



CHAPTER 10

java.io

Package **java.io**

Java 1.0

The `java.io` package is large, but most of the classes it contains fall into a well-structured hierarchy. Most of the package consists of byte streams—subclasses of `InputStream` or `OutputStream` and character streams—subclasses of `Reader` or `Writer`. Each of these stream subtypes has a specific purpose, and, despite its size, `java.io` is a straightforward package to understand and to use. In Java 1.4, the `java.io` package is complemented by a “New I/O API” defined in the `java.nio` package and its subpackages. The `java.nio` package is totally new, although it includes some compatibility with the classes in this package. It was designed for high-performance I/O, particularly for use in servers, and has a lower-level API than this package does. The I/O facilities of `java.io` are still quite adequate for most of the I/O required by typical client-side applications.

Before we consider the stream classes that comprise the bulk of this package, let’s examine the important non-stream classes. `File` represents a file or directory name in a system-independent way and provides methods for listing directories, querying file attributes, and renaming and deleting files. `FilenameFilter` is an interface that defines a method that accepts or rejects specified filenames. It is used by `File` to specify what types of files should be included in directory listings. `RandomAccessFile` allows you to read from or write to arbitrary locations of a file. Often, though, you’ll prefer sequential access to a file and should use one of the stream classes.

`InputStream` and `OutputStream` are abstract classes that define methods for reading and writing bytes. Their subclasses allow bytes to be read from and written to a variety of sources and sinks. `FileInputStream` and `FileOutputStream` read from and write to files. `ByteArrayInputStream` and `ByteArrayOutputStream` read from and write to an array of bytes in memory. `PipedInputStream` reads bytes from a `PipedOutputStream`, and `PipedOutputStream` writes bytes to a `PipedInputStream`. These classes work together to implement a *pipe* for communication between threads.

`FilterInputStream` and `FilterOutputStream` are special; they filter input and output bytes. When you create a `FilterInputStream`, you specify an `InputStream` for it to filter. When you call the `read()` method of a `FilterInputStream`, it calls the `read()` method of its `InputStream`, processes the bytes it reads, and returns the filtered bytes. Similarly, when you create a `FilterOutputStream`, you specify an `OutputStream` to be filtered. Calling the `write()` method of

Package java.io

a `FilterOutputStream` causes it to process your bytes in some way and then pass those filtered bytes to the `write()` method of its `OutputStream`.

`FilterInputStream` and `FilterOutputStream` do not perform any filtering themselves; this is done by their subclasses. `BufferedInputStream` and `BufferedOutputStream` are filtered streams that provide input and output buffering and can increase I/O efficiency. `DataInputStream` reads raw bytes from a stream and interprets them in various binary formats. It has various methods to read primitive Java data types in their standard binary formats. `DataOutputStream` allows you to write Java primitive data types in binary format.

The byte streams I just described are complemented by an analogous set of character input and output streams. `Reader` is the superclass of all character input streams, and `Writer` is the superclass of all character output streams. Most of the `Reader` and `Writer` streams have obvious byte-stream analogs. `BufferedReader` is a commonly used stream; it provides buffering for efficiency and also has a `readLine()` method to read a line of text at a time. `PrintWriter` is another very common stream; its methods allow output of a textual representation of any primitive Java type or of any object (via the object's `toString()` method).

The `ObjectInputStream` and `ObjectOutputStream` classes are special. These byte-stream classes are used for serializing and deserializing the internal state of objects for storage or interprocess communication.

Interfaces:

```
public interface DataInput;  
public interface DataOutput;  
public interface Externalizable extends Serializable;  
public interface FileFilter;  
public interface FilenameFilter;  
public interface ObjectInput extends DataInput;  
public interface ObjectInputValidation;  
public interface ObjectOutput extends DataOutput;  
public interface ObjectStreamConstants;  
public interface Serializable;
```

Classes:

```
public class File implements Comparable, Serializable;  
public final class FileDescriptor;  
public final class FilePermission extends java.security.Permission implements Serializable;  
public abstract class InputStream;  
    public class ByteArrayInputStream extends InputStream;  
    public class FileInputStream extends InputStream;  
    public class FilterInputStream extends InputStream;  
        public class BufferedInputStream extends FilterInputStream;  
        public class DataInputStream extends FilterInputStream implements DataInput;  
        public class LineNumberInputStream extends FilterInputStream;  
        public class PushbackInputStream extends FilterInputStream;  
    public class ObjectInputStream extends InputStream  
    implements ObjectInput, ObjectStreamConstants;  
    public class PipedInputStream extends InputStream;  
    public class SequenceInputStream extends InputStream;  
    public class StringBufferInputStream extends InputStream;  
public abstract static class ObjectInputStream.GetField;  
public abstract static class ObjectOutputStream.PutField;
```

```
public class ObjectOutputStream implements Serializable;
public class ObjectOutputStreamField implements Comparable;
public abstract class OutputStream;
    public class ByteArrayOutputStream extends OutputStream;
    public class FileOutputStream extends OutputStream;
    public class FilterOutputStream extends OutputStream;
        public class BufferedOutputStream extends FilterOutputStream;
        public class DataOutputStream extends FilterOutputStream implements DataOutput;
        public class PrintStream extends FilterOutputStream;
    public class ObjectOutputStream extends OutputStream
        implements ObjectOutput, ObjectOutputStreamConstants;
    public class PipedOutputStream extends OutputStream;
public class RandomAccessFile implements DataInput, DataOutput;
public abstract class Reader;
    public class BufferedReader extends Reader;
        public class LineNumberReader extends BufferedReader;
    public class CharArrayReader extends Reader;
    public abstract class FilterReader extends Reader;
        public class PushbackReader extends FilterReader;
    public class InputStreamReader extends Reader;
        public class FileReader extends InputStreamReader;
    public class PipedReader extends Reader;
    public class StringReader extends Reader;
public final class SerializablePermission extends java.security.BasicPermission;
public class StreamTokenizer;
public abstract class Writer;
    public class BufferedWriter extends Writer;
    public class CharArrayWriter extends Writer;
    public abstract class FilterWriter extends Writer;
    public class OutputStreamWriter extends Writer;
        public class FileWriter extends OutputStreamWriter;
    public class PipedWriter extends Writer;
    public class PrintWriter extends Writer;
    public class StringWriter extends Writer;
```

Exceptions:

```
public class IOException extends Exception;
    public class CharConversionException extends IOException;
    public class EOFException extends IOException;
    public class FileNotFoundException extends IOException;
    public class InterruptedIOException extends IOException;
    public abstract class ObjectStreamException extends IOException;
        public class InvalidClassException extends ObjectStreamException;
        public class InvalidObjectException extends ObjectStreamException;
        public class NotActiveException extends ObjectStreamException;
        public class NotSerializableException extends ObjectStreamException;
        public class OptionalDataException extends ObjectStreamException;
        public class StreamCorruptedException extends ObjectStreamException;
        public class WriteAbortedException extends ObjectStreamException;
    public class SyncFailedException extends IOException;
    public class UnsupportedEncodingException extends IOException;
    public class UTFDataFormatException extends IOException;
```

BufferedInputStream

BufferedInputStream

Java 1.0

java.io

This class is a `FilterInputStream` that provides input data buffering; efficiency is increased by reading in a large amount of data and storing it in an internal buffer. When data is requested, it is usually available from the buffer. Thus, most calls to read data do not actually have to read data from a disk, network, or other slow source. Create a `BufferedInputStream` by specifying the `InputStream` that is to be buffered in the call to the constructor. See also `BufferedReader`.

```
Object — InputStream — FilterInputStream — BufferedInputStream

public class BufferedInputStream extends FilterInputStream {
// Public Constructors
    public BufferedInputStream(java.io.InputStream in);
    public BufferedInputStream(java.io.InputStream in, int size);
// Public Methods Overriding FilterInputStream
    public int available() throws IOException;                                synchronized
1.2 public void close() throws IOException;
    public void mark(int readlimit);                                        synchronized
    public boolean markSupported();                                        constant
    public int read() throws IOException;                                synchronized
    public int read(byte[] b, int off, int len) throws IOException;        synchronized
    public void reset() throws IOException;                                synchronized
    public long skip(long n) throws IOException;                          synchronized
// Protected Instance Fields
    protected byte[] buf;
    protected int count;
    protected int marklimit;
    protected int markpos;
    protected int pos;
}
```

BufferedOutputStream

Java 1.0

java.io

This class is a `FilterOutputStream` that provides output data buffering; output efficiency is increased by storing values to be written in a buffer and actually writing them out only when the buffer fills up or when the `flush()` method is called. Create a `BufferedOutputStream` by specifying the `OutputStream` that is to be buffered in the call to the constructor. See also `BufferedWriter`.

```
Object — OutputStream — FilterOutputStream — BufferedOutputStream

public class BufferedOutputStream extends FilterOutputStream {
// Public Constructors
    public BufferedOutputStream(java.io.OutputStream out);
    public BufferedOutputStream(java.io.OutputStream out, int size);
// Public Methods Overriding FilterOutputStream
    public void flush() throws IOException;                                synchronized
    public void write(int b) throws IOException;                          synchronized
    public void write(byte[] b, int off, int len) throws IOException;      synchronized
// Protected Instance Fields
    protected byte[] buf;
    protected int count;
}
```

BufferedReader

Java 1.1

java.io

This class applies buffering to a character input stream, thereby improving the efficiency of character input. You create a **BufferedReader** by specifying some other character input stream from which it is to buffer input. (You can also specify a buffer size at this time, although the default size is usually fine.) Typically, you use this sort of buffering with a **FileReader** or **InputStreamReader**. **BufferedReader** defines the standard set of **Reader** methods and provides a **readLine()** method that reads a line of text (not including the line terminator) and returns it as a **String**. **BufferedReader** is the character-stream analog of **BufferedInputStream**. It also provides a replacement for the deprecated **readLine()** method of **DataInputStream**, which did not properly convert bytes into characters.

Object	Reader	BufferedReader
--------	--------	----------------

```

public class BufferedReader extends Reader {
// Public Constructors
    public BufferedReader(Reader in);
    public BufferedReader(Reader in, int sz);
// Public Instance Methods
    public String readLine() throws IOException;
// Public Methods Overriding Reader
    public void close() throws IOException;
    public void mark(int readAheadLimit) throws IOException;
    public boolean markSupported(); constant
    public int read() throws IOException;
    public int read(char[] cbuf, int off, int len) throws IOException;
    public boolean ready() throws IOException;
    public void reset() throws IOException;
    public long skip(long n) throws IOException;
}

```

Subclasses: **LineNumberReader****BufferedWriter**

Java 1.1

java.io

This class applies buffering to a character output stream, improving output efficiency by coalescing many small write requests into a single larger request. You create a **BufferedWriter** by specifying some other character output stream to which it sends its buffered and coalesced output. (You can also specify a buffer size at this time, although the default size is usually satisfactory.) Typically, you use this sort of buffering with a **FileWriter** or **OutputStreamWriter**. **BufferedWriter** defines the standard **write()**, **flush()**, and **close()** methods all output streams define, but it adds a **newLine()** method that outputs the platform-dependent line separator (usually a newline character, a carriage-return character, or both) to the stream. **BufferedWriter** is the character-stream analog of **BufferedOutputStream**.

Object	Writer	BufferedWriter
--------	--------	----------------

```

public class BufferedWriter extends Writer {
// Public Constructors
    public BufferedWriter(Writer out);
    public BufferedWriter(Writer out, int sz);
// Public Instance Methods
    public void newLine() throws IOException;
}

```

BufferedWriter

```
// Public Methods Overriding Writer
public void close() throws IOException;
public void flush() throws IOException;
public void write(int c) throws IOException;
public void write(char[] cbuf, int off, int len) throws IOException;
public void write(String s, int off, int len) throws IOException;
}
```

ByteArrayInputStream

Java 1.0

java.io

This class is a subclass of `InputStream` in which input data comes from a specified array of `byte` values. This is useful when you want to read data in memory as if it were coming from a file, pipe, or socket. Note that the specified array of bytes is not copied when a `ByteArrayInputStream` is created. See also `CharArrayReader`.

Object	InputStream	ByteArrayInputStream
--------	-------------	----------------------

```
public class ByteArrayInputStream extends java.io.InputStream {
// Public Constructors
    public ByteArrayInputStream(byte[] buf);
    public ByteArrayInputStream(byte[] buf, int offset, int length);
// Public Methods Overriding InputStream
    public int available(); synchronized
    1.2 public void close() throws IOException; empty
    1.1 public void mark(int readAheadLimit);
    1.1 public boolean markSupported(); constant
    public int read(); synchronized
    public int read(byte[] b, int off, int len); synchronized
    public void reset(); synchronized
    public long skip(long n); synchronized
// Protected Instance Fields
    protected byte[] buf;
    protected int count;
    1.1 protected int mark;
    protected int pos;
}
```

ByteArrayOutputStream

Java 1.0

java.io

This class is a subclass of `OutputStream` in which output data is stored in an internal `byte` array. The internal array grows as necessary and can be retrieved with `toByteArray()` or `toString()`. The `reset()` method discards any data currently stored in the internal array and stores data from the beginning again. See also `CharArrayWriter`.

Object	OutputStream	ByteArrayOutputStream
--------	--------------	-----------------------

```
public class ByteArrayOutputStream extends java.io.OutputStream {
// Public Constructors
    public ByteArrayOutputStream();
    public ByteArrayOutputStream(int size);
// Public Instance Methods
    public void reset(); synchronized
    public int size();
}
```

CharArrayWriter

```
public byte[] toByteArray(); synchronized
1.1 public String toString(String enc) throws UnsupportedOperationException;
public void writeTo(java.io.OutputStream out) throws IOException; synchronized
// Public Methods Overriding OutputStream
1.2 public void close() throws IOException; empty
public void write(int b); synchronized
public void write(byte[] b, int off, int len); synchronized
// Public Methods Overriding Object
public String toString();
// Protected Instance Fields
protected byte[] buf;
protected int count;
// Deprecated Public Methods
# public String toString(int hbyte);
}
```

CharArrayReader

Java 1.1

java.io

This class is a character input stream that uses a character array as the source of the characters it returns. You create a `CharArrayReader` by specifying the character array (or portion of an array) it is to read from. `CharArrayReader` defines the usual `Reader` methods and supports the `mark()` and `reset()` methods. Note that the character array you pass to the `CharArrayReader()` constructor is not copied. This means that changes you make to the elements of the array after you create the input stream affect the values read from the array. `CharArrayReader` is the character-array analog of `ByteArrayInputStream` and is similar to `StringReader`.

Object	Reader	CharArrayReader
--------	--------	-----------------

```
public class CharArrayReader extends Reader {
// Public Constructors
public CharArrayReader(char[] buf);
public CharArrayReader(char[] buf, int offset, int length);
// Public Methods Overriding Reader
public void close();
public void mark(int readAheadLimit) throws IOException;
public boolean markSupported(); constant
public int read() throws IOException;
public int read(char[] b, int off, int len) throws IOException;
public boolean ready() throws IOException;
public void reset() throws IOException;
public long skip(long n) throws IOException;
// Protected Instance Fields
protected char[] buf;
protected int count;
protected int markedPos;
protected int pos;
}
```

CharArrayWriter

Java 1.1

java.io

This class is a character output stream that uses an internal character array as the destination of characters written to it. When you create a `CharArrayWriter`, you may optionally specify an initial size for the character array, but you do not specify the character array

CharArrayWriter

itself; this array is managed internally by the `CharArrayWriter` and grows as necessary to accommodate all the characters written to it. The `toString()` and `toCharArray()` methods return a copy of all characters written to the stream, as a string and an array of characters, respectively. `CharArrayWriter` defines the standard `write()`, `flush()`, and `close()` methods all `Writer` subclasses define. It also defines a few other useful methods. `size()` returns the number of characters that have been written to the stream. `reset()` resets the stream to its initial state, with an empty character array; this is more efficient than creating a new `CharArrayWriter`. Finally, `writeTo()` writes the contents of the internal character array to some other specified character stream. `CharArrayWriter` is the character-stream analog of `ByteArrayOutputStream` and is quite similar to `StringWriter`.

```
Object -- Writer -- CharArrayWriter

public class CharArrayWriter extends Writer {
// Public Constructors
    public CharArrayWriter();
    public CharArrayWriter(int initialSize);
// Public Instance Methods
    public void reset();
    public int size();
    public char[] toCharArray();
    public void writeTo(Writer out) throws IOException;
// Public Methods Overriding Writer
    public void close(); // empty
    public void flush(); // empty
    public void write(int c);
    public void write(char[] c, int off, int len);
    public void write(String str, int off, int len);
// Public Methods Overriding Object
    public String toString();
// Protected Instance Fields
    protected char[] buf;
    protected int count;
}
```

CharConversionException

Java 1.1

java.io

serializable checked

A `CharConversionException` signals an error when converting bytes to characters or vice versa.

```
Object -- Throwable -- Exception -- IOException -- CharConversionException
      |
      +-- Serializable

public class CharConversionException extends IOException {
// Public Constructors
    public CharConversionException();
    public CharConversionException(String s);
}
```

DataInput

Java 1.0

java.io

This interface defines the methods required for streams that can read Java primitive data types in a machine-independent binary format. It is implemented by `DataInputStream` and `RandomAccessFile`. See `DataInputStream` for more information on the methods.


```
public interface DataInput {
// Public Instance Methods
    public abstract boolean readBoolean() throws IOException;
    public abstract byte readByte() throws IOException;
    public abstract char readChar() throws IOException;
    public abstract double readDouble() throws IOException;
    public abstract float readFloat() throws IOException;
    public abstract void readFully(byte[] b) throws IOException;
    public abstract void readFully(byte[] b, int off, int len) throws IOException;
    public abstract int readInt() throws IOException;
    public abstract String readLine() throws IOException;
    public abstract long readLong() throws IOException;
    public abstract short readShort() throws IOException;
    public abstract int readUnsignedByte() throws IOException;
    public abstract int readUnsignedShort() throws IOException;
    public abstract String readUTF() throws IOException;
    public abstract int skipBytes(int n) throws IOException;
}
```

Implementations: java.io.DataInputStream, ObjectInput, RandomAccessFile,
javax.imageio.stream.ImageInputStream

Passed To: java.io.DataInputStream.readUTF(), java.rmi.server.UID.read()

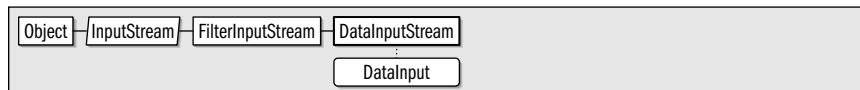
DataInputStream

Java 1.0

java.io

This class is a type of `FilterInputStream` that allows you to read binary representations of Java primitive data types in a portable way. Create a `DataInputStream` by specifying the `InputStream` that is to be filtered in the call to the constructor. `DataInputStream` reads only primitive Java types; use `ObjectInputStream` to read object values.

Many of the methods read and return a single Java primitive type, in binary format, from the stream. `readUnsignedByte()` and `readUnsignedShort()` read unsigned values and return them as int values, since unsigned `byte` and `short` types are not supported in Java. `read()` reads data into an array of bytes, blocking until at least some data is available. By contrast, `readFully()` reads data into an array of bytes, but blocks until all requested data becomes available. `skipBytes()` blocks until the specified number of bytes have been read and discarded. `readLine()` reads characters from the stream until it encounters a newline, a carriage return, or a newline/carriage return pair. The returned string is not terminated with a newline or carriage return. This method is deprecated as of Java 1.1; see `BufferedReader` for an alternative. `readUTF()` reads a string of Unicode text encoded in a slightly modified version of the UTF-8 transformation format. UTF-8 is an ASCII-compatible encoding of Unicode characters that is often used for the transmission and storage of Unicode text. This class uses a modified UTF-8 encoding that never contains embedded null characters.



```
public class DataInputStream extends FilterInputStream implements DataInput {
// Public Constructors
    public DataInputStream(java.io.InputStream in);
// Public Class Methods
    public static final String readUTF(DataInput in) throws IOException;
```

DataInputStream

```
// Methods Implementing DataInput
public final boolean readBoolean() throws IOException;
public final byte readByte() throws IOException;
public final char readChar() throws IOException;
public final double readDouble() throws IOException;
public final float readFloat() throws IOException;
public final void readFully(byte[ ] b) throws IOException;
public final void readFully(byte[ ] b, int off, int len) throws IOException;
public final int readInt() throws IOException;
public final long readLong() throws IOException;
public final short readShort() throws IOException;
public final int readUnsignedByte() throws IOException;
public final int readUnsignedShort() throws IOException;
public final String readUTF() throws IOException;
public final int skipBytes(int n) throws IOException;
// Public Methods Overriding FilterInputStream
public final int read(byte[ ] b) throws IOException;
public final int read(byte[ ] b, int off, int len) throws IOException;
// Deprecated Public Methods
# public final String readLine() throws IOException; Implements:DataInput
}
```

Passed To: javax.swing.text.html.parser.DTD.read()

DataOutput

Java 1.0

java.io

This interface defines the methods required for streams that can write Java primitive data types in a machine-independent binary format. It is implemented by `DataOutputStream` and `RandomAccessFile`. See `DataOutputStream` for more information on the methods.

```
public interface DataOutput {
// Public Instance Methods
public abstract void write(byte[ ] b) throws IOException;
public abstract void write(int b) throws IOException;
public abstract void write(byte[ ] b, int off, int len) throws IOException;
public abstract void writeBoolean(boolean v) throws IOException;
public abstract void writeByte(int v) throws IOException;
public abstract void writeBytes(String s) throws IOException;
public abstract void writeChar(int v) throws IOException;
public abstract void writeChars(String s) throws IOException;
public abstract void writeDouble(double v) throws IOException;
public abstract void writeFloat(float v) throws IOException;
public abstract void writeInt(int v) throws IOException;
public abstract void writeLong(long v) throws IOException;
public abstract void writeShort(int v) throws IOException;
public abstract void writeUTF(String str) throws IOException;
}
```

Implementations: java.io.DataOutputStream, ObjectOutput, RandomAccessFile,
javax.imageio.stream.ImageOutputStream

Passed To: java.rmi.server.UID.write()

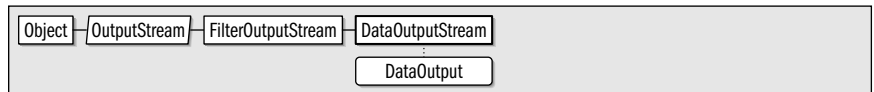
DataOutputStream

Java 1.0

java.io

This class is a subclass of `FilterOutputStream` that allows you to write Java primitive data types in a portable binary format. Create a `DataOutputStream` by specifying the `OutputStream` that is to be filtered in the call to the constructor. `DataOutputStream` has methods that output only primitive types; use `ObjectOutputStream` to output object values.

Many of this class's methods write a single Java primitive type, in binary format, to the output stream. `write()` writes a single byte, an array, or a subarray of bytes. `flush()` forces any buffered data to be output. `size()` returns the number of bytes written so far. `writeUTF()` outputs a Java string of Unicode characters using a slightly modified version of the UTF-8 transformation format. UTF-8 is an ASCII-compatible encoding of Unicode characters that is often used for the transmission and storage of Unicode text. Except for the `writeUTF()` method, this class is used for binary output of data. Textual output should be done with `PrintWriter` (or `PrintStream` in Java 1.0).



```

public class DataOutputStream extends FilterOutputStream implements DataOutput {
    // Public Constructors
    public DataOutputStream(java.io.OutputStream out);
    // Public Instance Methods
    public final int size();
    // Methods Implementing DataOutput
    public void write(int b) throws IOException;           synchronized
    public void write(byte[] b, int off, int len) throws IOException;   synchronized
    public final void writeBoolean(boolean v) throws IOException;
    public final void writeByte(int v) throws IOException;
    public final void writeBytes(String s) throws IOException;
    public final void writeChar(int v) throws IOException;
    public final void writeChars(String s) throws IOException;
    public final void writeDouble(double v) throws IOException;
    public final void writeFloat(float v) throws IOException;
    public final void writeInt(int v) throws IOException;
    public final void writeLong(long v) throws IOException;
    public final void writeShort(int v) throws IOException;
    public final void writeUTF(String str) throws IOException;
    // Public Methods Overriding FilterOutputStream
    public void flush() throws IOException;
    // Protected Instance Fields
    protected int written;
}
  
```

EOFException

Java 1.0

java.io

serializable checked

An `EOFException` is an `IOException` that signals the end of file.



```

public class EOFException extends IOException {
    // Public Constructors
  
```

EOFException

```
public EOFException();  
public EOFException(String s);  
}
```

Externalizable

Java 1.1

java.io

serializable

This interface defines the methods that must be implemented by an object that wants complete control over the way it is serialized. The `writeExternal()` and `readExternal()` methods should be implemented to write and read object data in some arbitrary format, using the methods of the `DataOutput` and `DataInput` interfaces. `Externalizable` objects must serialize their own fields and are also responsible for serializing the fields of their superclasses. Most objects do not need to define a custom output format and can use the `Serializable` interface instead of `Externalizable` for serialization.

Serializable Externalizable

```
public interface Externalizable extends Serializable {  
    // Public Instance Methods  
    public abstract void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;  
    public abstract void writeExternal(ObjectOutput out) throws IOException;  
}
```

Implementations: `java.awt.datatransfer.DataFlavor`, `java.rmi.server.RemoteRef`

File

Java 1.0

java.io

serializable comparable

This class supports a platform-independent definition of file and directory names. It also provides methods to list the files in a directory; check the existence, readability, writability, type, size, and modification time of files and directories; make new directories; rename files and directories; delete files and directories; and create and delete temporary and lock files. The constants defined by this class are the platform-dependent directory and path-separator characters, available as a `String` and a `char`.

`getName()` returns the name of the `File` with any directory names omitted. `getPath()` returns the full name of the file, including the directory name. `getParent()` and `getParentFile()` return the directory that contains the `File`; the only difference between the two methods is that one returns a `String`, while the other returns a `File`. `isAbsolute()` tests whether the `File` is an absolute specification. If not, `getAbsolutePath()` returns an absolute filename created by appending the relative filename to the current working directory. `getAbsoluteFile()` returns the equivalent absolute `File` object. `getCanonicalPath()` and `getCanonicalFile()` are similar methods: they return an absolute filename or `File` object that has been converted to its system-dependent canonical form. This can be useful when comparing two `File` objects to see if they refer to the same file or directory. In Java 1.4 and later, the `toURI()` method returns a `java.net.URI` object that uses a `file:` scheme to name this file. This file-to-URI transformation can be reversed by passing a `file:` URI object to the `File()` constructor.

`exists()`, `canWrite()`, `canRead()`, `isFile()`, `isDirectory()`, and `isHidden()` perform the obvious tests on the specified `File`. `length()` returns the length of the file. `lastModified()` returns the modification time of the file (which should be used for comparison with other file times only and not interpreted as any particular time format). `setLastModified()` allows the modification time to be set; `setReadOnly()` makes a file or directory read-only.

`list()` returns the names of all entries in a directory that are not rejected by an optional `FilenameFilter`. `listFiles()` returns an array of `File` objects that represent all entries in a

directory not rejected by an optional `FilenameFilter` or `FileFilter`. `listRoots()` returns an array of `File` objects representing all root directories on the system. On Unix systems, for example, there is typically only one root, `/`. On Windows systems, however, there is a different root for each drive letter: `c:\`, `d:\`, and `e:\`, for example.

`mkdir()` creates a directory, and `mkdirs()` creates all the directories in a `File` specification. `renameTo()` renames a file or directory; `delete()` deletes a file or directory. Prior to Java 1.2, the `File` class doesn't provide any way to create a file; that task is accomplished typically with `FileOutputStream`. As of Java 1.2, however, two special-purpose file creation methods have been added. The static `createTempFile()` method returns a `File` object that refers to a newly created empty file with a unique name that begins with the specified prefix (which must be at least three characters long) and ends with the specified suffix. One version of this method creates the file in a specified directory, and the other creates it in the system temporary directory. Applications can use temporary files for any purpose without worrying about overwriting files belonging to other applications. The other file-creation method of Java 1.2 is `createNewFile()`. This instance method attempts to create a new, empty file with the name specified by the `File` object. If it succeeds, it returns `true`. However, if the file already exists, it returns `false`. `createNewFile()` works atomically, and is therefore useful for file locking and other mutual-exclusion schemes. When working with `createTempFile()` or `createNewFile()`, consider using `deleteOnExit()` to request that the files be deleted when the Java VM exits normally.



```

public class File implements Comparable, Serializable {
    // Public Constructors
    public File(String pathname);
    1.4 public File(java.net.URL uri);
    public File(String parent, String child);
    public File(File parent, String child);
    // Public Constants
    public static final String pathSeparator;
    public static final char pathSeparatorChar;
    public static final String separator;
    public static final char separatorChar;
    // Public Class Methods
    1.2 public static File createTempFile(String prefix, String suffix) throws IOException;
    1.2 public static File createTempFile(String prefix, String suffix, File directory) throws IOException;
    1.2 public static File[] listRoots();
    // Property Accessor Methods (by property name)
    public boolean isAbsolute();
    1.2 public File getAbsoluteFile();
    public String getAbsolutePath();
    1.2 public File getCanonicalFile() throws IOException;
    1.1 public String getCanonicalPath() throws IOException;
    public boolean isDirectory();
    public boolean isFile();
    1.2 public boolean isHidden();
    public String getName();
    public String getParent();
    1.2 public File getParentFile();
    public String getPath();
    // Public Instance Methods
    public boolean canRead();
  
```

File

```
public boolean canWrite();
1.2 public int compareTo(File pathname);
1.2 public boolean createNewFile() throws IOException;
public boolean delete();
1.2 public void deleteOnExit();
public boolean exists();
public long lastModified();
public long length();
public String[] list();
public String[] list(FilenameFilter filter);
1.2 public File[] listFiles();
1.2 public File[] listFiles(FilenameFilter filter);
1.2 public File[] listFiles(java.io.FileFilter filter);
public boolean mkdir();
public boolean mkdirs();
public boolean renameTo(File dest);
1.2 public boolean setLastModified(long time);
1.2 public boolean setReadOnly();
1.4 public java.net.URI toURI();
1.2 public java.net.URL toURL() throws java.net.MalformedURLException;
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// Public Methods Overriding Object
public boolean equals(Object obj);
public int hashCode();
public String toString();
}
```

Passed To: Too many methods to list.

Returned By: Too many methods to list.

FileDescriptor

Java 1.0

java.io

This class is a platform-independent representation of a low-level handle to an open file or socket. The static `in`, `out`, and `err` variables are `FileDescriptor` objects that represent the standard input, output, and error streams, respectively. There is no public constructor method to create a `FileDescriptor` object. You can obtain one with the `getFD()` method of `FileInputStream`, `FileOutputStream`, or `RandomAccessFile`.

```
public final class FileDescriptor {
// Public Constructors
public FileDescriptor();
// Public Constants
public static final FileDescriptor err;
public static final FileDescriptor in;
public static final FileDescriptor out;
// Public Instance Methods
1.1 public void sync() throws SyncFailedException; native
public boolean valid();
}
```

Passed To: `FileInputStream.FileInputStream()`, `FileOutputStream.FileOutputStream()`, `FileReader.FileReader()`, `FileWriter.FileWriter()`, `SecurityManager.{checkRead(), checkWrite()}`

Returned By: `FileInputStream.getFD()`, `FileOutputStream.getFD()`, `RandomAccessFile.getFD()`, `java.net.DatagramSocketImpl.getFileDescriptor()`, `java.net.SocketImpl.getFileDescriptor()`

Type Of: `FileDescriptor.{err, in, out}`, `java.net.DatagramSocketImpl.fd`, `java.net.SocketImpl.fd`

FileFilter

Java 1.2

java.io

This interface defines an `accept()` method that filters a list of files. You can list the contents of a directory by calling the `listFiles()` method of the `File` object that represents the desired directory. If you want a filtered listing, such as a listing of files but not subdirectories or a listing of files whose names end in `.class`, you can pass a `FileFilter` object to `listFiles()`. For each entry in the directory, a `File` object is passed to the `accept()` method. If `accept()` returns `true`, that `File` is included in the return value of `listFiles()`. If `accept()` returns `false`, that entry is not included in the listing. `FileFilter` is new in Java 1.2. Use `FilenameFilter` if compatibility with previous releases of Java is required or if you prefer to filter filenames (i.e., `String` objects) rather than `File` objects.

```
public interface FileFilter {
    // Public Instance Methods
    public abstract boolean accept(File pathname);
}
```

Passed To: `File.listFiles()`

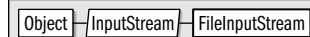
FileInputStream

Java 1.0

java.io

This class is a subclass of `InputStream` that reads bytes from a file specified by name or by a `File` or `FileDescriptor` object. `read()` reads a byte or array of bytes from the file. It returns `-1` when the end-of-file has been reached. To read binary data, you typically use this class in conjunction with a `BufferedInputStream` and `DataInputStream`. To read text, you typically use it with an `InputStreamReader` and `BufferedReader`. Call `close()` to close the file when input is no longer needed.

In Java 1.4 and later, use `getChannel()` to obtain a `FileChannel` object for reading from the underlying file using the New I/O API of `java.nio` and its subpackages.



```
public class FileInputStream extends java.io.InputStream {
    // Public Constructors
    public FileInputStream(String name) throws FileNotFoundException;
    public FileInputStream(File file) throws FileNotFoundException;
    public FileInputStream(FileDescriptor fdObj);
    // Public Instance Methods
    1.4 public java.nio.channels.FileChannel getChannel();
    public final FileDescriptor getFD() throws IOException;
    // Public Methods Overriding InputStream
    public int available() throws IOException; native
    public void close() throws IOException;
    public int read() throws IOException; native
    public int read(byte[] b) throws IOException;
    public int read(byte[] b, int off, int len) throws IOException;
    public long skip(long n) throws IOException; native
}
```

FileInputStream

```
// Protected Methods Overriding Object
protected void finalize() throws IOException;
}
```

FilenameFilter

Java 1.0

java.io

This interface defines the `accept()` method that must be implemented by any object that filters filenames (i.e., selects a subset of filenames from a list of filenames). There are no standard `FilenameFilter` classes implemented by Java, but objects that implement this interface are used by the `java.awt.FileDialog` object and the `File.list()` method. A typical `FilenameFilter` object might check that the specified `File` represents a file (not a directory), is readable (and possibly writable as well), and that its name ends with some desired extension.

```
public interface FilenameFilter {
// Public Instance Methods
    public abstract boolean accept(File dir, String name);
}
```

Passed To: `java.awt.FileDialog.setFilenameFilter()`, `java.awt.peer.FileDialogPeer.setFilenameFilter()`, `File.{list(), listFiles()}`

Returned By: `java.awt.FileDialog.getFilenameFilter()`

FileNotFoundException

Java 1.0

java.io

serializable checked

A `FileNotFoundException` is an `IOException` that signals that a specified file cannot be found.



```
public class FileNotFoundException extends IOException {
// Public Constructors
    public FileNotFoundException();
    public FileNotFoundException(String s);
}
```

Thrown By: `FileInputStream.FileInputStream()`, `FileOutputStream.FileOutputStream()`, `FileReader.FileReader()`, `RandomAccessFile.RandomAccessFile()`, `javax.imageio.stream.FileImageInputStream.FileImageInputStream()`, `javax.imageio.stream.FileImageOutputStream.FileImageOutputStream()`


FileOutputStream

Java 1.0

java.io

This class is a subclass of `OutputStream` that writes data to a file specified by name or by a `File` or `FileDescriptor` object. If the specified file already exists, a `FileOutputStream` can be configured to overwrite or append to the existing file. `write()` writes a byte or array of bytes to the file. To write binary data, you typically use this class in conjunction with a `BufferedOutputStream` and a `DataOutputStream`. To write text, you typically use it with a `PrintWriter`, `BufferedWriter` and an `OutputStreamWriter` (or you use the convenience class `FileWriter`). Use `close()` to close a `FileOutputStream` when no further output will be written to it.

In Java 1.4 and later, use `getChannel()` to obtain a `FileChannel` object for writing to the underlying file using the New I/O API of `java.nio` and its subpackages.



```

public class FileOutputStream extends java.io.OutputStream {
    // Public Constructors
    public FileOutputStream(FileDescriptor fdObj);
    public FileOutputStream(File file) throws FileNotFoundException;
    public FileOutputStream(String name) throws FileNotFoundException;
    1.1 public FileOutputStream(String name, boolean append) throws FileNotFoundException;
    1.4 public FileOutputStream(File file, boolean append) throws FileNotFoundException;
    // Public Instance Methods
    1.4 public java.nio.channels.FileChannel getChannel();
    public final FileDescriptor getFD() throws IOException;
    // Public Methods Overriding OutputStream
    public void close() throws IOException;
    public void write(int b) throws IOException; native
    public void write(byte[] b) throws IOException;
    public void write(byte[] b, int off, int len) throws IOException;
    // Protected Methods Overriding Object
    protected void finalize() throws IOException;
}

```

FilePermission

Java 1.2

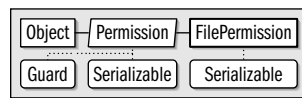
java.io

serializable permission

This class is a `java.security.Permission` that governs access to the local filesystem. A `FilePermission` has a name, or target, which specifies what file or files it pertains to, and a comma-separated list of actions that may be performed on the file or files. The supported actions are read, write, delete, and execute. Read and write permission are required by any methods that read or write a file. Delete permission is required by `File.delete()`, and execute permission is required by `Runtime.exec()`.

The name of a `FilePermission` may be as simple as a file or directory name. `FilePermission` also supports the use of certain wildcards, however, to specify a permission that applies to more than one file. If the name of the `FilePermission` is a directory name followed by `/*` (`*` on Windows platforms), it specifies all files in the named directory. If the name is a directory name followed by `/-` (`\-` on Windows), it specifies all files in the directory, and, recursively, all files in all subdirectories. A `*` alone specifies all files in the current directory, and a `-` alone specifies all files in or beneath the current directory. Finally, the special name `<<ALL FILES>>` matches all files anywhere in the filesystem.

Applications do not need to use this class directly. Programmers writing system-level code and system administrators configuring security policies may need to use it, however. Be very careful when granting any types of `FilePermission`. Restricting access (especially write access) to files is one of the cornerstones of the Java security model with regard to untrusted code.



```

public final class FilePermission extends java.security.Permission implements Serializable {
    // Public Constructors
    public FilePermission(String path, String actions);
    // Public Methods Overriding Permission
}

```

FilePermission

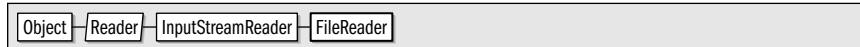
```
public boolean equals(Object obj);
public String getActions();
public int hashCode();
public boolean implies(java.security.Permission p);
public java.security.PermissionCollection newPermissionCollection();
}
```

FileReader

Java 1.1

java.io

`FileReader` is a convenience subclass of `InputStreamReader` that is useful when you want to read text (as opposed to binary data) from a file. You create a `FileReader` by specifying the file to be read in any of three possible forms. The `FileReader` constructor internally creates a `FileInputStream` to read bytes from the specified file and uses the functionality of its superclass, `InputStreamReader`, to convert those bytes from characters in the local encoding to the Unicode characters used by Java. Because `FileReader` is a trivial subclass of `InputStreamReader`, it does not define any `read()` methods or other methods of its own. Instead, it inherits all its methods from its superclass. If you want to read Unicode characters from a file that uses some encoding other than the default encoding for the locale, you must explicitly create your own `InputStreamReader` to perform the byte-to-character conversion.



```
public class FileReader extends InputStreamReader {
// Public Constructors
    public FileReader(FileDescriptor fd);
    public FileReader(File file) throws FileNotFoundException;
    public FileReader(String fileName) throws FileNotFoundException;
}
```

FileWriter

Java 1.1

java.io

`FileWriter` is a convenience subclass of `OutputStreamWriter` that is useful when you want to write text (as opposed to binary data) to a file. You create a `FileWriter` by specifying the file to be written to and, optionally, whether the data should be appended to the end of an existing file instead of overwriting that file. The `FileWriter` class creates an internal `FileOutputStream` to write bytes to the specified file and uses the functionality of its superclass, `OutputStreamWriter`, to convert the Unicode characters written to the stream into bytes using the default encoding of the default locale. (If you want to use an encoding other than the default, you cannot use `FileWriter`; in that case you must create your own `OutputStreamWriter` and `FileOutputStream`.) Because `FileWriter` is a trivial subclass of `OutputStreamWriter`, it does not define any methods of its own, but simply inherits them from its superclass.



```
public class FileWriter extends OutputStreamWriter {
// Public Constructors
    public FileWriter(File file) throws IOException;
    public FileWriter(FileDescriptor fd);
    public FileWriter(String fileName) throws IOException;
    1.4 public FileWriter(File file, boolean append) throws IOException;
```

```
    public FileWriter(String fileName, boolean append) throws IOException;
}
```

FilterInputStream

Java 1.0

java.io

This class provides method definitions required to filter data obtained from the `InputStream` specified when the `FilterInputStream` is created. It must be subclassed to perform some sort of filtering operation and cannot be instantiated directly. See the subclasses `BufferedInputStream`, `DataInputStream`, and `PushbackInputStream`.



```
public class FilterInputStream extends java.io.InputStream {
    // Protected Constructors
    protected FilterInputStream(java.io.InputStream in);
    // Public Methods Overriding InputStream
    public int available() throws IOException;
    public void close() throws IOException;
    public void mark(int readlimit);
    public boolean markSupported();
    public int read() throws IOException;
    public int read(byte[] b) throws IOException;
    public int read(byte[] b, int off, int len) throws IOException;
    public void reset() throws IOException;
    public long skip(long n) throws IOException;
    // Protected Instance Fields
    protected java.io.InputStream in;
}
```

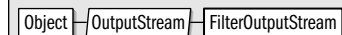
Subclasses: `BufferedInputStream`, `java.io.DataInputStream`, `LineNumberInputStream`, `PushbackInputStream`, `java.security.DigestInputStream`, `java.util.zip.CheckedInputStream`, `java.util.zip.InflaterInputStream`, `javax.crypto.CipherInputStream`, `javax.swing.ProgressMonitorInputStream`

FilterOutputStream

Java 1.0

java.io

This class provides method definitions required to filter the data to be written to the `OutputStream` specified when the `FilterOutputStream` is created. It must be subclassed to perform some sort of filtering operation and may not be instantiated directly. See the subclasses `BufferedOutputStream` and `DataOutputStream`.



```
public class FilterOutputStream extends java.io.OutputStream {
    // Public Constructors
    public FilterOutputStream(java.io.OutputStream out);
    // Public Methods Overriding OutputStream
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(int b) throws IOException;
    public void write(byte[] b) throws IOException;
    public void write(byte[] b, int off, int len) throws IOException;
    // Protected Instance Fields
}
```

FilterOutputStream

```
protected java.io.OutputStream out;  
}
```

Subclasses: `BufferedOutputStream`, `java.io.DataOutputStream`, `PrintStream`,
`java.security.DigestOutputStream`, `java.util.zip.CheckedOutputStream`, `java.util.zip.DeflaterOutputStream`,
`javax.crypto.CipherOutputStream`

FilterReader

Java 1.1

java.io

This abstract class is intended to act as a superclass for character input streams that read data from some other character input stream, filter it in some way, and then return the filtered data when a `read()` method is called. `FilterReader` is declared **abstract** so that it cannot be instantiated. But none of its methods are themselves abstract: they all simply call the requested operation on the input stream passed to the `FilterReader()` constructor. If you were allowed to instantiate a `FilterReader`, you'd find that it is a null filter (i.e., it simply reads characters from the specified input stream and returns them without any kind of filtering).

Because `FilterReader` implements a null filter, it is an ideal superclass for classes that want to implement simple filters but do not want to override all the methods of `Reader`. In order to create your own filtered character input stream, you should subclass `FilterReader` and override both its `read()` methods to perform the desired filtering operation. Note that you can implement one of the `read()` methods in terms of the other, and thus only implement the filtration once. Recall that the other `read()` methods defined by `Reader` are implemented in terms of these methods, so you do not need to override those. In some cases, you may need to override other methods of `FilterReader` and provide methods or constructors that are specific to your subclass. `FilterReader` is the character stream analog to `FilterInputStream`.

Object — Reader — FilterReader

```
public abstract class FilterReader extends Reader {  
    // Protected Constructors  
    protected FilterReader(Reader in);  
    // Public Methods Overriding Reader  
    public void close() throws IOException;  
    public void mark(int readAheadLimit) throws IOException;  
    public boolean markSupported();  
    public int read() throws IOException;  
    public int read(char[] cbuf, int off, int len) throws IOException;  
    public boolean ready() throws IOException;  
    public void reset() throws IOException;  
    public long skip(long n) throws IOException;  
    // Protected Instance Fields  
    protected Reader in;  
}
```

Subclasses: `PushbackReader`

FilterWriter

Java 1.1

java.io

This abstract class is intended to act as a superclass for character output streams that filter the data written to them before writing it to some other character output stream. `FilterWriter` is declared **abstract** so that it cannot be instantiated. But none of its methods are themselves abstract: they all simply invoke the corresponding method on the output

stream that was passed to the `FilterWriter` constructor. If you were allowed to instantiate a `FilterWriter` object, you'd find that it acts as a null filter (i.e., it simply passes the characters written to it along, without any filtration).

Because `FilterWriter` implements a null filter, it is an ideal superclass for classes that want to implement simple filters without having to override all of the methods of `Writer`. In order to create your own filtered character output stream, you should subclass `FilterWriter` and override all its `write()` methods to perform the desired filtering operation. Note that you can implement two of the `write()` methods in terms of the third and thus implement your filtering algorithm only once. In some cases, you may want to override other `Writer` methods and add other methods or constructors that are specific to your subclass. `FilterWriter` is the character-stream analog of `FilterOutputStream`.

```

Object — Writer — FilterWriter
public abstract class FilterWriter extends Writer {
// Protected Constructors
    protected FilterWriter(Writer out);
// Public Methods Overriding Writer
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(int c) throws IOException;
    public void write(char[] cbuf, int off, int len) throws IOException;
    public void write(String str, int off, int len) throws IOException;
// Protected Instance Fields
    protected Writer out;
}

```

InputStream

Java 1.0

java.io

This abstract class is the superclass of all input streams. It defines the basic input methods all input stream classes provide. `read()` reads a single byte or an array (or subarray) of bytes. It returns the byte read, the number of bytes read, or -1 if the end-of-file has been reached. `skip()` skips a specified number of bytes of input. `available()` returns the number of bytes that can be read without blocking. `close()` closes the input stream and frees up any system resources associated with it. The stream should not be used after `close()` has been called.

If `markSupported()` returns `true` for a given `InputStream`, that stream supports `mark()` and `reset()` methods. `mark()` marks the current position in the input stream so that `reset()` can return to that position (as long as no more than the specified number of bytes have been read between the calls to `mark()` and `reset()`). See also `Reader`.

```

public abstract class InputStream {
// Public Constructors
    public InputStream();
// Public Instance Methods
    public int available() throws IOException;           constant
    public void close() throws IOException;             empty
    public void mark(int readlimit);                   synchronized empty
    public boolean markSupported();                   constant
    public abstract int read() throws IOException;
}

```

InputStream

```
public int read(byte[] b) throws IOException;
public int read(byte[] b, int off, int len) throws IOException;
public void reset() throws IOException; synchronized
public long skip(long n) throws IOException;
}
```

Subclasses: ByteArrayInputStream, FileInputStream, FilterInputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream, javax.sound.sampled.AudioInputStream, org.omg.CORBA.portable.InputStream

Passed To: Too many methods to list.

Returned By: Too many methods to list.

Type Of: FilterInputStream.in, System.in

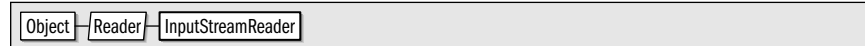
InputStreamReader

Java 1.1

java.io

This class is a character input stream that uses a byte input stream as its data source. It reads bytes from a specified `InputStream` and translates them into Unicode characters according to a particular platform- and locale-dependent character encoding. This is an important internationalization feature in Java 1.1 and later. `InputStreamReader` supports the standard `Reader` methods. It also has a `getEncoding()` method that returns the name of the encoding being used to convert bytes to characters.

When you create an `InputStreamReader`, you specify an `InputStream` from which the `InputStreamReader` is to read bytes and, optionally, the name of the character encoding used by those bytes. If you do not specify an encoding name, the `InputStreamReader` uses the default encoding for the default locale, which is usually the correct thing to do. In Java 1.4 and later, this class uses the charset conversion facilities of the `java.nio.charset` package, and allows you to explicitly specify the `Charset` or `CharsetDecoder` to be used. Prior to Java 1.4, the class allows you to specify on the name of the desired charset encoding.



```
public class InputStreamReader extends Reader {
// Public Constructors
    public InputStreamReader(java.io.InputStream in);
    public InputStreamReader(java.io.InputStream in, String charsetName) throws UnsupportedOperationException;
1.4 public InputStreamReader(java.io.InputStream in, java.nio.charset.Charset cs);
1.4 public InputStreamReader(java.io.InputStream in, java.nio.charset.CharsetDecoder dec);
// Public Instance Methods
    public String getEncoding();
// Public Methods Overriding Reader
    public void close() throws IOException;
    public int read() throws IOException;
    public int read(char[] cbuf, int offset, int length) throws IOException;
    public boolean ready() throws IOException;
}
```

Subclasses: FileReader

InterruptedException**Java 1.0**

java.io

serializable checked

An InterruptedException is an IOException that signals that an input or output operation was interrupted. The `bytesTransferred` field contains the number of bytes read or written before the operation was interrupted.



```

public class InterruptedException extends IOException {
// Public Constructors
    public InterruptedException();
    public InterruptedException(String s);
// Public Instance Fields
    public int bytesTransferred;
}
  
```

Subclasses: java.net.SocketTimeoutException

InvalidClassException**Java 1.1**

java.io

serializable checked

An InvalidClassException signals that the serialization mechanism has encountered one of several possible problems with the class of an object that is being serialized or deserialized. The `classname` field should contain the name of the class in question, and the `getMessage()` method is overridden to return this class name with the message.



```

public class InvalidClassException extends ObjectOutputStreamException {
// Public Constructors
    public InvalidClassException(String reason);
    public InvalidClassException(String cname, String reason);
// Public Methods Overriding Throwable
    public String getMessage();
// Public Instance Fields
    public String classname;
}
  
```

InvalidObjectException**Java 1.1**

java.io

serializable checked

This exception should be thrown by the `validateObject()` method of an object that implements the `ObjectInputValidation` interface when a deserialized object fails an input validation test for any reason.



```

public class InvalidObjectException extends ObjectOutputStreamException {
// Public Constructors
    public InvalidObjectException(String reason);
}
  
```

InvalidObjectException

Thrown By: java.awt.font.TextAttribute.readResolve(), ObjectInputStream.registerValidation(), ObjectInputValidation.validateObject(), java.text.AttributedCharacterIterator.Attribute.readResolve(), java.text.DateFormat.Field.readResolve(), java.text.MessageFormat.Field.readResolve(), java.text.NumberFormat.Field.readResolve()

IOException

Java 1.0

java.io

serializable checked

An IOException signals that an exceptional condition has occurred during input or output. This class has several more specific subclasses. See EOFException, FileNotFoundException, InterruptedIOException, and UTFDataFormatException.



```
public class IOException extends Exception {  
    // Public Constructors  
    public IOException();  
    public IOException(String s);  
}
```

Subclasses: Too many classes to list.

Passed To: java.awt.print.PrinterIOException.PrinterIOException()

Returned By: java.awt.print.PrinterIOException.getIOException()

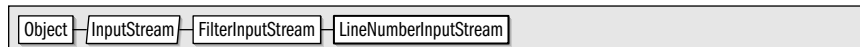
Thrown By: Too many methods to list.

LineNumberInputStream

Java 1.0; Deprecated in Java 1.1

java.io

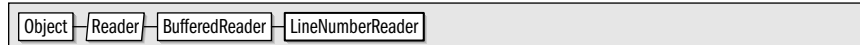
This class is a FilterInputStream that keeps track of the number of lines of data that have been read. getLineNumber() returns the current line number; setLineNumber() sets the line number of the current line. Subsequent lines are numbered starting from that number. This class is deprecated as of Java 1.1 because it does not properly convert bytes to characters. Use LineNumberReader instead.



```
public class LineNumberInputStream extends FilterInputStream {  
    // Public Constructors  
    public LineNumberInputStream(java.io.InputStream in);  
    // Public Instance Methods  
    public int getLineNumber();  
    public void setLineNumber(int lineNumber);  
    // Public Methods Overriding FilterInputStream  
    public int available() throws IOException;  
    public void mark(int readlimit);  
    public int read() throws IOException;  
    public int read(byte[] b, int off, int len) throws IOException;  
    public void reset() throws IOException;  
    public long skip(long n) throws IOException;  
}
```


LineNumberReader**Java 1.1****java.io**

This class is a character input stream that keeps track of the number of lines of text that have been read from it. It supports the usual `Reader` methods and also the `readLine()` method introduced by its superclass. In addition to these methods, you can call `getLineNumber()` to query the number of lines set so far. You can also call `setLineNumber()` to set the line number for the current line. Subsequent lines are numbered sequentially from this specified starting point. This class is a character-stream analog to `LineNumberInputStream`, which has been deprecated as of Java 1.1.



```
public class LineNumberReader extends BufferedReader {
```

```
// Public Constructors
```

```
    public LineNumberReader(Reader in);
```

```
    public LineNumberReader(Reader in, int sz);
```

```
// Public Instance Methods
```

```
    public int getLineNumber();
```

```
    public void setLineNumber(int lineNumber);
```

```
// Public Methods Overriding BufferedReader
```

```
    public void mark(int readAheadLimit) throws IOException;
```

```
    public int read() throws IOException;
```

```
    public int read(char[] cbuf, int off, int len) throws IOException;
```

```
    public String readLine() throws IOException;
```

```
    public void reset() throws IOException;
```

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

```
}
```

NotActiveException**Java 1.1****java.io***serializable checked*

This exception is thrown in several circumstances. It indicates that the invoked method was not invoked at the right time or in the correct context. Typically, it means that an `ObjectOutputStream` or `ObjectInputStream` is not currently active and therefore the requested operation cannot be performed.



```
public class NotActiveException extends ObjectStreamException {
```

```
// Public Constructors
```

```
    public NotActiveException();
```

```
    public NotActiveException(String reason);
```

```
}
```

Thrown By: `ObjectInputStream.registerValidation()`

NotSerializableException**Java 1.1****java.io***serializable checked*

This exception signals that an object cannot be serialized. It is thrown when serialization is attempted on an instance of a class that does not implement the `Serializable` interface. Note that it is also thrown when an attempt is made to serialize a `Serializable` object that refers to (or contains) an object that is not `Serializable`. A subclass of a class that is

NotSerializableException

Serializable can prevent itself from being serialized by throwing this exception from its writeObject() and/or readObject() methods.



```
public class NotSerializableException extends ObjectStreamException {  
    // Public Constructors  
    public NotSerializableException();  
    public NotSerializableException(String classname);  
}
```

ObjectInput

Java 1.1

java.io

This interface extends the **DataInput** interface and adds methods for deserializing objects and reading bytes and arrays of bytes.



```
public interface ObjectInput extends DataInput {  
    // Public Instance Methods  
    public abstract int available() throws IOException;  
    public abstract void close() throws IOException;  
    public abstract int read() throws IOException;  
    public abstract int read(byte[] b) throws IOException;  
    public abstract int read(byte[] b, int off, int len) throws IOException;  
    public abstract Object readObject() throws ClassNotFoundException, IOException;  
    public abstract long skip(long n) throws IOException;  
}
```

Implementations: **ObjectInputStream**

Passed To: java.awt.datatransfer.DataFlavor.readExternal(), Externalizable.readExternal(),
java.rmi.server.ObjID.read()

Returned By: java.rmi.server.RemoteCall.getInputStream()

ObjectInputStream

Java 1.1

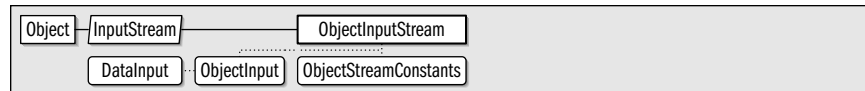
java.io

ObjectInputStream deserializes objects, arrays, and other values from a stream that was previously created with an **ObjectOutputStream**. The **readObject()** method deserializes objects and arrays (which should then be cast to the appropriate type); various other methods read primitive data values from the stream. Note that only objects that implement the **Serializable** or **Externalizable** interface can be serialized and deserialized.

A class may implement its own private **readObject(ObjectInputStream)** method to customize the way it is deserialized. If you define such a method, there are several **ObjectInputStream** methods you can use to help you deserialize the object. **defaultReadObject()** is the easiest. It reads the content of the object just as an **ObjectInputStream** would normally do. If you wrote additional data before or after the default object contents, you should read that data before or after calling **defaultReadObject()**. When working with multiple versions or implementations of a class, you may have to deserialize a set of fields that do not match the fields of your class. In this case, give your class a static field named **serialPersistentFields** whose value is an array of **ObjectStreamField** objects that describe the fields to be deserialized. If you do this, your **readObject()** method can call **readFields()** to read the specified fields from the stream and return them in a **ObjectInputStream.GetField** object.

See `ObjectStreamField` and `ObjectInputStream.GetField` for more details. Finally, you can call `registerValidation()` from a custom `readObject()` method. This method registers an `ObjectInputValidation` object (typically the object being deserialized) to be notified when a complete tree of objects has been deserialized, and the original call to the `readObject()` method of the `ObjectInputStream` is about to return to its caller.

The remaining methods include miscellaneous stream-manipulation methods and several protected methods for use by subclasses that want to customize the deserialization behavior of `ObjectInputStream`.



```

public class ObjectInputStream extends java.io.InputStream implements ObjectInput, ObjectStreamConstants {
    // Public Constructors
    public ObjectInputStream(java.io.InputStream in) throws IOException;
    // Protected Constructors
    1.2 protected ObjectInputStream() throws IOException, SecurityException;
    // Inner Classes
    1.2 public abstract static class GetField;
    // Public Instance Methods
    public void defaultReadObject() throws IOException, ClassNotFoundException;
    1.2 public ObjectInputStream.GetField readFields() throws IOException, ClassNotFoundException;
    1.4 public Object readUnshared() throws IOException, ClassNotFoundException;
    public void registerValidation(ObjectInputValidation obj, int prio) throws NotActiveException,
        InvalidObjectException;
    // Methods Implementing DataInput
    public boolean readBoolean() throws IOException;
    public byte readByte() throws IOException;
    public char readChar() throws IOException;
    public double readDouble() throws IOException;
    public float readFloat() throws IOException;
    public void readFully(byte[] buf) throws IOException;
    public void readFully(byte[] buf, int off, int len) throws IOException;
    public int readInt() throws IOException;
    public long readLong() throws IOException;
    public short readShort() throws IOException;
    public int readUnsignedByte() throws IOException;
    public int readUnsignedShort() throws IOException;
    public String readUTF() throws IOException;
    public int skipBytes(int len) throws IOException;
    // Methods Implementing ObjectInput
    public int available() throws IOException;
    public void close() throws IOException;
    public int read() throws IOException;
    public int read(byte[] buf, int off, int len) throws IOException;
    public final Object readObject() throws IOException, ClassNotFoundException;
    // Protected Instance Methods
    protected boolean enableResolveObject(boolean enable) throws SecurityException;
    1.3 protected ObjectStreamClass readClassDescriptor() throws IOException, ClassNotFoundException;
    1.2 protected Object readObjectOverride() throws IOException, ClassNotFoundException;
    protected void readStreamHeader() throws IOException, StreamCorruptedException;
    protected Class resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException;
    protected Object resolveObject(Object obj) throws IOException;
    1.3 protected Class resolveProxyClass(String[] interfaces) throws IOException, ClassNotFoundException;
    // Deprecated Public Methods
  
```

ObjectInputStream

```
# public String readLine() throws IOException; Implements:DataInput
}
```

Passed To: java.beans.beancontext.BeanContextServicesSupport.bcsPreDeserializationHook(),
java.beans.beancontext.BeanContextSupport.{bcsPreDeserializationHook(), deserialize(),
readChildren()}, javax.rmi.CORBA.StubDelegate.readObject(),
javax.swing.text.StyleContext.{readAttributes(), readAttributeSet()}

ObjectInputStream.GetField

Java 1.2

java.io

This class holds the values of named fields read by an `ObjectInputStream`. It gives the programmer precise control over the deserialization process and is typically used when implementing an object with a set of fields that do not match the set of fields (and the serialization stream format) of the original implementation of the object. This class allows the implementation of a class to change without breaking serialization compatibility.

In order to use the `GetField` class, your class must implement a private `readObject()` method that is responsible for custom deserialization. Typically, when using the `GetField` class, you have also specified an array of `ObjectStreamField` objects as the value of a private static field named `serialPersistentFields`. This array specifies the names and types of all fields expected to be found when reading from a serialization stream. If there is no `serialPersistentField` field, the array of `ObjectStreamField` objects is created from the actual fields (excluding `static` and `transient` fields) of the class.

Within the `readObject()` method of your class, call the `readFields()` method of `ObjectInputStream()`. This method reads the values of all fields from the stream and stores them in an `ObjectInputStream.GetField` object that it returns. This `GetField` object is essentially a mapping from field names to field values, and you can extract the values of whatever fields you need in order to restore the proper state of the object being deserialized. The various `get()` methods return the values of named fields of specified types. Each method takes a default value as an argument, in case no value for the named field was present in the serialization stream. (This can happen when deserializing an object written by an earlier version of the class, for example.) Use the `defaulted()` method to determine whether the `GetField` object contains a value for the named field. If this method returns `true`, the named field had no value in the stream, so the `get()` method of the `GetField` object has to return the specified default value. The `getObjectStreamClass()` method of a `GetField` object returns the `ObjectStreamClass` object for the object being deserialized. This `ObjectStreamClass` can obtain the array of `ObjectStreamField` objects for the class. See also `ObjectOutputStream.PutField`.

```
public abstract static class ObjectInputStream.GetField {
    // Public Constructors
    public GetField();
    // Public Instance Methods
    public abstract boolean defaulted(String name) throws IOException;
    public abstract boolean get(String name, boolean val) throws IOException;
    public abstract byte get(String name, byte val) throws IOException;
    public abstract char get(String name, char val) throws IOException;
    public abstract short get(String name, short val) throws IOException;
    public abstract int get(String name, int val) throws IOException;
    public abstract long get(String name, long val) throws IOException;
    public abstract float get(String name, float val) throws IOException;
    public abstract double get(String name, double val) throws IOException;
    public abstract Object get(String name, Object val) throws IOException;
```

```
public abstract ObjectOutputStream getObjectStreamClass();
}
```

Returned By: ObjectOutputStream.readFields()

ObjectInputValidation

Java 1.1

java.io

A class implements this interface and defines the `validateObject()` method in order to validate itself when it and all the objects it depends on have been completely deserialized from an `ObjectInputStream`. The `validateObject()` method is only invoked, however, if the object is passed to `ObjectInputStream.registerValidation()`; this must be done from the `readObject()` method of the object. Note that if an object is deserialized as part of a larger object graph, its `validateObject()` method is not invoked until the entire graph is read, and the original call to `ObjectInputStream.readObject()` is about to return. `validateObject()` should throw an `InvalidObjectException` if the object fails validation. This stops object serialization, and the original call to `ObjectInputStream.readObject()` terminates with the `InvalidObjectException`.

```
public interface ObjectInputValidation {
    // Public Instance Methods
    public abstract void validateObject() throws InvalidObjectException;
}
```

Passed To: ObjectOutputStream.registerValidation()

ObjectOutput

Java 1.1

java.io

This interface extends the `DataOutput` interface and adds methods for serializing objects and writing bytes and arrays of bytes.



```
graph LR
    DataOutput --> ObjectOutput
```

```
public interface ObjectOutput extends DataOutput {
    // Public Instance Methods
    public abstract void close() throws IOException;
    public abstract void flush() throws IOException;
    public abstract void write(byte[] b) throws IOException;
    public abstract void write(int b) throws IOException;
    public abstract void write(byte[] b, int off, int len) throws IOException;
    public abstract void writeObject(Object obj) throws IOException;
}
```

Implementations: ObjectOutputStream

Passed To: java.awt.datatransfer.DataFlavor.writeExternal(), Externalizable.writeExternal(), ObjectOutputStream.PutField.write(), java.rmi.server.ObjID.write(), java.rmi.server.RemoteRef.getRefClass()

Returned By: java.rmi.server.RemoteCall.{getOutputStream(), getResultStream()}

ObjectOutputStream

Java 1.1

java.io

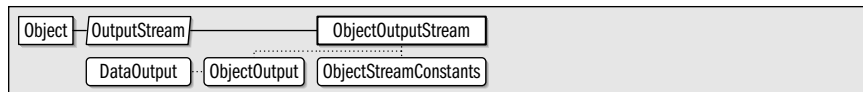
The `ObjectOutputStream` serializes objects, arrays, and other values to a stream. The `writeObject()` method serializes an object or array, and various other methods write primi-

ObjectOutputStream

tive data values to the stream. Note that only objects that implement the `Serializable` or `Externalizable` interface can be serialized.

A class that wants to customize the way instances are serialized should declare a private `writeObject(ObjectOutputStream)` method. This method is invoked when an object is being serialized and can use several additional methods of `ObjectOutputStream`. `defaultWriteObject()` performs the same serialization that would happen if no `writeObject()` method existed. An object can call this method to serialize itself and then use other methods of `ObjectOutputStream` to write additional data to the serialization stream. The class must define a matching `readObject()` method to read that additional data, of course. When working with multiple versions or implementations of a class, you may have to serialize a set of fields that do not precisely match the fields of your class. In this case, give your class a static field named `serialPersistentFields` whose value is an array of `ObjectStreamField` objects that describe the fields to be serialized. In your `writeObject()` method, call `putFields()` to obtain an `ObjectOutputStream.PutField` object. Store field names and values into this object, and then call `writeFields()` to write them out to the serialization stream. See `ObjectStreamField` and `ObjectOutputStream.PutField` for further details.

The remaining methods of `ObjectOutputStream` are miscellaneous stream-manipulation methods and protected methods for use by subclasses that want to customize its serialization behavior.



```
public class ObjectOutputStream extends java.io.OutputStream implements ObjectOutput, ObjectStreamConstants {
    // Public Constructors
    public ObjectOutputStream(java.io.OutputStream out) throws IOException;
    // Protected Constructors
    1.2 protected ObjectOutputStream() throws IOException, SecurityException;
    // Inner Classes
    1.2 public abstract static class PutField;
    // Public Instance Methods
    public void defaultWriteObject() throws IOException;
    1.2 public ObjectOutputStream.PutField putFields() throws IOException;
    public void reset() throws IOException;
    1.2 public void useProtocolVersion(int version) throws IOException;
    1.2 public void writeFields() throws IOException;
    1.4 public void writeUnshared(Object obj) throws IOException;
    // Methods Implementing DataOutput
    public void writeBoolean(boolean val) throws IOException;
    public void writeByte(int val) throws IOException;
    public void writeBytes(String str) throws IOException;
    public void writeChar(int val) throws IOException;
    public void writeChars(String str) throws IOException;
    public void writeDouble(double val) throws IOException;
    public void writeFloat(float val) throws IOException;
    public void writeInt(int val) throws IOException;
    public void writeLong(long val) throws IOException;
    public void writeShort(int val) throws IOException;
    public void writeUTF(String str) throws IOException;
    // Methods Implementing ObjectOutput
    public void close() throws IOException;
```

```

public void flush() throws IOException;
public void write(int val) throws IOException;
public void write(byte[] buf) throws IOException;
public void write(byte[] buf, int off, int len) throws IOException;
public final void writeObject(Object obj) throws IOException;
// Protected Instance Methods
protected void annotateClass(Class c) throws IOException; empty
1.3 protected void annotateProxyClass(Class c) throws IOException; empty
protected void drain() throws IOException;
protected boolean enableReplaceObject(boolean enable) throws SecurityException;
protected Object replaceObject(Object obj) throws IOException;
1.3 protected void writeClassDescriptor(ObjectStreamClass desc) throws IOException;
1.2 protected void writeObjectOverride(Object obj) throws IOException; empty
protected void writeStreamHeader() throws IOException;
}

```

Passed To: java.awt.AWTEventMulticaster.{save(), saveInternal()},
 java.beans.beancontext.BeanContextServicesSupport.bcsPreSerializationHook(),
 java.beans.beancontext.BeanContextSupport.{bcsPreSerializationHook(), serialize(), writeChildren()},
 javax.rmi.CORBA.StubDelegate.writeObject(), javax.swing.text.StyleContext.{writeAttributes(),
 writeAttributeSet()}

ObjectOutputStream.PutField

Java 1.2

java.io

This class holds values of named fields and allows them to be written to an `ObjectOutputStream` during the process of object serialization. It gives the programmer precise control over the serialization process and is typically used when the set of fields defined by a class do not match the set of fields (and the serialization stream format) defined by the original implementation of the class. In other words, `ObjectOutputStream.PutField` allows the implementation of a class to change without breaking serialization compatibility.

To use the `PutField` class, you typically define a private static `serialPersistentFields` field that refers to an array of `ObjectStreamField` objects. This array defines the set of fields written to the `ObjectOutputStream`, and therefore defines the serialization format. If you do not declare a `serialPersistentFields` field, the set of fields is all fields of the class, excluding static and transient fields.

In addition to the `serialPersistentFields` field, your class must also define a private `writeObject()` method that is responsible for the custom serialization of your class. In this method, call the `putFields()` method of `ObjectOutputStream` to obtain an `ObjectOutputStream.PutField` object. Once you have this object, use its various `put()` methods to specify the names and values of the field to be written out. The set of named fields should match those specified by `serialPersistentFields`. You may specify the fields in any order; the `PutField` class is responsible for writing them out in the correct order. Once you have specified the values of all fields, call the `write()` method of your `PutField` object in order to write the field values out to the serialization stream.

To reverse this custom serialization process, see `ObjectInputStream.GetField`.

```

public abstract static class ObjectOutputStream.PutField {
// Public Constructors
public PutField();

```

ObjectOutputStream.PutField

```
// Public Instance Methods
public abstract void put(String name, long val);
public abstract void put(String name, int val);
public abstract void put(String name, float val);
public abstract void put(String name, Object val);
public abstract void put(String name, double val);
public abstract void put(String name, byte val);
public abstract void put(String name, boolean val);
public abstract void put(String name, short val);
public abstract void put(String name, char val);
// Deprecated Public Methods
# public abstract void write(ObjectOutputStream out) throws IOException;
}
```

Returned By: ObjectOutputStream.putFields()

ObjectStreamClass

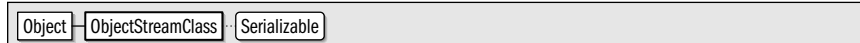
Java 1.1

java.io

serializable

This class represents a class that is being serialized. An `ObjectStreamClass` object contains the name of a class, its unique version identifier, and the name and type of the fields that constitute the serialization format for the class. `getSerialVersionUID()` returns a unique version identifier for the class. It returns either the value of the private `serialVersionUID` field of the class or a computed value that is based upon the public API of the class. In Java 1.2 and later, `getFields()` returns an array of `ObjectStreamField` objects that represent the names and types of the fields of the class to be serialized. `getField()` returns a single `ObjectStreamField` object that represents a single named field. By default, these methods use all the fields of a class except those that are `static` or `transient`. However, this default set of fields can be overridden by declaring a private `serialPersistentFields` field in the class. The value of this field should be the desired array of `ObjectStreamField` objects.

`ObjectStreamClass` class does not have a constructor; you should use the static `lookup()` method to obtain an `ObjectStreamClass` object for a given `Class` object. The `forClass()` instance method performs the opposite operation; it returns the `Class` object that corresponds to a given `ObjectStreamClass`. Most applications never need to use this class.



```
public class ObjectStreamClass implements Serializable {
// No Constructor
// Public Constants
1.2 public static final ObjectStreamField[ ] NO_FIELDS;
// Public Class Methods
public static ObjectStreamClass lookup(Class c);
// Public Instance Methods
public Class forClass();
1.2 public ObjectStreamField getField(String name);
1.2 public ObjectStreamField[ ] getFields();
public String getName();
public long getSerialVersionUID();
// Public Methods Overriding Object
public String toString();
}
```


Passed To: ObjectInputStream.resolveClass(), ObjectOutputStream.writeClassDescriptor()

Returned By: ObjectInputStream.readClassDescriptor(),
ObjectInputStream.GetField.getObjectStreamClass(), ObjectStreamClass.lookup()

ObjectStreamConstants

Java 1.2

java.io

This interface defines various constants used by the Java object-serialization mechanism. Two important constants are `PROTOCOL_VERSION_1` and `PROTOCOL_VERSION_2`, which specify the version of the serialization protocol to use. In Java 1.2, you can pass either of these values to the `useProtocolVersion()` method of an `ObjectOutputStream`. By default, Java 1.2 uses Version 2 of the protocol, and Java 1.1 uses Version 1 when serializing objects. Java 1.2 can deserialize objects written using either version of the protocol, as can Java 1.1.7 and later. If you want to serialize an object so that it can be read by versions of Java prior to Java 1.1.7, use `PROTOCOL_VERSION_1`.

The other constants defined by this interface are low-level values used by the serialization protocol. You do not need to use them unless you are reimplementing the serialization mechanism yourself.

```
public interface ObjectStreamConstants {
    // Public Constants
    public static final int baseWireHandle;                =8257536
    public static final int PROTOCOL_VERSION_1;           =1
    public static final int PROTOCOL_VERSION_2;           =2
    public static final byte SC_BLOCK_DATA;               =8
    public static final byte SC_EXTERNALIZABLE;           =4
    public static final byte SC_SERIALIZABLE;             =2
    public static final byte SC_WRITE_METHOD;             =1
    public static final short STREAM_MAGIC;               =-21267
    public static final short STREAM_VERSION;             =5
    public static final SerializablePermission SUBCLASS_IMPLEMENTATION_PERMISSION;
    public static final SerializablePermission SUBSTITUTION_PERMISSION;
    public static final byte TC_ARRAY;                   =117
    public static final byte TC_BASE;                   =112
    public static final byte TC_BLOCKDATA;              =119
    public static final byte TC_BLOCKDATALONG;          =122
    public static final byte TC_CLASS;                  =118
    public static final byte TC_CLASSDESC;              =114
    public static final byte TC_ENDBLOCKDATA;           =120
    public static final byte TC_EXCEPTION;              =123
    1.3 public static final byte TC_LONGSTRING;          =124
    public static final byte TC_MAX;                    =125
    public static final byte TC_NULL;                   =112
    public static final byte TC_OBJECT;                 =115
    1.3 public static final byte TC_PROXYCLASSDESC;      =125
    public static final byte TC_REFERENCE;              =113
    public static final byte TC_RESET;                  =121
    public static final byte TC_STRING;                 =116
}
```

Implementations: ObjectInputStream, ObjectOutputStream

ObjectStreamException

Java 1.1

java.io

serializable checked

This class is the superclass of a number of more specific exception types that may be raised in the process of serializing and deserializing objects with the `ObjectOutputStream` and `ObjectInputStream` classes.



```

public abstract class ObjectStreamException extends IOException {
    // Protected Constructors
    protected ObjectStreamException();
    protected ObjectStreamException(String classname);
}
  
```

Subclasses: `InvalidClassException`, `InvalidObjectException`, `NotActiveException`, `NotSerializableException`, `OptionalDataException`, `StreamCorruptedException`, `WriteAbortedException`

Thrown By: `java.awt.AWTKeyStroke.readResolve()`, `java.awt.color.ICC_Profile.readResolve()`, `java.security.cert.Certificate.writeReplace()`, `java.security.cert.Certificate.CertificateRep.readResolve()`, `java.security.cert.CertPath.writeReplace()`, `java.security.cert.CertPath.CertPathRep.readResolve()`, `javax.print.attribute.EnumSyntax.readResolve()`

ObjectStreamField

Java 1.2

java.io

comparable

This class represents a named field of a specified type (i.e., a specified `Class`). When a class serializes itself by writing a set of fields that are different from the fields it uses in its own implementation, it defines the set of fields to be written with an array of `ObjectStreamField` objects. This array should be the value of a private static field named `serialPersistentFields`. The methods of this class are used internally by the serialization mechanism and are not typically used elsewhere. See also `ObjectOutputStream.PutField` and `ObjectInputStream.GetField`.



```

public class ObjectStreamField implements Comparable {
    // Public Constructors
    public ObjectStreamField(String name, Class type);
    1.4 public ObjectStreamField(String name, Class type, boolean unshared);
    // Property Accessor Methods (by property name)
    public String getName();
    public int getOffset();
    public boolean isPrimitive();
    public Class getType();
    public char getTypeCode();
    public String getTypeString();
    1.4 public boolean isUnshared();
    // Methods Implementing Comparable
    public int compareTo(Object obj);
    // Public Methods Overriding Object
    public String toString();
    // Protected Instance Methods
    protected void setOffset(int offset);
}
  
```

Returned By: ObjectOutputStreamClass.{getField(), getFields()}

Type Of: ObjectOutputStreamClass.NO_FIELDS

OptionalDataException

Java 1.1

java.io

serializable checked

Thrown by the readObject() method of an ObjectInputStream when it encounters primitive type data where it expects object data. Despite the exception name, this data is not optional, and object deserialization is stopped.



```

public class OptionalDataException extends ObjectOutputStreamException {
// No Constructor
// Public Instance Fields
    public boolean eof;
    public int length;
}
  
```

OutputStream

Java 1.0

java.io

This abstract class is the superclass of all output streams. It defines the basic output methods all output stream classes provide. write() writes a single byte or an array (or subarray) of bytes. flush() forces any buffered output to be written. close() closes the stream and frees up any system resources associated with it. The stream may not be used once close() has been called. See also Writer.

```

public abstract class OutputStream {
// Public Constructors
    public OutputStream();
// Public Instance Methods
    public void close() throws IOException;
    public void flush() throws IOException;
    public abstract void write(int b) throws IOException;
    public void write(byte[] b) throws IOException;
    public void write(byte[] b, int off, int len) throws IOException;
}
  
```

Subclasses: ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream, PipedOutputStream, org.omg.CORBA.portable.OutputStream

Passed To: Too many methods to list.

Returned By: Process.getOutputStream(), Runtime.getLocalizedOutputStream(), java.net.Socket.getOutputStream(), java.net.SocketImpl.getOutputStream(), java.net.URLConnection.getOutputStream(), java.nio.channels.Channels.newOutputStream(), java.rmi.server.LogStream.getOutputStream(), java.sql.Blob.setBinaryStream(), java.sql.Clob.setAsciiStream(), javax.print.StreamPrintService.getOutputStream(), javax.xml.transform.stream.StreamResult.getOutputStream()

Type Of: FilterOutputStream.out

OutputStreamWriter

Java 1.1

java.io

This class is a character output stream that uses a byte output stream as the destination for its data. When characters are written to an `OutputStreamWriter`, it translates them into bytes according to a particular locale- and/or platform-specific character encoding and writes those bytes to the specified `OutputStream`. This is an important internationalization feature in Java 1.1 and later. `OutputStreamWriter` supports the usual `Writer` methods. It also has a `getEncoding()` method that returns the name of the encoding being used to convert characters to bytes.

When you create an `OutputStreamWriter`, specify the `OutputStream` to which it writes bytes and, optionally, the name of the character encoding that should be used to convert characters to bytes. If you do not specify an encoding name, the `OutputStreamWriter` uses the default encoding of the default locale, which is usually the correct thing to do. In Java 1.4 and later, this class uses the charset conversion facilities of the `java.nio.charset` package, and allows you to explicitly specify the `Charset` or `CharsetEncoder` to be used. Prior to Java 1.4, the class allows you to specify on the name of the desired charset encoding.

```

Object — Writer — OutputStreamWriter

public class OutputStreamWriter extends Writer {
// Public Constructors
    public OutputStreamWriter(java.io.OutputStream out);
    public OutputStreamWriter(java.io.OutputStream out, String charsetName)
        throws UnsupportedOperationException;
1.4 public OutputStreamWriter(java.io.OutputStream out, java.nio.charset.CharsetEncoder enc);
1.4 public OutputStreamWriter(java.io.OutputStream out, java.nio.charset.Charset cs);
// Public Instance Methods
    public String getEncoding();
// Public Methods Overriding Writer
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(int c) throws IOException;
    public void write(char[] cbuf, int off, int len) throws IOException;
    public void write(String str, int off, int len) throws IOException;
}

```

Subclasses: `FileWriter`

PipedInputStream

Java 1.0

java.io

This class is an `InputStream` that implements one half of a pipe and is useful for communication between threads. A `PipedInputStream` must be connected to a `PipedOutputStream` object, which may be specified when the `PipedInputStream` is created or with the `connect()` method. Data read from a `PipedInputStream` object is received from the `PipedOutputStream` to which it is connected. See `InputStream` for information on the low-level methods for reading data from a `PipedInputStream`. A `FilterInputStream` can provide a higher-level interface for reading data from a `PipedInputStream`.

```

Object — InputStream — PipedInputStream

public class PipedInputStream extends java.io.InputStream {
// Public Constructors
    public PipedInputStream();

```

```

    public PipedInputStream(PipedOutputStream src) throws IOException;
    // Protected Constants
    1.1 protected static final int PIPE_SIZE;                                =1024
    // Public Instance Methods
    public void connect(PipedOutputStream src) throws IOException;
    // Public Methods Overriding InputStream
    public int available() throws IOException;                                synchronized
    public void close() throws IOException;
    public int read() throws IOException;                                    synchronized
    public int read(byte[] b, int off, int len) throws IOException;          synchronized
    // Protected Instance Methods
    1.1 protected void receive(int b) throws IOException;                    synchronized
    // Protected Instance Fields
    1.1 protected byte[] buffer;
    1.1 protected int in;
    1.1 protected int out;
}

```

Passed To: PipedOutputStream.{connect(), PipedOutputStream()}

PipedOutputStream

Java 1.0

java.io

This class is an `OutputStream` that implements one half of a pipe and is useful for communication between threads. A `PipedOutputStream` must be connected to a `PipedInputStream`, which may be specified when the `PipedOutputStream` is created or with the `connect()` method. Data written to the `PipedOutputStream` is available for reading on the `PipedInputStream`. See `OutputStream` for information on the low-level methods for writing data to a `PipedOutputStream`. A `FilterOutputStream` can provide a higher-level interface for writing data to a `PipedOutputStream`.

Object	OutputStream	PipedOutputStream
--------	--------------	-------------------

```

public class PipedOutputStream extends java.io.OutputStream {
    // Public Constructors
    public PipedOutputStream();
    public PipedOutputStream(PipedInputStream snk) throws IOException;
    // Public Instance Methods
    public void connect(PipedInputStream snk) throws IOException;          synchronized
    // Public Methods Overriding OutputStream
    public void close() throws IOException;
    public void flush() throws IOException;                                synchronized
    public void write(int b) throws IOException;
    public void write(byte[] b, int off, int len) throws IOException;
}

```

Passed To: PipedInputStream.{connect(), PipedInputStream()}

PipedReader

Java 1.1

java.io

`PipedReader` is a character input stream that reads characters from a `PipedWriter` character output stream to which it is connected. `PipedReader` implements one half of a pipe and is useful for communication between two threads of an application. A `PipedReader` cannot be used until it is connected to a `PipedWriter` object, which may be passed to the `PipedReader()` constructor or to the `connect()` method. `PipedReader` inherits most of the

PipedReader

methods of its superclass. See `Reader` for more information. `PipedReader` is the character-stream analog of `PipedInputStream`.

```
Object — Reader — PipedReader

public class PipedReader extends Reader {
// Public Constructors
    public PipedReader();
    public PipedReader(PipedWriter src) throws IOException;
// Public Instance Methods
    public void connect(PipedWriter src) throws IOException;
// Public Methods Overriding Reader
    public void close() throws IOException;
1.2 public int read() throws IOException;                                synchronized
    public int read(char[] cbuf, int off, int len) throws IOException;    synchronized
1.2 public boolean ready() throws IOException;                          synchronized
}
```

Passed To: `PipedWriter.{connect(), PipedWriter()}`

PipedWriter

Java 1.1

java.io

`PipedWriter` is a character output stream that writes characters to the `PipedReader` character input stream to which it is connected. `PipedWriter` implements one half of a pipe and is useful for communication between two threads of an application. A `PipedWriter` cannot be used until it is connected to a `PipedReader` object, which may be passed to the `PipedWriter()` constructor, or to the `connect()` method. `PipedWriter` inherits most of the methods of its superclass. See `Writer` for more information. `PipedWriter` is the character stream analog of `PipedOutputStream`.

```
Object — Writer — PipedWriter

public class PipedWriter extends Writer {
// Public Constructors
    public PipedWriter();
    public PipedWriter(PipedReader snk) throws IOException;
// Public Instance Methods
    public void connect(PipedReader snk) throws IOException;    synchronized
// Public Methods Overriding Writer
    public void close() throws IOException;
    public void flush() throws IOException;                      synchronized
1.2 public void write(int c) throws IOException;
    public void write(char[] cbuf, int off, int len) throws IOException;
}
```

Passed To: `PipedReader.{connect(), PipedReader()}`

PrintStream

Java 1.0

java.io

This class is a `FilterOutputStream` that implements a number of methods for displaying textual representations of Java primitive data types. The `print()` methods output standard textual representations of each data type. The `println()` methods do the same and follow the representations with newlines. Each method converts a Java primitive type to a `String` representation and outputs the resulting string. When an `Object` is passed to a `print()` or `println()`, it is converted to a `String` by calling its `toString()` method. `PrintStream` is

the `OutputStream` type that makes it easiest to output text. As such, it is the most commonly used of the output streams. The `System.out` variable is a `PrintStream`.

Note that in Java 1.0 this class does not handle Unicode characters correctly; it discards the top 8 bits of all 16-bit characters and thus works only with Latin-1 (ISO8859-1) characters. Although this problem has been fixed as of Java 1.1, `PrintStream` has been superseded by `PrintWriter` as of Java 1.1. The constructors of this class have been deprecated, but the class itself has not, because it is still used by the `System.out` and `System.err` standard output streams.

`PrintStream`, and its `PrintWriter` replacement, output textual representations of Java data types. Use `DataOutputStream` to output binary representations of data.

Object	OutputStream	FilterOutputStream	PrintStream
--------	--------------	--------------------	-------------

```

public class PrintStream extends FilterOutputStream {
    // Public Constructors
    public PrintStream(java.io.OutputStream out);
    public PrintStream(java.io.OutputStream out, boolean autoFlush);
    1.4 public PrintStream(java.io.OutputStream out, boolean autoFlush, String encoding)
        throws UnsupportedOperationException;
    // Public Instance Methods
    public boolean checkError();
    public void print(float f);
    public void print(long l);
    public void print(int i);
    public void print(double d);
    public void print(char[] s);
    public void print(String s);
    public void print(Object obj);
    public void print(char c);
    public void print(boolean b);
    public void println();
    public void println(float x);
    public void println(long x);
    public void println(int x);
    public void println(String x);
    public void println(Object x);
    public void println(double x);
    public void println(char[] x);
    public void println(char x);
    public void println(boolean x);
    // Public Methods Overriding FilterOutputStream
    public void close();
    public void flush();
    public void write(int b);
    public void write(byte[] buf, int off, int len);
    // Protected Instance Methods
    1.1 protected void setError();
}

```

Subclasses: `java.rmi.server.LogStream`

Passed To: Too many methods to list.

Returned By: `java.rmi.server.LogStream.getDefaultStream()`, `java.rmi.server.RemoteServer.getLog()`, `java.sql.DriverManager.getLogStream()`, `javax.swing.DebugGraphics.logStream()`

Type Of: `System.{err, out}`

PrintWriter

Java 1.1

java.io

This class is a character output stream that implements a number of `print()` and `println()` methods that output textual representations of primitive values and objects. When you create a `PrintWriter` object, you specify a character or byte output stream that it should write its characters to and, optionally, whether the `PrintWriter` stream should be automatically flushed whenever `println()` is called. If you specify a byte output stream as the destination, the `PrintWriter()` constructor automatically creates the necessary `OutputStreamWriter` object to convert characters to bytes using the default encoding.

`PrintWriter` implements the normal `write()`, `flush()`, and `close()` methods all `Writer` subclasses define. It is more common to use the higher-level `print()` and `println()` methods, each of which converts its argument to a string before outputting it. `println()` can also terminate the line (and optionally flush the buffer) after printing its argument.

The methods of `PrintWriter` never throw exceptions. Instead, when errors occur, they set an internal flag you can check by calling `checkError()`. `checkError()` first flushes the internal stream and then returns `true` if any exception has occurred while writing values to that stream. Once an error has occurred on a `PrintWriter` object, all subsequent calls to `checkError()` return `true`; there is no way to reset the error flag.

`PrintWriter` is the character stream analog to `PrintStream`, which it supersedes. You can usually trivially replace any `PrintStream` objects in a program with `PrintWriter` objects. This is particularly important for internationalized programs. The only valid remaining use for the `PrintStream` class is for the `System.out` and `System.err` standard output streams. See `PrintStream` for details.



```
public class PrintWriter extends Writer {
    // Public Constructors
    public PrintWriter(java.io.OutputStream out);
    public PrintWriter(Writer out);
    public PrintWriter(java.io.OutputStream out, boolean autoFlush);
    public PrintWriter(Writer out, boolean autoFlush);
    // Public Instance Methods
    public boolean checkError();
    public void print(int i);
    public void print(long l);
    public void print(char c);
    public void print(boolean b);
    public void print(float f);
    public void print(double d);
    public void print(char[] s);
    public void print(Object obj);
    public void print(String s);
    public void println();
    public void println(int x);
    public void println(long x);
    public void println(boolean x);
    public void println(char x);
    public void println(String x);
    public void println(Object x);
    public void println(char[] x);
    public void println(float x);
    public void println(double x);
}
```



```
// Public Methods Overriding Writer
public void close();
public void flush();
public void write(char[] buf);
public void write(String s);
public void write(int c);
public void write(String s, int off, int len);
public void write(char[] buf, int off, int len);
// Protected Instance Methods
protected void setError();
// Protected Instance Fields
1.2 protected Writer out;
}
```

Passed To: java.awt.Component.list(), java.awt.Container.list(), Throwable.printStackTrace(), java.security.cert.CertPathBuilderException.printStackTrace(), java.security.cert.CertPathValidatorException.printStackTrace(), java.security.cert.CertStoreException.printStackTrace(), java.sql.DriverManager.setLogWriter(), java.util.Properties.list(), javax.naming.NamingException.printStackTrace(), javax.sql.ConnectionPoolDataSource.setLogWriter(), javax.sql.DataSource.setLogWriter(), javax.sql.XADataSource.setLogWriter(), javax.xml.transform.TransformerException.printStackTrace()

Returned By: java.sql.DriverManager.getLogWriter(), javax.sql.ConnectionPoolDataSource.getLogWriter(), javax.sql.DataSource.getLogWriter(), javax.sql.XADataSource.getLogWriter()

PushbackInputStream

Java 1.0

java.io

This class is a `FilterInputStream` that implements a one-byte pushback buffer or, as of Java 1.1, a pushback buffer of a specified length. The `unread()` methods push bytes back into the stream; these bytes are the first ones read by the next call to a `read()` method. This class is sometimes useful when writing parsers. See also `PushbackReader`.

Object — InputStream — FilterInputStream — PushbackInputStream

```
public class PushbackInputStream extends FilterInputStream {
// Public Constructors
public PushbackInputStream(java.io.InputStream in);
1.1 public PushbackInputStream(java.io.InputStream in, int size);
// Public Instance Methods
public void unread(int b) throws IOException;
1.1 public void unread(byte[] b) throws IOException;
1.1 public void unread(byte[] b, int off, int len) throws IOException;
// Public Methods Overriding FilterInputStream
public int available() throws IOException;
1.2 public void close() throws IOException; synchronized
public boolean markSupported(); constant
public int read() throws IOException;
public int read(byte[] b, int off, int len) throws IOException;
1.2 public long skip(long n) throws IOException;
// Protected Instance Fields
1.1 protected byte[] buf;
1.1 protected int pos;
}
```

PushbackReader

Java 1.1

java.io

This class is a character input stream that uses another input stream as its input source and adds the ability to push characters back onto the stream. This feature is often useful when writing parsers. When you create a `PushbackReader` stream, you specify the stream to be read from and, optionally, the size of the pushback buffer (i.e., the number of characters that may be pushed back onto the stream or unread). If you do not specify a size for this buffer, the default size is one character. `PushbackReader` inherits or overrides all standard `Reader` methods and adds three `unread()` methods that push a single character, an array of characters, or a portion of an array of characters back onto the stream. This class is the character stream analog of `PushbackInputStream`.

Object	Reader	FilterReader	PushbackReader
--------	--------	--------------	----------------

```

public class PushbackReader extends FilterReader {
    // Public Constructors
    public PushbackReader(Reader in);
    public PushbackReader(Reader in, int size);
    // Public Instance Methods
    public void unread(int c) throws IOException;
    public void unread(char[] cbuf) throws IOException;
    public void unread(char[] cbuf, int off, int len) throws IOException;
    // Public Methods Overriding FilterReader
    public void close() throws IOException;
    1.2 public void mark(int readAheadLimit) throws IOException;
    public boolean markSupported(); constant
    public int read() throws IOException;
    public int read(char[] cbuf, int off, int len) throws IOException;
    public boolean ready() throws IOException;
    1.2 public void reset() throws IOException;
}

```

RandomAccessFile

Java 1.0

java.io

This class allows you to read and write arbitrary bytes, text, and primitive Java data types from or to any specified location in a file. Because this class provides random, rather than sequential, access to files, it is neither a subclass of `InputStream` nor of `OutputStream`, but provides an entirely independent method for reading and writing data from or to files. `RandomAccessFile` implements the same interfaces as `DataInputStream` and `DataOutputStream`, and thus defines the same methods for reading and writing data as those classes do.

The `seek()` method provides random access to the file; it is used to select the position in the file where data should be read or written. The various read and write methods update this file position so that a sequence of read or write operations can be performed on a contiguous portion of the file without having to call the `seek()` method before each read or write.

The `mode` argument to the constructor methods should be "r" for a file that will be read-only or "rw" for a file that will be written (and perhaps read as well). In Java 1.4 and later, two other values for the `mode` argument are allowed as well. A mode of "rwd" opens the file for reading and writing and requires that, if the file resides on a local filesystem, every update to the file content be written synchronously to the underlying file. The "rws" mode is similar but requires synchronous updates to both the file's con-

tent and its “metadata” (which includes things such as file access times). Using “rws” mode may require that the file metadata be modified every time the file is read.

In Java 1.4 and later, use the `getChannel()` method to obtain a `FileChannel` object that you can use to access the file using the New I/O API of `java.nio` and its subpackages. If the `RandomAccessFile` was opened with a mode of “r”, then the `FileChannel` will allow only reading. Otherwise, it will allow both reading and writing.



```

classDiagram
    class RandomAccessFile {
        <<abstract>>
        +DataInput
        +DataOutput
    }
    class DataInput
    class DataOutput
    RandomAccessFile <|-- DataInput
    RandomAccessFile <|-- DataOutput
  
```

```

public class RandomAccessFile implements DataInput, DataOutput {
    // Public Constructors
    public RandomAccessFile(File file, String mode) throws FileNotFoundException;
    public RandomAccessFile(String name, String mode) throws FileNotFoundException;

    // Public Instance Methods
    public void close() throws IOException; native
    1.4 public final java.nio.channels.FileChannel getChannel();
    public final FileDescriptor getFD() throws IOException;
    public long getFilePointer() throws IOException; native
    public long length() throws IOException; native
    public int read() throws IOException; native
    public int read(byte[] b) throws IOException;
    public int read(byte[] b, int off, int len) throws IOException;
    public void seek(long pos) throws IOException; native
    1.2 public void setLength(long newLength) throws IOException; native

    // Methods Implementing DataInput
    public final boolean readBoolean() throws IOException;
    public final byte readByte() throws IOException;
    public final char readChar() throws IOException;
    public final double readDouble() throws IOException;
    public final float readFloat() throws IOException;
    public final void readFully(byte[] b) throws IOException;
    public final void readFully(byte[] b, int off, int len) throws IOException;
    public final int readInt() throws IOException;
    public final String readLine() throws IOException;
    public final long readLong() throws IOException;
    public final short readShort() throws IOException;
    public final int readUnsignedByte() throws IOException;
    public final int readUnsignedShort() throws IOException;
    public final String readUTF() throws IOException;
    public int skipBytes(int n) throws IOException;

    // Methods Implementing DataOutput
    public void write(int b) throws IOException; native
    public void write(byte[] b) throws IOException;
    public void write(byte[] b, int off, int len) throws IOException;
    public final void writeBoolean(boolean v) throws IOException;
    public final void writeByte(int v) throws IOException;
    public final void writeBytes(String s) throws IOException;
    public final void writeChar(int v) throws IOException;
    public final void writeChars(String s) throws IOException;
    public final void writeDouble(double v) throws IOException;
    public final void writeFloat(float v) throws IOException;
    public final void writeInt(int v) throws IOException;
    public final void writeLong(long v) throws IOException;
    public final void writeShort(int v) throws IOException;
  
```

RandomAccessFile

```
public final void writeUTF(String str) throws IOException;
}
```

Passed To: javax.imageio.stream.FileImageInputStream.FileImageInputStream(),
javax.imageio.stream.FileImageOutputStream.FileImageOutputStream()

Reader

Java 1.1

java.io

This abstract class is the superclass of all character input streams. It is an analog to `InputStream`, which is the superclass of all byte input streams. `Reader` defines the basic methods that all character output streams provide. `read()` returns a single character or an array (or subarray) of characters, blocking if necessary; it returns `-1` if the end of the stream has been reached. `ready()` returns `true` if there are characters available for reading. If `ready()` returns `true`, the next call to `read()` is guaranteed not to block. `close()` closes the character input stream. `skip()` skips a specified number of characters in the input stream. If `markSupported()` returns `true`, `mark()` marks a position in the stream and, if necessary, creates a look-ahead buffer of the specified size. Future calls to `reset()` restore the stream to the marked position if they occur within the specified look-ahead limit. Note that not all stream types support this mark-and-reset functionality. To create a subclass of `Reader`, you need only implement the three-argument version of `read()` and the `close()` method. Most subclasses implement additional methods, however.

```
public abstract class Reader {
    // Protected Constructors
    protected Reader();
    protected Reader(Object lock);
    // Public Instance Methods
    public abstract void close() throws IOException;
    public void mark(int readAheadLimit) throws IOException;
    public boolean markSupported(); constant
    public int read() throws IOException;
    public int read(char[] cbuf) throws IOException;
    public abstract int read(char[] cbuf, int off, int len) throws IOException;
    public boolean ready() throws IOException; constant
    public void reset() throws IOException;
    public long skip(long n) throws IOException;
    // Protected Instance Fields
    protected Object lock;
}
```

Subclasses: `BufferedReader`, `CharArrayReader`, `FilterReader`, `InputStreamReader`, `PipedReader`, `StringReader`

Passed To: Too many methods to list.

Returned By: `java.awt.datatransfer.DataFlavor.getReaderForText()`,
`java.nio.channels.Channels.newReader()`, `java.sql.Clob.getCharacterStream()`,
`java.sql.ResultSet.getCharacterStream()`, `java.sql.SQLInput.readCharacterStream()`,
`javax.print.Doc.getReaderForText()`, `javax.print.SimpleDoc.getReaderForText()`,
`javax.xml.transform.stream.StreamSource.getReader()`, `org.xml.sax.InputSource.getCharacterStream()`

Type Of: `FilterReader.in`

SequenceInputStream

Java 1.0

java.io

This class provides a way of seamlessly concatenating the data from two or more input streams. It provides an `InputStream` interface to a sequence of `InputStream` objects. Data is read from the streams in the order in which the streams are specified. When the end of one stream is reached, data is automatically read from the next stream. This class might be useful, for example, when implementing an include file facility for a parser of some sort.



```

public class SequenceInputStream extends java.io.InputStream {
// Public Constructors
    public SequenceInputStream(java.util.Enumeration e);
    public SequenceInputStream(java.io.InputStream s1, java.io.InputStream s2);
// Public Methods Overriding InputStream
1.1 public int available() throws IOException;
    public void close() throws IOException;
    public int read() throws IOException;
    public int read(byte[] b, int off, int len) throws IOException;
}
  
```

Serializable

Java 1.1

java.io

serializable

The `Serializable` interface defines no methods or constants. A class should implement this interface simply to indicate that it allows itself to be serialized and deserialized with `ObjectOutputStream.writeObject()` and `ObjectInputStream.readObject()`.

Objects that need special handling during serialization or deserialization may implement one or both of the following methods. Note, however, that these methods are not part of the `Serializable` interface:

```

private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;
  
```

Typically, the `writeObject()` method performs any necessary cleanup or preparation for serialization, invokes the `defaultWriteObject()` method of the `ObjectOutputStream` to serialize the nontransient fields of the class, and optionally writes any additional data that is required. Similarly, the `readObject()` method typically invokes the `defaultReadObject()` method of the `ObjectInputStream`, reads any additional data written by the corresponding `writeObject()` method, and performs any extra initialization required by the object. The `readObject()` method may also register an `ObjectInputValidation` object to validate the object once it is completely deserialized.

```

public interface Serializable {
}
  
```

Implementations: Too many classes to list.

Passed To: Too many methods to list.

Returned By: Too many methods to list.

Type Of: org.omg.CORBA.ValueBaseHolder.value

SerializablePermission

Java 1.2

java.io

serializable permission

This class is a `java.security.Permission` that governs the use of certain sensitive features of serialization. `SerializablePermission` objects have a name, or target, but do not have an action list. The name “enableSubclassImplementation” represents permission to serialize and deserialize objects using subclasses of `ObjectOutputStream` and `ObjectInputStream`. This capability is protected by a permission because malicious code can define object stream subclasses that incorrectly serialize and deserialize objects.

The only other name supported by `SerializablePermission` is “enableSubstitution”, which represents permission for one object to be substituted for another during serialization or deserialization. The `ObjectOutputStream.enableReplaceObject()` and `ObjectInputStream.enableResolveObject()` methods require a permission of this type.

Applications never need to use this class. Programmers writing system-level code may use it, and system administrators configuring security policies should be familiar with it.



```

public final class SerializablePermission extends java.security.BasicPermission {
// Public Constructors
    public SerializablePermission(String name);
    public SerializablePermission(String name, String actions);
}

```

Type Of: `ObjectStreamConstants.SUBCLASS_IMPLEMENTATION_PERMISSION`,
`SUBSTITUTION_PERMISSION`

StreamCorruptedException

Java 1.1

java.io

serializable checked

This exception signals that the data stream being read by an `ObjectInputStream` has been corrupted and does not contain valid serialized object data.



```

public class StreamCorruptedException extends ObjectStreamException {
// Public Constructors
    public StreamCorruptedException();
    public StreamCorruptedException(String reason);
}

```

Thrown By: `ObjectInputStream.readStreamHeader()`, `java.rmi.server.RemoteCall.getResultStream()`

StreamTokenizer

Java 1.0

java.io

This class performs lexical analysis of a specified input stream and breaks the input into tokens. It can be extremely useful when writing simple parsers. `nextToken()` returns the next token in the stream; this is either one of the constants defined by the class (which represent end-of-file, end-of-line, a parsed floating-point number, and a parsed word) or a character value. `pushBack()` pushes the token back onto the stream, so that it is returned by the next call to `nextToken()`. The public variables `sval` and `nval` contain the string and numeric values (if applicable) of the most recently read token. They are

applicable when the returned token is `TT_WORD` or `TT_NUMBER`. `lineno()` returns the current line number.

The remaining methods allow you to specify how tokens are recognized. `wordChars()` specifies a range of characters that should be treated as parts of words. `whitespaceChars()` specifies a range of characters that serve to delimit tokens. `ordinaryChars()` and `ordinaryChar()` specify characters that are never part of tokens and should be returned as-is. `resetSyntax()` makes all characters ordinary. `eollsSignificant()` specifies whether end-of-line is significant. If so, the `TT_EOL` constant is returned for end-of-lines; otherwise, they are treated as whitespace. `commentChar()` specifies a character that begins a comment that lasts until the end of the line. No characters in the comment are returned. `slashStarComments()` and `slashSlashComments()` specify whether the `StreamTokenizer` should recognize C- and C++-style comments. If so, no part of the comment is returned as a token. `quoteChar()` specifies a character used to delimit strings. When a string token is parsed, the quote character is returned as the token value, and the body of the string is stored in the `sval` variable. `lowerCaseMode()` specifies whether `TT_WORD` tokens should be converted to all lowercase characters before being stored in `sval`. `parseNumbers()` specifies that the `StreamTokenizer` should recognize and return double-precision, floating-point number tokens.

```
public class StreamTokenizer {
// Public Constructors
# public StreamTokenizer(java.io.InputStream is);
1.1 public StreamTokenizer(Reader r);
// Public Constants
public static final int TT_EOF;           =-1
public static final int TT_EOL;           =10
public static final int TT_NUMBER;        =-2
public static final int TT_WORD;          =-3
// Public Instance Methods
public void commentChar(int ch);
public void eollsSignificant(boolean flag);
public int lineno();
public void lowerCaseMode(boolean fl);
public int nextToken() throws IOException;
public void ordinaryChar(int ch);
public void ordinaryChars(int low, int hi);
public void parseNumbers();
public void pushBack();
public void quoteChar(int ch);
public void resetSyntax();
public void slashSlashComments(boolean flag);
public void slashStarComments(boolean flag);
public void whitespaceChars(int low, int hi);
public void wordChars(int low, int hi);
// Public Methods Overriding Object
public String toString();
// Public Instance Fields
public double nval;
public String sval;
public int ttype;
}
```

StringBufferInputStream

Java 1.0; Deprecated in Java 1.1

java.io

This class is a subclass of `InputStream` in which input bytes come from the characters of a specified `String` object. This class does not correctly convert the characters of a `StringBuffer` into bytes and is deprecated as of Java 1.1. Use `StringReader` instead to convert characters into bytes or use `ByteArrayInputStream` to read bytes from an array of bytes.

```

Object --> InputStream --> StringBufferInputStream

public class StringBufferInputStream extends java.io.InputStream {
    // Public Constructors
    public StringBufferInputStream(String s);
    // Public Methods Overriding InputStream
    public int available();                                synchronized
    public int read();                                    synchronized
    public int read(byte[] b, int off, int len);            synchronized
    public void reset();                                    synchronized
    public long skip(long n);                                synchronized
    // Protected Instance Fields
    protected String buffer;
    protected int count;
    protected int pos;
}

```

StringReader

Java 1.1

java.io

This class is a character input stream that uses a `String` object as the source of the characters it returns. When you create a `StringReader`, you must specify the `String` to read from. `StringReader` defines the normal `Reader` methods and supports `mark()` and `reset()`. If `reset()` is called before `mark()` has been called, the stream is reset to the beginning of the specified string. `StringReader` is a character stream analog to `StringBufferInputStream`, which is deprecated as of Java 1.1. `StringReader` is also similar to `CharArrayReader`.

```

Object --> Reader --> StringReader

public class StringReader extends Reader {
    // Public Constructors
    public StringReader(String s);
    // Public Methods Overriding Reader
    public void close();
    public void mark(int readAheadLimit) throws IOException;
    public boolean markSupported();                        constant
    public int read() throws IOException;
    public int read(char[] cbuf, int off, int len) throws IOException;
    public boolean ready() throws IOException;
    public void reset() throws IOException;
    public long skip(long ns) throws IOException;
}

```

StringWriter

Java 1.1

java.io

This class is a character output stream that uses an internal `StringBuffer` object as the destination of the characters written to the stream. When you create a `StringWriter`, you may optionally specify an initial size for the `StringBuffer`, but you do not specify the `StringBuffer`.

UnsupportedEncodingException

itself; it is managed internally by the `StringWriter` and grows as necessary to accommodate the characters written to it. `StringWriter` defines the standard `write()`, `flush()`, and `close()` methods all `Writer` subclasses define, as well as two methods to obtain the characters that have been written to the stream's internal buffer. `toString()` returns the contents of the internal buffer as a `String`, and `getBuffer()` returns the buffer itself. Note that `getBuffer()` returns a reference to the actual internal buffer, not a copy of it, so any changes you make to the buffer are reflected in subsequent calls to `toString()`. `StringWriter` is quite similar to `CharArrayWriter`, but does not have a byte-stream analog.

```
Object --> Writer --> StringWriter

public class StringWriter extends Writer {
// Public Constructors
    public StringWriter();
    public StringWriter(int initialSize);
// Public Instance Methods
    public StringBuffer getBuffer();
// Public Methods Overriding Writer
    public void close() throws IOException;           empty
    public void flush();                             empty
    public void write(int c);
    public void write(String str);
    public void write(String str, int off, int len);
    public void write(char[] cbuf, int off, int len);
// Public Methods Overriding Object
    public String toString();
}
```

SyncFailedException

Java 1.1

java.io

serializable checked

This exception signals that a call to `FileDescriptor.sync()` did not complete successfully.

```
Object --> Throwable --> Exception --> IOException --> SyncFailedException
                                     |
                                     +-- Serializable

public class SyncFailedException extends IOException {
// Public Constructors
    public SyncFailedException(String desc);
}
```

Thrown By: `FileDescriptor.sync()`

UnsupportedEncodingException

Java 1.1

java.io

serializable checked

This exception signals that a requested character encoding is not supported by the current Java Virtual Machine.

```
Object --> Throwable --> Exception --> IOException --> UnsupportedEncodingException
                                     |
                                     +-- Serializable

public class UnsupportedEncodingException extends IOException {
// Public Constructors
    public UnsupportedEncodingException();
}
```

UnsupportedEncodingException

```
public UnsupportedEncodingException(String s);
}
```

Thrown By: `ByteArrayOutputStream.toString()`, `InputStreamReader.InputStreamReader()`, `OutputStreamWriter.OutputStreamWriter()`, `PrintStream.PrintStream()`, `String.getBytes()`, `String()`, `java.net.URLDecoder.decode()`, `java.net.URLEncoder.encode()`, `java.util.logging.Handler.setEncoding()`, `java.util.logging.StreamHandler.setEncoding()`

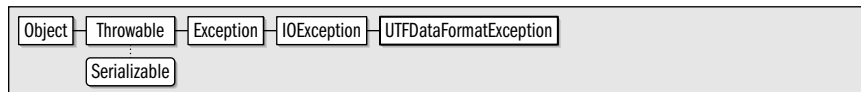
UTFDataFormatException

Java 1.0

java.io

serializable checked

This exception is an `IOException` that signals that a malformed UTF-8 string has been encountered by a class that implements the `DataInput` interface. UTF-8 is an ASCII-compatible transformation format for Unicode characters that is often used to store and transmit Unicode text.



```
public class UTFDataFormatException extends IOException {
    // Public Constructors
    public UTFDataFormatException();
    public UTFDataFormatException(String s);
}
```

WriteAbortedException

Java 1.1

java.io

serializable checked

This exception is thrown when reading a stream of data that is incomplete because an exception was thrown while it was being written. The `detail` field may contain the exception that terminated the output stream. In Java 1.4 and later, this exception can also be obtained with the standard `Throwable` `getCause()` method. The `getMessage()` method has been overridden to include the message of this `detail` exception, if any.



```
public class WriteAbortedException extends ObjectStreamException {
    // Public Constructors
    public WriteAbortedException(String s, Exception ex);
    // Public Methods Overriding Throwable
    1.4 public Throwable getCause();
    public String getMessage();
    // Public Instance Fields
    public Exception detail;
}
```

Writer

Java 1.1

java.io

This abstract class is the superclass of all character output streams. It is an analog to `OutputStream`, which is the superclass of all byte output streams. `Writer` defines the basic `write()`, `flush()`, and `close()` methods all character output streams provide. The five versions of the `write()` method write a single character, a character array or subarray, or a string or substring to the destination of the stream. The most general version of this

method—the one that writes a specified portion of a character array—is abstract and must be implemented by all subclasses. By default, the other `write()` methods are implemented in terms of this abstract one. The `flush()` method is another abstract method all subclasses must implement. It should force any output buffered by the stream to be written to its destination. If that destination is itself a character or byte output stream, it should invoke the `flush()` method of the destination stream as well. The `close()` method is also abstract. A subclass must implement this method so that it flushes and then closes the current stream and also closes whatever destination stream it is connected to. Once the stream is closed, any future calls to `write()` or `flush()` should throw an `IOException`.

```
public abstract class Writer {
    // Protected Constructors
    protected Writer();
    protected Writer(Object lock);
    // Public Instance Methods
    public abstract void close() throws IOException;
    public abstract void flush() throws IOException;
    public void write(String str) throws IOException;
    public void write(char[] cbuf) throws IOException;
    public void write(int c) throws IOException;
    public void write(String str, int off, int len) throws IOException;
    public abstract void write(char[] cbuf, int off, int len) throws IOException;
    // Protected Instance Fields
    protected Object lock;
}
```

Subclasses: `BufferedWriter`, `CharArrayWriter`, `FilterWriter`, `OutputStreamWriter`, `PipedWriter`, `PrintWriter`, `StringWriter`

Passed To: Too many methods to list.

Returned By: `java.nio.channels.Channels.newWriter()`, `java.sql.Clob.setCharacterStream()`, `javax.swing.text.AbstractWriter.getWriter()`, `javax.xml.transform.stream.StreamResult.getWriter()`

Type Of: `FilterWriter.out`, `PrintWriter.out`