("skip value is negative");
        }
        synchronized (lock) {
            ensureOpen();
            long r = n;
            while (r > 0) {
                if (nextChar >= nChars)
                    fill();
                if (nextChar >= nChars) /* EOF */
                    break;
                if (skipLF) {
                    skipLF = false;
                    if (cb[nextChar] == '\n') {
                        nextChar++;
                    }
                }
                long d = nChars - nextChar;
                if (r <= d) {
                    nextChar += r;
                    r = 0;
                    break;
                }
                else {
                    r -= d;
                    nextChar = nChars;
                }
            }
            return n - r;
        }
    }

    /**
     * Tells whether this stream is ready to be read.  A buffered character
     * stream is ready if the buffer is not empty, or if the underlying
     * character stream is ready.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public boolean ready() throws IOException {
        synchronized (lock) {
            ensureOpen();
            /*
             * If newline needs to be skipped and the next char to be read
             * is a newline character, then just skip it right away.
             */
            if (skipLF) {
                /* Note that in.ready() will return true if and only if the next
                 * read on the stream will not block.
                 */
                if (nextChar >= nChars && in.ready()) {
                    fill();
                }
                if (nextChar < nChars) {
                    if (cb[nextChar] == '\n')
                        nextChar++;
                    skipLF = false;
                }
            }
            return (nextChar < nChars) || in.ready();
        }
    }

    /**
     * Tells whether this stream supports the mark() operation, which it does.
     */
    public boolean markSupported() {
        return true;
    }

    /**
     * Marks the present position in the stream.  Subsequent calls to reset()
     * will attempt to reposition the stream to this point.
     *
     * @param readAheadLimit   Limit on the number of characters that may be
     *                         read while still preserving the mark. An attempt
     *                         to reset the stream after reading characters
     *                         up to this limit or beyond may fail.
     *                         A limit value larger than the size of the input
     *                         buffer will cause a new buffer to be allocated
     *                         whose size is no smaller than limit.
     *                         Therefore large values should be used with care.
     *
     * @exception  IllegalArgumentException  If {@code readAheadLimit < 0}
     * @exception  IOException  If an I/O error occurs
     */
    public void mark(int readAheadLimit) throws IOException {
        if (readAheadLimit < 0) {
            throw new IllegalArgumentException("Read-ahead limit < 0");
        }
        synchronized (lock) {
            ensureOpen();
            this.readAheadLimit = readAheadLimit;
            markedChar = nextChar;
            markedSkipLF = skipLF;
        }
    }
```

```
/**
 * Resets the stream to the most recent mark.
 *
 * @exception  IOException  If the stream has never been marked,
 *                          or if the mark has been invalidated
 */
public void reset() throws IOException {
    synchronized (lock) {
        ensureOpen();
        if (markedChar < 0)
            throw new IOException((markedChar == INVALIDATED)
                                  ? "Mark invalid"
                                  : "Stream not marked");
        nextChar = markedChar;
        skipLF = markedSkipLF;
    }
}

public void close() throws IOException {
    synchronized (lock) {
        if (in == null)
            return;
        try {
            in.close();
        } finally {
            in = null;
            cb = null;
        }
    }
}

/**
 * Returns a {@code Stream}, the elements of which are lines read from
 * this {@code BufferedReader}.  The {@link Stream} is lazily populated,
 * i.e., read only occurs during the
 * <a href="../util/stream/package-summary.html#StreamOps">terminal
 * stream operation</a>.
 *
 * <p> The reader must not be operated on during the execution of the
 * terminal stream operation. Otherwise, the result of the terminal stream
 * operation is undefined.
 *
 * <p> After execution of the terminal stream operation there are no
 * guarantees that the reader will be at a specific position from which to
 * read the next character or line.
 *
 * <p> If an {@link IOException} is thrown when accessing the underlying
 * {@code BufferedReader}, it is wrapped in an {@link
 * UncheckedIOException} which will be thrown from the {@code Stream}
 * method that caused the read to take place. This method will return a
 * Stream if invoked on a BufferedReader that is closed. Any operation on
 * that stream that requires reading from the BufferedReader after it is
 * closed, will cause an UncheckedIOException to be thrown.
 *
 * @return a {@code Stream<String>} providing the lines of text
 *         described by this {@code BufferedReader}
 *
```

```
     * @since 1.8
     */
    public Stream<String> lines() {
        Iterator<String> iter = new Iterator<String>() {
            String nextLine = null;

            @Override
            public boolean hasNext() {
                if (nextLine != null) {
                    return true;
                } else {
                    try {
                        nextLine = readLine();
                        return (nextLine != null);
                    } catch (IOException e) {
                        throw new UncheckedIOException(e);
                    }
                }
            }

            @Override
            public String next() {
                if (nextLine != null || hasNext()) {
                    String line = nextLine;
                    nextLine = null;
                    return line;
                } else {
                    throw new NoSuchElementException();
                }
            }
        };
        return StreamSupport.stream(Spliterators.spliteratorUnknownSize(
                iter, Spliterator.ORDERED | Spliterator.NONNULL), false);
    }
}
```

This is the Java 8 version of Reader -- an abstract class. Remember that BufferedReader contains a Reader. You will note the decorator pattern, i.e., BufferedReader is-a Reader and BufferedReader has-a Reader (in the case, the Reader that it has is a FileReader).

Note: When BufferedReader calls the read() method of its file reader, it is calling the overloaded version of read() -- highlighted in blue below -- that has more parameters.

```
/*
 * Copyright (c) 1996, 2012, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Abstract class for reading character streams.  The only methods that a
 * subclass must implement are read(char[], int, int) and close().  Most
 * subclasses, however, will override some of the methods defined here in order
 * to provide higher efficiency, additional functionality, or both.
 *
 *
 * @see BufferedReader
 * @see   LineNumberReader
 * @see CharArrayReader
 * @see InputStreamReader
 * @see   FileReader
 * @see FilterReader
 * @see   PushbackReader
 * @see PipedReader
 * @see StringReader
 * @see Writer
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public abstract class Reader implements Readable, Closeable {

    /**
     * The object used to synchronize operations on this stream.  For
     * efficiency, a character-stream object may use an object other than
     * itself to protect critical sections.  A subclass should therefore use
     * the object in this field rather than <tt>this</tt> or a synchronized
     * method.
     */
    protected Object lock;

    /**
     * Creates a new character-stream reader whose critical sections will
     * synchronize on the reader itself.
     */
    protected Reader() {
        this.lock = this;
    }

    /**
```

```
     * Creates a new character-stream reader whose critical sections will
     * synchronize on the given object.
     *
     * @param lock  The Object to synchronize on.
     */
    protected Reader(Object lock) {
        if (lock == null) {
            throw new NullPointerException();
        }
        this.lock = lock;
    }

    /**
     * Attempts to read characters into the specified character buffer.
     * The buffer is used as a repository of characters as-is: the only
     * changes made are the results of a put operation. No flipping or
     * rewinding of the buffer is performed.
     *
     * @param target the buffer to read characters into
     * @return The number of characters added to the buffer, or
     *         -1 if this source of characters is at its end
     * @throws IOException if an I/O error occurs
     * @throws NullPointerException if target is null
     * @throws java.nio.ReadOnlyBufferException if target is a read only buffer
     * @since 1.5
     */
    public int read(java.nio.CharBuffer target) throws IOException {
        int len = target.remaining();
        char[] cbuf = new char[len];
        int n = read(cbuf, 0, len);
        if (n > 0)
            target.put(cbuf, 0, n);
        return n;
    }

    /**
     * Reads a single character.  This method will block until a character is
     * available, an I/O error occurs, or the end of the stream is reached.
     *
     * <p> Subclasses that intend to support efficient single-character input
     * should override this method.
     *
     * @return     The character read, as an integer in the range 0 to 65535
     *             (<tt>0x00-0xffff</tt>), or -1 if the end of the stream has
     *             been reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read() throws IOException {
        char cb[] = new char[1];
        if (read(cb, 0, 1) == -1)
            return -1;
        else
            return cb[0];
    }

    /**
```

```
     * Reads characters into an array.  This method will block until some input
     * is available, an I/O error occurs, or the end of the stream is reached.
     *
     * @param       cbuf  Destination buffer
     *
     * @return      The number of characters read, or -1
     *              if the end of the stream
     *              has been reached
     *
     * @exception   IOException  If an I/O error occurs
     */
    public int read(char cbuf[]) throws IOException {
        return read(cbuf, 0, cbuf.length);
    }

    /**
     * Reads characters into a portion of an array.  This method will block
     * until some input is available, an I/O error occurs, or the end of the
     * stream is reached.
     *
     * @param      cbuf  Destination buffer
     * @param      off   Offset at which to start storing characters
     * @param      len   Maximum number of characters to read
     *
     * @return     The number of characters read, or -1 if the end of the
     *             stream has been reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    abstract public int read(char cbuf[], int off, int len) throws IOException;

    /** Maximum skip-buffer size */
    private static final int maxSkipBufferSize = 8192;

    /** Skip buffer, null until allocated */
    private char skipBuffer[] = null;

    /**
     * Skips characters.  This method will block until some characters are
     * available, an I/O error occurs, or the end of the stream is reached.
     *
     * @param  n  The number of characters to skip
     *
     * @return    The number of characters actually skipped
     *
     * @exception  IllegalArgumentException  If <code>n</code> is negative.
     * @exception  IOException  If an I/O error occurs
     */
    public long skip(long n) throws IOException {
        if (n < 0L)
            throw new IllegalArgumentException("skip value is negative");
        int nn = (int) Math.min(n, maxSkipBufferSize);
        synchronized (lock) {
            if ((skipBuffer == null) || (skipBuffer.length < nn))
                skipBuffer = new char[nn];
            long r = n;
            while (r > 0) {
                int nc = read(skipBuffer, 0, (int)Math.min(r, nn));
                if (nc == -1)
                    break;
                r -= nc;
            }
            return n - r;
        }
    }

    /**
     * Tells whether this stream is ready to be read.
     *
     * @return True if the next read() is guaranteed not to block for input,
     * false otherwise.  Note that returning false does not guarantee that the
     * next read will block.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public boolean ready() throws IOException {
        return false;
    }

    /**
     * Tells whether this stream supports the mark() operation. The default
     * implementation always returns false. Subclasses should override this
     * method.
     *
     * @return true if and only if this stream supports the mark operation.
     */
    public boolean markSupported() {
        return false;
    }

    /**
     * Marks the present position in the stream.  Subsequent calls to reset()
     * will attempt to reposition the stream to this point.  Not all
     * character-input streams support the mark() operation.
     *
     * @param  readAheadLimit  Limit on the number of characters that may be
     *                         read while still preserving the mark.  After
     *                         reading this many characters, attempting to
     *                         reset the stream may fail.
     *
     * @exception  IOException  If the stream does not support mark(),
     *                          or if some other I/O error occurs
     */
    public void mark(int readAheadLimit) throws IOException {
        throw new IOException("mark() not supported");
    }

    /**
     * Resets the stream.  If the stream has been marked, then attempt to
     * reposition it at the mark.  If the stream has not been marked, then
     * attempt to reset it in some way appropriate to the particular stream,
     * for example by repositioning it to its starting point.  Not all
     * character-input streams support the reset() operation, and some support
     * reset() without supporting mark().
     *
     * @exception  IOException  If the stream has not been marked,
     *                          or if the mark has been invalidated,
     *                          or if the stream does not support reset(),
     *                          or if some other I/O error occurs
     */
    public void reset() throws IOException {
        throw new IOException("reset() not supported");
    }

    /**
     * Closes the stream and releases any system resources associated with
     * it.  Once the stream has been closed, further read(), ready(),
     * mark(), reset(), or skip() invocations will throw an IOException.
     * Closing a previously closed stream has no effect.
     *
     * @exception  IOException  If an I/O error occurs
     */
     abstract public void close() throws IOException;

}
```

And here is the Java 8 version of FileReader, the specific type of Reader that BufferedReader actually contains. Note that there is no implementation of the read() method that BufferedReader uses.

```
/*
 * Copyright (c) 1996, 2001, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;


/**
 * Convenience class for reading character files.  The constructors of this
 * class assume that the default character encoding and the default byte-buffer
 * size are appropriate.  To specify these values yourself, construct an
 * InputStreamReader on a FileInputStream.
 *
 * <p><code>FileReader</code> is meant for reading streams of characters.
 * For reading streams of raw bytes, consider using a
 * <code>FileInputStream</code>.
 *
 * @see InputStreamReader
 * @see FileInputStream
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */
public class FileReader extends InputStreamReader {

   /**
    * Creates a new <tt>FileReader</tt>, given the name of the
    * file to read from.
    *
    * @param fileName the name of the file to read from
    * @exception  FileNotFoundException  if the named file does not exist,
    *                   is a directory rather than a regular file,
    *                   or for some other reason cannot be opened for
    *                   reading.
    */
    public FileReader(String fileName) throws FileNotFoundException {
        super(new FileInputStream(fileName));
    }

   /**
    * Creates a new <tt>FileReader</tt>, given the <tt>File</tt>
    * to read from.
    *
    * @param file the <tt>File</tt> to read from
    * @exception  FileNotFoundException  if the file does not exist,
    *                   is a directory rather than a regular file,
    *                   or for some other reason cannot be opened for
    *                   reading.
    */
    public FileReader(File file) throws FileNotFoundException {
        super(new FileInputStream(file));
    }

   /**
    * Creates a new <tt>FileReader</tt>, given the
    * <tt>FileDescriptor</tt> to read from.
    *
    * @param fd the FileDescriptor to read from
    */
    public FileReader(FileDescriptor fd) {
        super(new FileInputStream(fd));
    }

}
```

However, FileReader is a child of InputStreamReader.

```java
/*
 * Copyright (c) 1996, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

package java.io;

import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import sun.nio.cs.StreamDecoder;


/**
 * An InputStreamReader is a bridge from byte streams to character streams: It
 * reads bytes and decodes them into characters using a specified {@link
 * java.nio.charset.Charset charset}.  The charset that it uses
 * may be specified by name or may be given explicitly, or the platform's
 * default charset may be accepted.
 *
 * <p> Each invocation of one of an InputStreamReader's read() methods may
 * cause one or more bytes to be read from the underlying byte-input stream.
 * To enable the efficient conversion of bytes to characters, more bytes may
 * be read ahead from the underlying stream than are necessary to satisfy the
 * current read operation.
 *
 * <p> For top efficiency, consider wrapping an InputStreamReader within a
 * BufferedReader.  For example:
 *
 * <pre>
 * BufferedReader in
 *   = new BufferedReader(new InputStreamReader(System.in));
 * </pre>
 *
 * @see BufferedReader
 * @see InputStream
 * @see java.nio.charset.Charset
 *
 * @author      Mark Reinhold
 * @since       JDK1.1
 */

public class InputStreamReader extends Reader {

    private final StreamDecoder sd;

    /**
     * Creates an InputStreamReader that uses the default charset.
     *
     * @param  in   An InputStream
     */
    public InputStreamReader(InputStream in) {
        super(in);
        try {
            sd = StreamDecoder.forInputStreamReader(in, this, (String)null); //
## check lock object
        } catch (UnsupportedEncodingException e) {
            // The default encoding should always be available
            throw new Error(e);
        }
    }

    /**
     * Creates an InputStreamReader that uses the named charset.
     *
     * @param  in
     *         An InputStream
     *
     * @param  charsetName
     *         The name of a supported
     *         {@link java.nio.charset.Charset charset}
     *
     * @exception  UnsupportedEncodingException
     *             If the named charset is not supported
     */
    public InputStreamReader(InputStream in, String charsetName)
        throws UnsupportedEncodingException
    {
        super(in);
        if (charsetName == null)
            throw new NullPointerException("charsetName");
        sd = StreamDecoder.forInputStreamReader(in, this, charsetName);
    }

    /**
     * Creates an InputStreamReader that uses the given charset.
     *
     * @param  in       An InputStream
     * @param  cs       A charset
     *
     * @since 1.4
     * @spec JSR-51
     */
    public InputStreamReader(InputStream in, Charset cs) {
        super(in);
        if (cs == null)
            throw new NullPointerException("charset");
        sd = StreamDecoder.forInputStreamReader(in, this, cs);
    }

    /**
     * Creates an InputStreamReader that uses the given charset decoder.
     *
     * @param  in       An InputStream
     * @param  dec      A charset decoder
     *
     * @since 1.4
     * @spec JSR-51
     */
    public InputStreamReader(InputStream in, CharsetDecoder dec) {
        super(in);
        if (dec == null)
            throw new NullPointerException("charset decoder");
        sd = StreamDecoder.forInputStreamReader(in, this, dec);
    }

    /**
     * Returns the name of the character encoding being used by this stream.
     *
     * <p> If the encoding has an historical name then that name is returned;
     * otherwise the encoding's canonical name is returned.
     *
     * <p> If this instance was created with the {@link
     * #InputStreamReader(InputStream, String)} constructor then the returned
     * name, being unique for the encoding, may differ from the name passed to
     * the constructor. This method will return <code>null</code> if the
     * stream has been closed.
     * </p>
     * @return The historical name of this encoding, or
     *         <code>null</code> if the stream has been closed
     *
     * @see java.nio.charset.Charset
     *
     * @revised 1.4
     * @spec JSR-51
     */
    public String getEncoding() {
        return sd.getEncoding();
    }

    /**
     * Reads a single character.
     *
     * @return The character read, or -1 if the end of the stream has been
     *         reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read() throws IOException {
        return sd.read();
    }
    /**
     * Reads characters into a portion of an array.
     *
     * @param      cbuf     Destination buffer
     * @param      offset   Offset at which to start storing characters
     * @param      length   Maximum number of characters to read
     *
     * @return     The number of characters read, or -1 if the end of the
     *             stream has been reached
     *
     * @exception  IOException  If an I/O error occurs
     */
    public int read(char cbuf[], int offset, int length) throws IOException {
        return sd.read(cbuf, offset, length);
    }

    /**
     * Tells whether this stream is ready to be read.  An InputStreamReader is
     * ready if its input buffer is not empty, or if bytes are available to be
     * read from the underlying byte stream.
     *
     * @exception  IOException  If an I/O error occurs
     */
    public boolean ready() throws IOException {
        return sd.ready();
    }

    public void close() throws IOException {
        sd.close();
    }
}
```