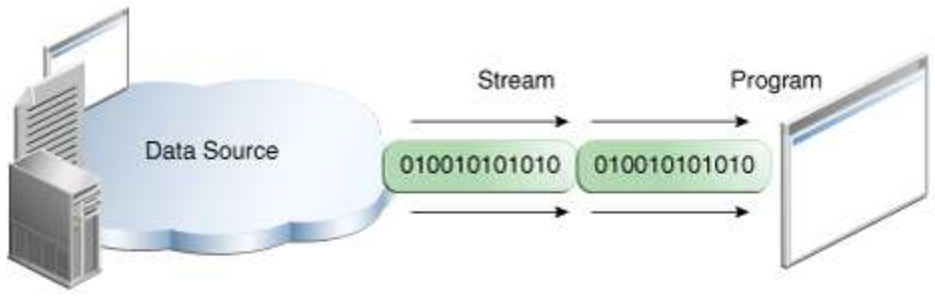
H2K Infosys, LLC
Dream, Strive & Achieve Victory

www.H2KINFOSYS.com
USA: 770-777-1269
Training@H2KInfosys.com
UK : (020) 3371 7615

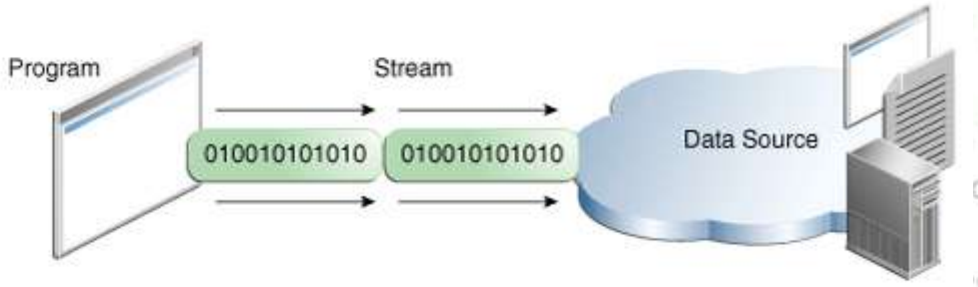
IO Streams

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time:



A program uses an *output stream* to write data to a destination, one item at time:



Byte Streams

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from **InputStream** and **OutputStream**.

We'll explore **FileInputStream** and **FileOutputStream** by examining an example program named CopyBytes, which uses byte streams to copy xanadu.txt, one byte at a time.

Closing a stream when it's no longer needed is very important — so important that `CopyBytes` uses a `finally` block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("h2kFileIn.txt");
            out = new FileOutputStream("h2kFileOut.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Character Streams

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter. The CopyCharacters example illustrates these classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("h2kFileIn.txt");
            outputStream = new FileWriter("h2kFileOut.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end.

Let's modify the `CopyCharacters` example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, `BufferedReader` and `PrintWriter`. We'll explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new
FileReader("h2kFileIn.txt"));
            outputStream = new PrintWriter(new FileWriter("h2kFileOut.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Buffered Streams

Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

```
inputStream = new BufferedReader(new FileReader("h2kFileIn.txt"));
outputStream = new BufferedWriter(new FileWriter("h2kFileOut.txt"));
```

Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer.

To flush a stream manually, invoke its `flush` method. The `flush` method is valid on any output stream, but has no effect unless the stream is buffered.

Object Streams

The object stream classes are ObjectInputStream (in) and ObjectOutputStream (out).

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a ClassNotFoundException.

```
Object ob = new Object();
out.writeObject(ob);
```

Reading Object:

```
Object ob1 = in.readObject();
```

“File” in Java:

```
public class File extends Object implements Serializable, Comparable<File>
```

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion etc.
File object has enormous list of methods for your help.

Methods	Description
public String getName()	Returns the name of the file or directory denoted by this abstract pathname.
public String getParent()	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
public String getPath()	Converts this abstract pathname into a pathname string.
public boolean isAbsolute()	Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise
public boolean canRead()	Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
public boolean canWrite()	Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
public boolean exists()	Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise
public boolean isDirectory()	Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
public boolean createNewFile() throws IOException	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.
public boolean delete()	Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.
public void deleteOnExit()	Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
public File[] listFiles()	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
public boolean mkdir()	Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise
public boolean renameTo(File dest)	Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise