

## Stream i/o

### Data streams

An application may have access to several *data streams*. A data stream is essentially nothing more than a sequence of bytes. If the stream is a source of data for the application, it is an *input stream*. The application *reads* data from an input stream, removing it from the stream. If it is a destination for data from the application (or “sink”), it is an *output stream*. The application *writes* data to an output stream, appending it to the stream.



The actual source of the data in an input stream might be a user’s keyboard, a file, another program, a network connection, an external device, etc. Likewise, the destination of an output stream could be a terminal window, a file, another program, a network connection, and so on. In this document, we consider streams associated with files, the keyboard, and a terminal window. Note, though, that the actual source or sink of a data stream generally does not matter to an application.

A data stream can be finite – for instance, if the source of an input stream is a file – or conceptually unbounded – for instance, if the source of an input stream is a sensor that reports temperature every ten seconds. An application generally has a way of determining that an input stream is exhausted, that all the data has been read and no more data will appear in the stream, and a way of indicating that an output stream is complete, that no more data will be written to the stream.

The bytes that comprise a data stream can be interpreted in many ways. If they are to be interpreted as characters, the stream is usually referred to as *character stream*. Otherwise, the stream is a *binary stream*. For example, if the source of an input stream is a text file, the stream is a character stream. If the source is a file in which each group of four bytes is a two’s complement binary integer, the stream is a binary stream.

We need to be a little careful with our terminology in regard to Java. Specifically, Java represents characters with the 16-bit Unicode character set. In Java, the term *character stream* refers to a data stream whose elements are to be interpreted as Unicode characters. Any other data stream – even one whose elements are ordinary 8-bit ASCII characters – is a *byte stream*.

### OOJ library classes

In this section, we review the OOJ i/o library, available at <http://www.cs.uno.edu/~fred/OOJ/Utilities/Libraries/>. The standard package `java.io` is considered in the next section. If you are not interested in this library, you may skip this section.

The package `java.io` is rather formidable, and not easily digestible. The package `OOJ.basicIO` includes very basic classes for using simple data streams. It is based on a version of Bertrand Meyer's libraries for the programming language *Eiffel*. Some programmers staunchly oppose using libraries that are not either standard or home grown. Considering the amount of vendor and third-party software we trust, this seems to us a little narrow minded. After all, a major thrust of the paradigm is producing reusable software and building on the work of others.

The fundamental classes in `OOJ.basicIO` are `BasicFileReader` and `BasicFileWriter`. A `BasicFileReader` instance is associated with an input stream, and a `BasicFileWriter` instance is associated with an output stream. The data streams are assumed to contain characters represented with the default system encoding, typically ASCII.

### *BasicFileReader*

`BasicFileReader` has two constructors. The first requires that the name of an input file be provided as a `String` argument:

```
public BasicFileReader (String fileName)
    Create an input stream for the named file.
```

Exactly what constitutes a legal file name is system dependent. The file must exist, be accessible to the process, and should be a standard (ASCII) text file. Invoking the constructor creates an input stream whose source is the file. Reading from the stream is simply reading from the file. For example, if `fred.txt` is a text file,

```
BasicFileReader input = new BasicFileReader("fred.txt");
```

will create an object that reads the file `fred.txt`.

The second constructor requires no arguments, and creates a input stream associated with "standard input."

```
public BasicFileReader ()
    Create an input stream for standard input.
```

The source of standard input is determined by the operating system when the application is run. Typically standard input comes from the keyboard.

### *Reading from an input stream*

Once a `BasicFileReader` has been created for a stream, we can use its methods to read the data in the stream. There are five commands for reading one or more bytes from the stream, and four queries for determining what was read. A read command is typically followed by a query.

The commands are:

```
public void readChar ()
    Read a new character from this input stream.

public void readInt ()
    Read a new integer from this input stream.

public void readDouble ()
```

Read a new double from this input stream.

```
public void readLine ()  
    Read the rest of the line from this input stream.
```

```
public void readWord ()  
    Read a new word from this input.
```

The queries are:

```
public char lastChar ()  
    Character most recently read by readChar.
```

```
public int lastInt ()  
    int most recently read by readInt.
```

```
public double lastDouble ()  
    double most recently read by readDouble.
```

```
public String lastString ()  
    String most recently read by readWord or readLine.
```

There is also a query for determining whether or not the stream is exhausted:

```
public boolean eof ()  
    End of input stream has been reached.
```

For example, suppose that the first three lines of the file `fred.txt` contain the following, where “•” represents a space and “↵” represents the “newline” character at the end of the line:

```
•Hello↵  
••testing•↵  
••123xyz••↵
```

and that a `BasicFileReader` is created to read from the file:

```
BasicFileReader input = new BasicFileReader("fred.txt");
```

Executing the command

```
input.readChar();
```

will cause the first character of the file (a space) to be read. The statement

```
char c = input.lastChar();
```

will assign the character to the variable `c`. Executing `readChar` again will cause the next character (the “H”) to be read, and so on.

The following loop will read characters skipping spaces, and assign to `c` the character following the spaces:

```
input.readChar();  
while (input.lastChar() == ' ' )  
    input.readChar();
```

```
c = input.lastChar();
```

If the second line of file described above were being read, the character “t” would be assigned to `c`.

The method `readInt` skips any “white space” at the beginning of the input stream. “White space” is any sequence of spaces, tabs, end of lines, *etc.* The characters following the white space must denote an optionally-signed decimal integer. The method reads these characters, and the integer value denoted can be obtained with the query `lastInt`. For example, if the third line of the file described above were being read, the statement

```
input.readInt();
```

would read characters up to (but not including) the “x”. The statement

```
int i = input.lastInt();
```

would assign the value 123 to the variable `i`.

The method `readDouble` is similar, but requires that the characters following the white space have the form of an optionally signed double literal.

The method `readWord` reads the next word in the input stream, where a “word” is defined to be a sequence of non white space characters. The method `readLine` reads the remainder of the current line.

Suppose, for example, the file `fred.txt` contains five lines:

```
•Hello↵
••testing•↵
••123xyz••↵
↵
•+3e+4abc•↵
```

We show the input stream after each command, and what the relevant query returns.

command	input stream	lastChar()	lastString()	lastInt()	lastDouble()
	•Hello↵ ••testing•↵ ••123xyz••↵ ↵ •+3e+4abc•↵	?	?	?	?
input.readChar();					
	Hello↵ ••testing•↵ ••123xyz••↵ ↵ •+3e+4abc•↵	' '	?  "testing"	?  123	?  3e4
input.readChar();					
	ello↵ ••testing•↵ ••123xyz••↵ ↵ •+3e+4abc•↵	'H'	?	?	?
input.readLine();					
	••testing•↵ ••123xyz••↵ ↵ •+3e+4abc•↵	'H'	"ello"	?	?
input.readWord();					
	•↵ ••123xyz••↵ ↵ •+3e+4abc•↵	'H'	"testing"	?	?
input.readInt();					
	xyz••↵ ↵ •+3e+4abc•↵	'H'	"testing"	123	?
input.readLine();					
	↵ •+3e+4abc•↵	'H'	"xyz "	123	?
input.readDouble();					
	abc•↵	'H'	"xyz "	123	3e4

The query eof (“end of file”) can be used to determine when the input stream is exhausted. For instance, the following method counts the number of lines in the named file.

```
/**
 * The number of lines in the specified file.
 */
public int lineCount (String fileName) {
    BasicFileReader input = new BasicFileReader(fileName);
    int count = 0;
    while (!input.eof()) {
        input.readLine();
        count = count + 1;
    }
    return count;
}
```

Suppose each line of a file contains space separated integer student number, integer test score, and student name. For example, a line might look like this:

```
96669 66 Back, Helen
```

The following method reads the file, and produces a list of test scores.

```
/**
 * Test scores from Student data.
 *   require:
 *     each line of the named file must have the format:
 *       integer integerTestScore unspecified
 */
public IntegerList scores (String fileName) {
    BasicFileReader input = new BasicFileReader(fileName);
    IntegerList scores = new IntegerList();
    while (!input.eof()) {
        input.readInt();        // skip 1st int on line
        input.readInt();
        scores.append(new Integer(input.lastInt()));
        input.readLine();
    }
    return scores;
}
```

### *BasicFileWriter*

BasicFileWriter also has two constructors, one requiring a file name as argument and the other creating a stream to standard output.

```
public BasicFileWriter (String fileName)
    Create an output stream for the named file.
```

```
public BasicFileWriter ()
```

Create an output stream for standard output.

If the file specified in the first constructor does not exist, it will be created. If it exists, it will be overwritten. As with standard input, the source of standard output is determined by the operating system when the application is run. Typically standard output goes to a terminal window.

The commands write standard text (typically ASCII characters) to the stream. There are two versions of each output command, one that appends a “newline” character and one that doesn’t.

```
public void display (char ch)
```

Write the specified char to the output stream.

```
public void display (int value)
```

Write a decimal representation of the specified int to the output stream.

```
public void display (double value)
```

Write a decimal representation of the specified double to the output stream.

```
public void display (Object obj)
```

Write a String representation of the specified Object to the output stream.

```
public void display (String st)
```

Write the specified String to the output stream.

```
public void displayLine (char ch)
```

Write the specified char to the output stream, followed by a line separator.

```
public void displayLine (int value)
```

Write a decimal representation of the specified int to the output stream, followed by a line separator.

```
public void displayLine (double value)
```

Write a decimal representation of the specified double to the output stream, followed by a line separator.

```
public void displayLine (Object obj)
```

Write a String representation of the specified Object to the output stream, followed by a line separator.

```
public void displayLine (String st)
```

Write the specified String to the output stream, followed by a line separator.

Utility methods are provided for writing blank lines, flushing the output buffer, and closing the stream:

```
public void blankLine (int n)
```

Write the specified number of blank lines to the output stream.

```
public void flush ()
```

Flush the output buffer: write any buffered data to the output

```
public void close ()  
    Close the output stream.
```

Output is typically buffered in memory until a line is completed. For instance, if output is a `BasicFileWriter`,

```
basicFileWriter output = new basicFileWriter();
```

executing the command

```
output.display("Hello");
```

will not send the characters to the output stream immediately. The characters will be stored in a memory buffer until a complete line can be written. If the next command is

```
output.displayLine(" World!");
```

the eleven characters “Hello World” followed by a newline will be sent to the output stream.

Sometimes you would like to characters to be displayed immediately without a newline. For instance, if you are trying to prompt the user for input, you might want the characters

```
Please enter the file name:
```

to be written in the user’s window, with the cursor remaining at the end of the line. The command `flush` causes any buffered output to be written to the output stream. So the pair of commands

```
output.display("Please enter the file name: ");  
output.flush();
```

displays the string immediately without waiting for a line-ending newline.

The command `close` also flushes the buffer, and closes the output stream.

### *Exceptions*

The package `OOJ.basicIO` defines three exceptions: `DataException`, `EOFException`, and `IOException`. These are all unchecked exceptions (`RuntimeExceptions`), so you can ignore them if you wish.

A `DataException` is thrown by `readInt` or `readDouble` if the characters in the input stream don’t have the appropriate format. For instance, if `readInt` is invoked and the next non-blank character in the input stream is a letter, then a `DataException` is thrown.

Catching a `DataException` can be used to direct a “retry” in an interactive input method. For instance, suppose we want the user to enter an integer temperature value. We might write

```
int temp;  
output.display("Enter temperature, as an integer: ");  
output.flush();  
input.readInt();  
temp = input.lastInt();
```



If the user enters something besides a legal integer, we might want to give him another chance. We catch the `DataException`, and try again.

```
int temp;
boolean needTemp = true;
while (needTemp) {
    try {
        output.display("Enter temperature, as an integer: ");
        output.flush();
        input.readInt();
        temp = input.lastInt();
        needTemp = false;
    } catch (DataException e) {
        output.displayLine("Please enter an integer!");
    }
}
```

When the invocation of `readInt` succeeds, we set `needTemp` to `false`, and are done. If it fails because the user has not entered a legal integer, we catch the exception, write out an informational message and try again.

The exception `EOFException` is thrown by a read method if there are no more characters in the input stream, or not enough characters to satisfy the read request. For instance, if `readInt` is invoked and only characters remaining in the input stream are white space characters, the method will read past the white space and then throw an `EOFException`.

Any other problem with reading or writing results in an `IOException` being thrown.

## Standard java.io library classes

The standard package `java.io` is a menagerie containing, at last count, ten interfaces, fifty classes, and sixteen exceptions. The functionality can be categorized as

- facilities for reading and writing data streams;
- facilities for manipulating files;
- facilities for serializing objects.

File manipulation facilities include the class `File`, which models a file in the local file system, and the class `RandomAccessFile`, which provides mechanisms for reading and writing a file in a non-sequential manner. Object serialization provides a means for writing objects to a byte stream, and later recreating the objects from the byte stream. We will not consider these facilities in this document.

The classes that support reading and writing from data streams can be organized into four categories:

- classes for reading byte streams;
- classes for writing byte streams;
- classes for reading character streams;
- classes for writing character streams.

Recall that a Java character stream is a data stream whose elements are 16-bit Unicode characters. Any other data stream is a byte stream. The character stream classes were developed after the byte stream classes. So although there are similarities, byte stream and character stream classes are not entirely symmetric.

Each category has an abstract class at the top of its hierarchy: `InputStream` and `OutputStream` for byte streams, and `Reader` and `Writer` for character streams. Other classes in the hierarchies extend the functionality of the base classes in two ways: some add functionality by extending the base classes; others add functionality by wrapping an instance of another class. The benefits of composition (wrapping) are explained in [NH 15]. Principally, composition allows the extended functionality to be applied dynamically, at run-time. We consider some of the central classes in each category.

### *Input byte streams*

#### Abstract class `InputStream`

The top of the hierarchy is the abstract class `InputStream`. Fundamental methods for reading the stream are specified in this class, including:

```
public abstract int read () throws IOException
    Read and return the next byte of data from the input stream. Return -1 if the end of the
    stream is encountered (the stream is exhausted).

    ensure:
        0 <= this.read() <= 255 || this.read() == -1
```

Note that this method is neither a proper command nor a proper query. It changes the state of the stream and returns a value. Most of the `java.io` input methods are of this flavor.

The method `read` returns the byte value as a non-negative `int`. For instance, if the next byte in the stream is the ASCII character 'A', the integer value 65 will be returned: the character 'A' is denoted by the eight bits 01000001 in ASCII, and 01000001 is the binary representation of 65.

If no data is available but the end of the stream has not been seen, the method waits for more data to appear. For instance, if the data source is the keyboard, and all the data previously keyed by the user has been read, the method `read` blocks until the user keys another line. (How the user closes an input stream whose source is the keyboard depends on the operating system. Keying `Control-D` on a line by itself is typical.)

If the next byte cannot be read (for a reason other than the end of the stream being encountered), the method throws a `java.io.IOException`. This is a checked exception: the method invoking `read` must either catch the exception or include a `throws` clause indicating that it might also throw the exception.

Recall that the primitive Java type `char` is a numeric type. A `char` value is automatically extended to an `int` when necessary, and an `int` value can be explicitly converted (by casting) to a `char`. Furthermore, the 16-bit Unicode representation of a standard ASCII character is the 8-bit ASCII character with high-order 0's added. For example, the Unicode representation of the Latin 'A' is the 16-bits 00000000 01000001. Considered as a binary integer, this is also 65. If `input` is

an `InputStream` containing ASCII characters (from the keyboard, for instance), we can safely convert the ASCII characters to values of type `char`:

```
int i = input.read();
if (i != -1)
    char c = (char)i;
```

Two additional read methods are specified in the class `InputStream`. These both require a byte array as argument, and store data into the array. For instance,

```
public int read (byte[] b) throws IOException
```

Read some number of bytes from the input stream and store them into the array `b`. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.

If no byte is available because the stream is at end of file, the value `-1` is returned; otherwise, at least one byte is read and stored into `b`. The number of bytes read is, at most, equal to the length of `b`.

If the first byte cannot be read for any reason other than end of file, then an `IOException` is thrown. In particular, an `IOException` is thrown if the input stream has been closed.

Other methods specified include

```
public void close () throws IOException
```

Close the input stream and release associated resources.

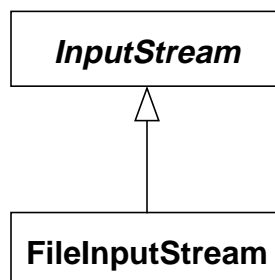
```
public long skip (long n) throws IOException
```

Skip over and discard `n` bytes of data from this input stream. The actual number of bytes skipped is returned. If `n` is negative, no bytes are skipped.

For details of these and other methods, see the library specifications at <http://java.sun.com>.

### Class `FileInputStream`

`FileInputStream` is a straightforward concrete extension of `InputStream`.



`FileInputStream` specifies a file as the source of the input stream, but otherwise adds no functionality to that specified by `InputStream`. A `FileInputStream` is generally wrapped

with a `DataInputStream`, `BufferedInputStream`, or `InputStreamReader` to provide a richer interface. We postpone examples until we consider these classes.

The file is identified – either with a `String` file name, a `File` object, or a system-specific file descriptor – in the `FileInputStream` constructor. The file is implicitly opened when the `FileInputStream` instance is created. The constructors are:

```
public FileInputStream (String name)
    throws FileNotFoundException, SecurityException

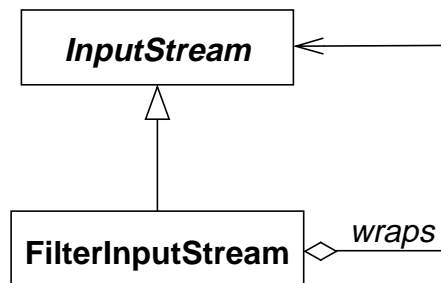
public FileInputStream (File file)
    throws FileNotFoundException, SecurityException

public FileInputStream (FileDescriptor fd)
    throws SecurityException
```

The exception `java.io.FileNotFoundException` is a checked exception; `java.lang.SecurityException` is unchecked.

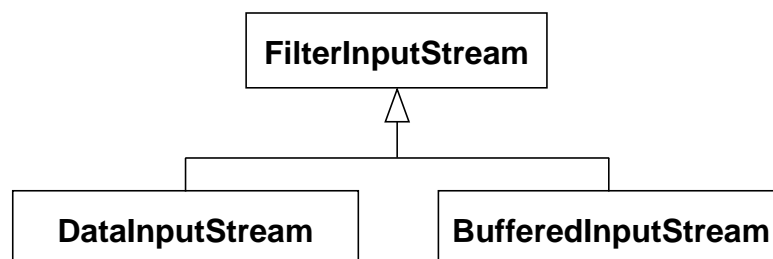
### Class `FilterInputStream`

`FilterInputStream` provides no additional functionality, but serves as a base class for `InputStream` wrappers.



Note that `FilterInputStream` is also a subclass of `InputStream`. This allows one `FilterInputStream` to wrap another `FilterInputStream`.

Two commonly used subclasses of `FilterInputStream` are `DataInputStream` and `BufferedInputStream`.



`DataInputStream` provides methods for reading values of primitive Java data types from the input stream. Included are the following:

```
public boolean readBoolean () throws IOException
    Read one byte and return true if the byte is non-zero, false if the byte is zero.

public char readChar () throws IOException
    Read two bytes and return the value as a Unicode character.

public double readDouble () throws IOException
    Read eight bytes and return the value as a double.

public int readInt () throws IOException
    Read four bytes and return the value as an int.

public byte readByte () throws IOException
    Read and return one input byte. The byte is treated as a signed value in the range -128
    through 127, inclusive.
```

Note that these methods do *not* read character representation of the values, but binary representations. You would not use these methods to read an ASCII file for instance. They throw a `java.io.EOFException` if the end of the input stream is encountered during the read attempt. (`java.io.EOFException` is a subclass of `java.io.IOException`.)

A `BufferedInputStream` uses an in-memory buffer to store input from the stream. That is, a large number of bytes are read from the input stream and stored in an internal buffer. Bytes are then read directly from the internal buffer. When the buffer is exhausted, it is filled again with another chunk of data from the input stream. Of course, the operating system also buffers input data in memory if possible. Using a `BufferedInputStream`, however, reduces the number of calls to the operating system. The operating system need only be accessed occasionally to fill the buffer.

As we've said, these subclasses of `FilterInputStream` wrap an `InputStream`. The `InputStream` component is provided as a constructor argument:

```
public DataInputStream (InputStream in)
    Create a DataInputStream that reads from the given InputStream.

public BufferedInputStream (InputStream in)
    Create a BufferedInputStream that buffers input from the given
    InputStream in a buffer with the default size of 2048 bytes.
```

To see how this works, suppose the file `noise.dat` contains a sequence of 32-bit (four byte) integer values. The file can be opened and wrapped with a `DataInputStream` as

```
FileInputStream in = new FileInputStream("noise.dat");
DataInputStream data = new DataInputStream(in);
```

or simply as

```
DataInputStream data =
    new DataInputStream(new FileInputStream("noise.dat"));
```

The integer values can be read by using the `DataInputStream` method `readInt`:

```
int i;
try {
    while (true) {
        i = data.readInt();
        process(i);
    }
} catch (EOFException e) {
    data.close();
}
```

Using an exception to detect the expected end of input condition is annoying.

We mentioned that since `FilterInputStream` is a subclass of `InputStream`, one `FilterInputStream` can wrap another. If we wanted to buffer input from the above file, we could first wrap the `FileInputStream` in `BufferedInputStream`:

```
FileInputStream in = new FileInputStream("noise.dat");
BufferedInputStream bf = new BufferedInputStream(in);
DataInputStream data = new DataInputStream(bf);
```

Of course, there is really no need to name all these instances. We could just as easily write:

```
DataInputStream data =
    new DataInputStream(
        new BufferedInputStream(
            new FileInputStream("noise.dat")));
```

### *Input character streams*

#### Abstract class Reader

The top level of the input character stream hierarchy is the abstract class `Reader`. `Reader` is similar in purpose and structure to `InputStream`, but `Reader` reads a stream of Unicode characters rather than bytes. Its basic read method is specified as follows:

```
public int read () throws IOException
    Read and return the next character of data from the input stream. The character is
    returned as an integer in the range 0 to 65535 (0 to 216-1). Returns -1 if the end of the
    stream is encountered (the stream is exhausted).

    ensure:
        0 <= this.read() <= 65535 || this.read() == -1
```

Note that the method returns a value of type `int`, not of type `char`. (Why? So that there is a convenient “non-character” value that can be returned if the end of the stream has been reached.) The postcondition guarantees that the `int` can be safely cast to a `char`. If `reader` is a `Reader`, we can write

```
int i = reader.read();
```

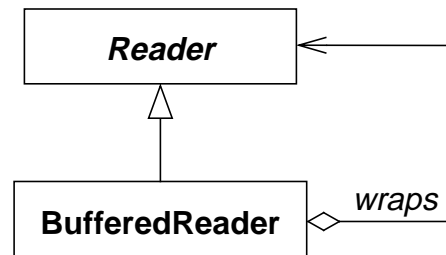
```

    if (i != -1)
        char c = (char)i;

```

### Class BufferedReader

BufferedReader is used to buffer character stream input in much the same way that BufferedInputStream is used to buffer byte stream input.



BufferedReader has a handy method for reading a line of input:

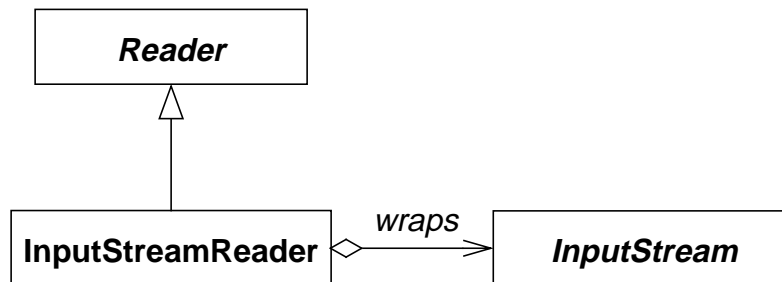
```

public String readLine () throws IOException
    Read and return a line of text. Return null if the end of the stream is encountered.
    Line terminating characters are not included in the String.

```

### Class InputStreamReader

InputStreamReader is an adapter class that wraps an InputStream and provides the functionality of a Reader.



The **InputStreamReader** converts each byte of the **InputStream** to a Unicode character using an encoding scheme that can be specified when the **InputStreamReader** is created. If no encoding scheme is specified, a system default is used.

```

public InputStreamReader (InputStream in)
    Create an InputStreamReader that reads from the given InputStream and
    translates bytes to characters using the system default encoding.

public InputStreamReader (InputStream in, String enc)
    throws UnsupportedOperationException

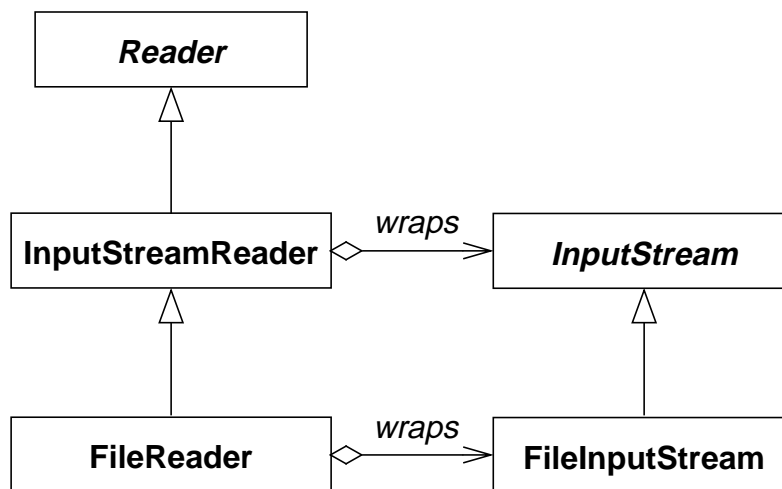
```

Create an `InputStreamReader` that reads from the given `InputStream` and translates bytes to characters using the specified encoding.

A typical default encoding is ISO-8859-1 (Latin 1). This is an 8-bit character set that covers most Western European languages. The first 128 characters (characters 0 to 127) are identical to ASCII. To explicitly specify this encoding, give the string "ISO8859\_1" as the second argument to the constructor.

### Class `FileReader`

`FileReader` extends `InputStreamReader`. In effect, a `FileReader` is an `InputStreamReader` using the system default encoding and wrapped around a `FileInputStream`.



The constructors are similar to those of `FileInputStream`:

```
public FileReader (String name)
    throws FileNotFoundException, SecurityException

public FileReader (File file)
    throws FileNotFoundException, SecurityException

public FileReader (FileDescriptor fd)
    throws SecurityException
```

### *Examples*

#### A line-by-line reader

First, let's write a simple method that takes the name of a text file as argument, counts the number of lines in the file, and returns the count. We will use the `readLine` method of `BufferedReader` to read each line of input.

```
/**
```



```

    * The number of lines in the specified file.
    *   require:
    *       the specified file must be a text file.
    */
int lineCount (String fileName) throws IOException {
    BufferedReader input =
        new BufferedReader(new FileReader(fileName));
    int count = 0;
    while (input.readLine() != null)
        count = count + 1;
    input.close();
    return count;
}

```

Creating the `FileReader` creates a `FileInputStream` for reading the file and opens the file. The `FileReader` is passed to the `BufferedReader` constructor. The `BufferedReader` wraps the `FileReader`, providing the `readLine` functionality.

### Two character-by-character readers

As a second example, let's write a method that counts the number of spaces in a text file. Again, we use a `BufferedReader`, and read characters one at a time.

```

/**
 * The number of spaces in the specified file.
 *   require:
 *       the specified file must be a text file.
 */
int spaceCount (String fileName) throws IOException {
    BufferedReader input =
        new BufferedReader(new FileReader(fileName));
    int count = 0;
    int ch = input.read();
    while (ch != -1) {
        if (ch == ' ')
            count = count + 1;
        ch = input.read();
    }
    input.close();
    return count;
}

```

Next, suppose we want a method that counts the number of words in a file, where a word is any sequence of non white space characters. Words are separated by white space. (Recall that “white space” is composed of spaces, tabs, end of lines, *etc.*) The Reader classes only provide methods for reading a file character by character or line by line. Our approach is to read the file character by character and check for white space. The standard class `Character` has a useful method:

```

public static boolean isWhitespace (char ch)

```

Determine if the specified character is white space.

Using this method, we can write:

```
/**
 * The number of words in the specified file.
 *   require:
 *     the specified file must be a text file.
 */
int wordCount (String fileName) throws IOException {
    BufferedReader input =
        new BufferedReader(new FileReader(fileName));
    int count = 0;
    int ch = input.read();
    while (ch != -1) {
        if (Character.isWhitespace((char)ch))
            do // skip white space
                ch = input.read();
            while (ch != -1 && Character.isWhitespace((char)ch));
        else {
            count = count + 1;
            do // skip rest of word
                ch = input.read();
            while (ch != -1 && !Character.isWhitespace((char)ch));
        }
    }
    input.close();
    return count;
}
```

The first character in a word or sequence of white space is identified, and the remaining characters in the word or white space are skipped.

### Using a tokenizer

The class `java.util.StringTokenizer` can be useful for a problem like the previous. A `StringTokenizer` is given a `String` as argument when it is created, and breaks the string into a sequence of “tokens” separated by “delimiters.” A token is simply a sequence of characters. A delimiter is a character that separates tokens. The delimiter characters can be specified when the `StringTokenizer` is created. Two constructors are:

```
public StringTokenizer (String str, String delim)
    Construct a string tokenizer for the specified string. The characters in the delim argument are the delimiters for separating tokens.

public StringTokenizer (String str)
    Construct a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is: the space character, the tab character, the newline character, the carriage-return character, and the form-feed character.
```

Functions provided by the StringTokenizer include:

```
public boolean hasMoreTokens ()
    There are more tokens available from this tokenizer's string.

public String nextToken ()
    The next token from this string tokenizer.

require:

    this.hasMoreTokens()
```

In out case, the default set of white space delimiters is adequate. using a tokenizer, we can implement the method as follows:

```
int wordCount (String fileName) throws IOException {
    BufferedReader input =
        new BufferedReader(new FileReader(fileName));
    int count = 0;
    String line = input.readLine();
    while (line != null) {
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            tokenizer.nextToken();
            count = count+1;
        }
        line = input.readLine();
    }
    input.close();
    return count;
}
```

Note that the token returned by nextToken is ignored.

As a final example, suppose each line of a file contains space separated integer student number, integer test score, and student name. For example, a line might look like this:

```
96669 66 Back, Helen
```

We write a method that reads the file and produces a test average (mean). Again, we'll use a BufferedReader and a StringTokenizer.

But we must convert the second token of each line into an integer value. To do this, we use the method parseInt from the standard class Integer. This method is specified as

```
public static int parseInt (String s)
    throws NumberFormatException
    Parse the string argument as a signed decimal integer. The characters in the string must
    all be decimal digits, except that the first character may be an ASCII minus sign '-' to
    indicate a negative value. The resulting integer value is returned. Throws Number-
    FormatException if the string does not contain a parsable integer.
```

```

/**
 * Average of test scores from Student data.
 *   require:
 *     each line of the named file must have the format:
 *       integer integerTestScore unspecified
 */
public double average (String fileName) throws
    IOException, NumberFormatException {
    BufferedReader input =
        new BufferedReader(new FileReader(fileName));
    int count = 0;
    int total = 0;
    String line = input.readLine();
    while (line != null) {
        StringTokenizer tokenizer = new StringTokenizer(line);
        tokenizer.nextToken(); // skip first token on line
        int grade = Integer.parseInt(tokenizer.nextToken());
        total = total + grade;
        count = count + 1;
        line = input.readLine();
    }
    input.close();
    return (double)total/(double)count;
}

```

The class `java.io.StreamTokenizer` can also be helpful in parsing input text. We do not consider that class here.

### *Output byte streams*

The collection of output stream classes mirrors the input classes. At the top of the output byte stream hierarchy is the abstract class `OutputStream`. Fundamental methods are:

```

public abstract void write (int b) throws IOException
    Write the specified byte (low order 8 bits of the int provided) to the output stream.

public void close () throws IOException
    Close the output stream and release any associated resources.

public void flush () throws IOException
    Write any buffered bytes to the output stream.

```

`FileOutputStream` extends `OutputStream` by allowing a file to be specified as the destination of the output in much the same way that `FileInputStream` extends `InputStream`.

`FilterOutputStream` provides a base class for `OutputStream` wrappers, again in a way similar to its input counterpart, `FilterInputStream`. `BufferedOutputStream` and `DataOutputStream` extend `FilterOutputStream`, and provide functionality symmetric to their input stream counterparts. (The class `PrintStream` also extends `FilterOutput-`

Stream. However, `PrintWriter`, discussed below, should be used instead of `PrintStream`.) Details on these classes can be obtained from the standard documentation. We do not consider them further.

### *Output character streams*

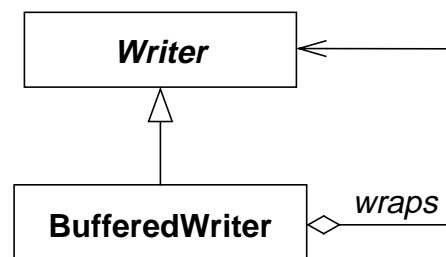
#### Class `Writer`

The abstract class `Writer` is at the top of the output character stream hierarchy. It is similar in functionality to `OutputStream`, but its `write` method writes a Unicode character rather than a byte.

```
public abstract void write (int c) throws IOException
    Write a character consisting of the low order 16 bits of the int provided to the output
    stream.
```

#### Class `BufferedWriter`

The wrapper class `BufferedWriter` extends `Writer`, and is symmetric to the Reader class `BufferedReader`.



`BufferedWriter` constructors require that a `Writer` be provided as argument.

A `BufferedWriter` writes characters to a memory buffer rather than directly to the output stream. The buffer is emptied to the output stream as needed. The class defines methods for explicitly emptying the buffer, and for writing a newline.

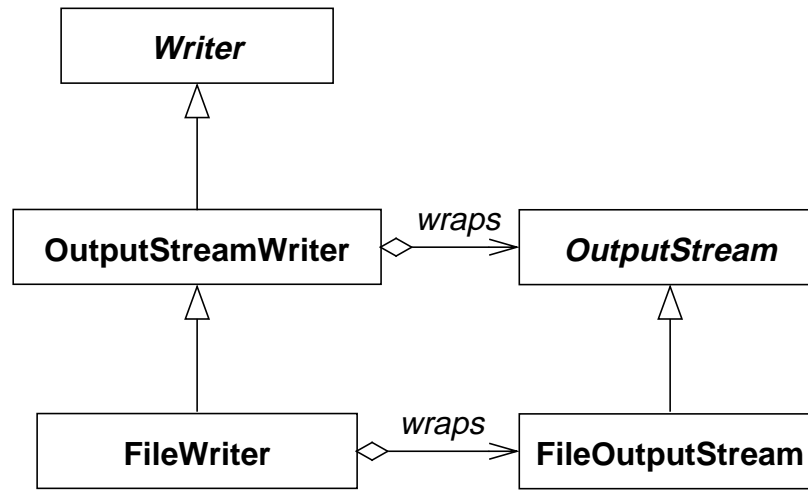
```
public void newLine () throws IOException
    Write a system-defined line separator.

public void flush () throws IOException
    Flush the buffer.
```

#### Classes `OutputStreamWriter` and `FileWriter`

`OutputStreamWriter` adapts an `OutputStream` to a `Writer`, and is symmetric to `InputStreamReader`. An `OutputStreamReader` converts the character stream produced by the `Writer` to a byte stream, using either a specified encoding or the system default encoding.

`FileWriter` extends `OutputStreamWriter` in a manner similar to `FileReader`'s extension of `InputStreamReader`.



Constructors for `FileWriter` require that the file be specified either with a `String` name, a `File` object, or a system-dependent file descriptor.

```
public FileWriter (String fileName) throws IOException
    Construct a FileWriter object given a file name.
```

```
public FileWriter (File file) throws IOException
    Construct a FileWriter object given a File object.
```

```
public FileWriter (FileDescriptor fd)
    Construct a FileWriter object associated with a file descriptor.
```

A `FileWriter` will overwrite an existing file. One constructor allows you to specify that an existing file is to be appended rather than overwritten.

```
public FileWriter (String fileName, boolean append)
    throws IOException
    Construct a FileWriter object given a file name. If append is true, data will be
    written to the end of the file rather than the beginning.
```

As an example, let's write a method that copies a file, replacing each sequence of one or more spaces with a single space.

```
/**
 * Copy input file to output file, replacing each sequence of
 * one or more spaces with a single space.
 * require:
 *     the specified inputFile must be a text file.
 */
void squeezeCopy (String inputFile, String outputFile)
    throws IOException {
```

```

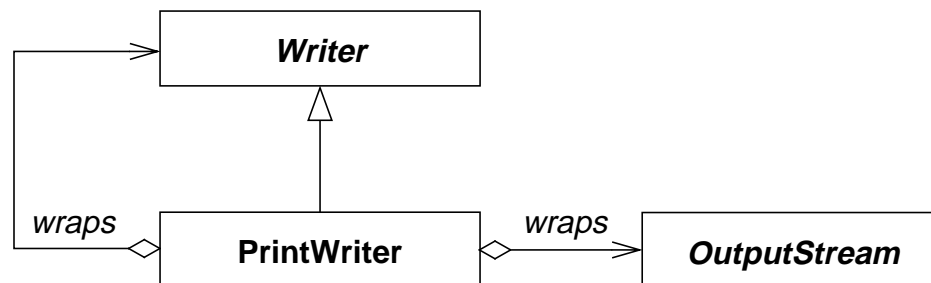
BufferedReader input =
    new BufferedReader(new FileReader(inputFile));
BufferedWriter output =
    new BufferedWriter(new FileWriter(outputFile));
int ch = input.read();
while (ch != -1) {
    output.write(ch);
    if (ch == ' ')
        do // skip other spaces
            ch = input.read();
        while (ch != -1 && ch == ' ');
    else
        ch = input.read();
}
input.close();
output.close();
}

```

### Class PrintWriter

The class `PrintWriter` is one of the most useful output stream classes. `PrintWriter` extends `Writer` and wraps either an `OutputStream` or a `Writer`. (If an `OutputStream` is specified in the `PrintWriter` constructor, an intermediate `OutputStreamWriter` wrapping the `OutputStream` is automatically created. Thus properly `PrintWriter` wraps a `Writer` which might be an `OutputStreamWriter`.)

`PrintWriter` provides functionality for writing string representations of primitive values and objects. If an `OutputStream` is wrapped, characters are converted to bytes using the system default encoding scheme.



The four constructors are specified as follows:

```

public PrintWriter (OutputStream out)
    Create a PrintWriter that sends output to the specified OutputStream. An
    intermediate OutputStreamWriter that converts characters to bytes using the
    system default encoding is also constructed.

public PrintWriter (OutputStream out, boolean autoFlush)

```

Create a `PrintWriter` that sends output to the specified `OutputStream`. An intermediate `OutputStreamWriter` that converts characters to bytes using the system default encoding is also constructed.

If `autoFlush` is `true`, the `PrintWriter` calls its `flush` method after every invocation of `println`.

```
public PrintWriter (Writer out)
```

Create a `PrintWriter` that sends output to the specified `Writer`.

```
public PrintWriter (Writer out, boolean autoFlush)
```

Create a `PrintWriter` that sends output to the specified `Writer`.

If `autoFlush` is `true`, the `PrintWriter` calls its `flush` method after every invocation of `println`.

Among the methods provided are these.

```
public void print (boolean b)
```

Write `"true"` or `"false"` to the output stream depending on the value specified.

```
public void print (char c)
```

Write the specified character to the output stream.

```
public void print (double d)
```

Write a string representation of the specified `double` to the output stream.

```
public void print (Object obj)
```

Write a string representation of the specified `Object` to the output stream, using the `Object`'s `toString` method.

```
public void print (String s)
```

Write the specified `String` to the output stream.

```
public void println ()
```

Write a (system dependent) line separator to the output stream

```
public void println (boolean b)
```

Write `"true"` or `"false"` to the output stream depending on the value specified, followed by a line separator.

```
public void println (char c)
```

Write the specified character to the output stream, followed by a line separator.

```
public void println (double d)
```

Write a string representation of the specified `double` to the output stream, followed by a line separator.

```
public void println (Object obj)
```

Write a string representation of the specified `Object` to the output stream, using the `Object`'s `toString` method, followed by a line separator.

```
public void println (String s)
```



Write the specified `String` to the output stream, followed by a line separator.

### System constants

The data streams standard input, standard output, and standard error are accessible through constants defined in the class `java.lang.System`. Standard input is specified as an `InputStream`, while standard output and standard error are specified as `PrintStreams`. (Error messages, of course, should be written to standard error, not standard output.)

```
public static final PrintStream err; // standard error
public static final PrintStream out; // standard out
public static final InputStream in; // standard in
```

Here's a simple method that copies a file line by line to standard output. If no file is specified, the method copies standard input to standard output. The method assumes that the input is made up of a number of lines.

Note that the `InputStream System.out` is first adapted to a `Reader` by wrapping it in an `InputStreamReader`. The `InputStreamReader` is wrapped in a `BufferedReader`.

```
/**
 * Copy the specified file to standard output. If fileName
 * is null, copy standard input to standard output.
 * require:
 *     the specified file must be a text file,
 *     composed of lines. (In particular, if not empty
 *     it must end with a line terminator.)
 */
void miniCat (String fileName) throws IOException {
    BufferedReader input;
    if (fileName == null)
        input = new BufferedReader(
            new InputStreamReader(System.in));
    else
        input = new BufferedReader(
            new FileReader(fileName));
    String line = input.readLine();
    while (line != null) {
        System.out.println(line);
        line = input.readLine();
    }
    input.close();
    System.out.close();
}
```