Character streams

Version 1.1 of the Java Development Kit introduced support for *character streams* to the `java.io` package.

Prior to JDK 1.1, the standard I/O facilities supported only byte streams, via the `InputStream` and `OutputStream` classes and their subclasses. Character streams are like byte streams, but they contain 16-bit Unicode characters rather than eight-bit bytes. They are implemented by the `Reader` and `Writer` classes and their subclasses. `Reader`s and `Writer`s support essentially the same operations as `InputStream`s and `OutputStream`s, except that where byte-stream methods operate on bytes or byte arrays, character-stream methods operate on characters, character arrays, or strings.

Most of the functionality available for byte streams is also provided for character streams. This is reflected in the name of each character-stream class, whose prefix is usually shared with the name of the corresponding byte-stream class. For example, there is a `PushbackReader` class that provides the same functionality for character streams that is provided by `PushbackInputStream` for byte streams.

Why use character streams?

The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize.

Java stores strings in Unicode, an international standard character encoding that is capable of representing most of the world's written languages. Typical user-readable text files, however, use encodings that are not necessarily related to Unicode, or even to ASCII, and there are many such encodings. Character streams hide the complexity of dealing with these encodings by providing two classes that serve as bridges between byte streams and character streams. The `InputStreamReader` class implements a character-input stream that reads bytes from a byte-input stream and converts them to characters according to a specified encoding. Similarly, the `OutputStreamWriter` class implements a character-output stream that converts characters into bytes according a specified encoding and writes them to a byte-output stream.

A second advantage of character streams is that they are potentially much more efficient than byte streams. The implementations of many of Java's original byte streams are oriented around byte-at-a-time read and write operations. The character-stream classes, in contrast, are oriented around buffer-at-a-time read and write operations. This difference, in combination with a more efficient locking scheme, allows the character stream classes to make up for the added overhead of encoding conversion in many cases.

API overview

The character-stream classes have been designed to parallel the existing byte-stream classes in the `java.io` package. As noted above, the name of each character-stream class ends in `Reader` or `Writer`, as appropriate, while its prefix is usually shared with the corresponding byte-stream class, if any. The following table summarizes the new classes; in the left column, indentation indicates subclass relationships.

Character-stream class	***Description***	***Corresponding byte class***
`Reader`	Abstract class for character-input streams	`InputStream`
`BufferedReader`	Buffers input, parses lines	`BufferedInputStream`
`LineNumberReader`	Keeps track of line numbers	`LineNumberInputStream`
`CharArrayReader`	Reads from a character array	ByteArrayInputStream
`InputStreamReader`	Translates a byte stream into a character stream	*(none)*
`FileReader`	Translates bytes from a file into a character stream	`FileInputStream`
`FilterReader`	Abstract class for filtered character input	`FilterInputStream`
`PushbackReader`	Allows characters to be pushed back	`PushbackInputStream`
`PipedReader`	Reads from a `PipedWriter`	`PipedInputStream`
`StringReader`	Reads from a `String`	`StringBufferInputStream`
`Writer`	Abstract class for character-output streams	`OutputStream`

`BufferedWriter`	Buffers output, uses platform's line separator	`BufferedOutputStream`
`CharArrayWriter`	Writes to a character array	`ByteArrayOutputStream`
`FilterWriter`	Abstract class for filtered character output	`FilterOutputStream`
`OutputStreamWriter`	Translates a character stream into a byte stream	*(none)*
`FileWriter`	Translates a character stream into a byte file	`FileOutputStream`
`PrintWriter`	Prints values and objects to a `Writer`	`PrintStream`
`PipedWriter`	Writes to a `PipedReader`	`PipedOutputStream`
`StringWriter`	Writes to a `String`	*(none)*

Related changes

PrintStream

The `PrintStream` class has been modified to use the platform's default character encoding and the platform's default line terminator. Thus each `PrintStream` incorporates an `OutputStreamWriter`, and it passes all characters through this writer to produce bytes for output. The `println` methods use the platform's default line terminator, which is defined by the system property `line.separator` and is not necessarily a single newline character (`'\n'`). Bytes and byte arrays written via the existing `write` methods are not passed through the writer.

The primary motivation for changing the `PrintStream` class is that it will make `System.out` and `System.err` more useful to people writing Java programs on platforms where the local encoding is something other than ASCII. `PrintStream` is, in other words, provided primarily for use in debugging and for compatibility with existing code. Code that produces textual output should use the new `PrintWriter` class, which allows the character encoding to be specified or the default encoding to be accepted. For convenience, the `PrintWriter` class provides constructors that take an `OutputStream` object and create an intermediate `OutputStreamWriter` object that uses the default encoding.

Other classes

The following constructors and methods have been deprecated because they do not properly convert between bytes and characters:

```
       String DataInputStream.readLine()
  InputStream Runtime.getLocalizedInputStream(InputStream)
OutputStream Runtime.getLocalizedOutputStream(OutputStream)
              StreamTokenizer(InputStream)
              String(byte ascii[], int hibyte, int offset, int count)
              String(byte ascii[], int hibyte)
        void String.getBytes(int srcBegin, int srcEnd, byte dst[], int dstBegin)
```

Finally, the following constructor and methods have been added:

```
              StreamTokenizer(Reader)
      byte[] String.getBytes()
        void Throwable.printStackTrace(PrintWriter)
```