Java IO Streams

File Class

- It does not specify how the information is retrieved from or stored in files.
- Describe the properties of file
- Used to obtain the information associated with the file such as permissions, time,date, path and so on.

Example

File f1=new File(“D:/sample.txt”);

What is IO Stream?

- *I/O Stream* represents input source or output destination.
- Stream can represent **different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays**.
- Streams support different kinds of data, **including simple bytes, primitive data types, localized characters, and objects.**
- Some streams simply pass on data; others manipulate and transform the data in useful ways.
- Stream is a sequence of data.
- Program uses an *input stream* to read data from a source, one item at a time
- Program uses an *output stream* to write data to a destination, one item at time

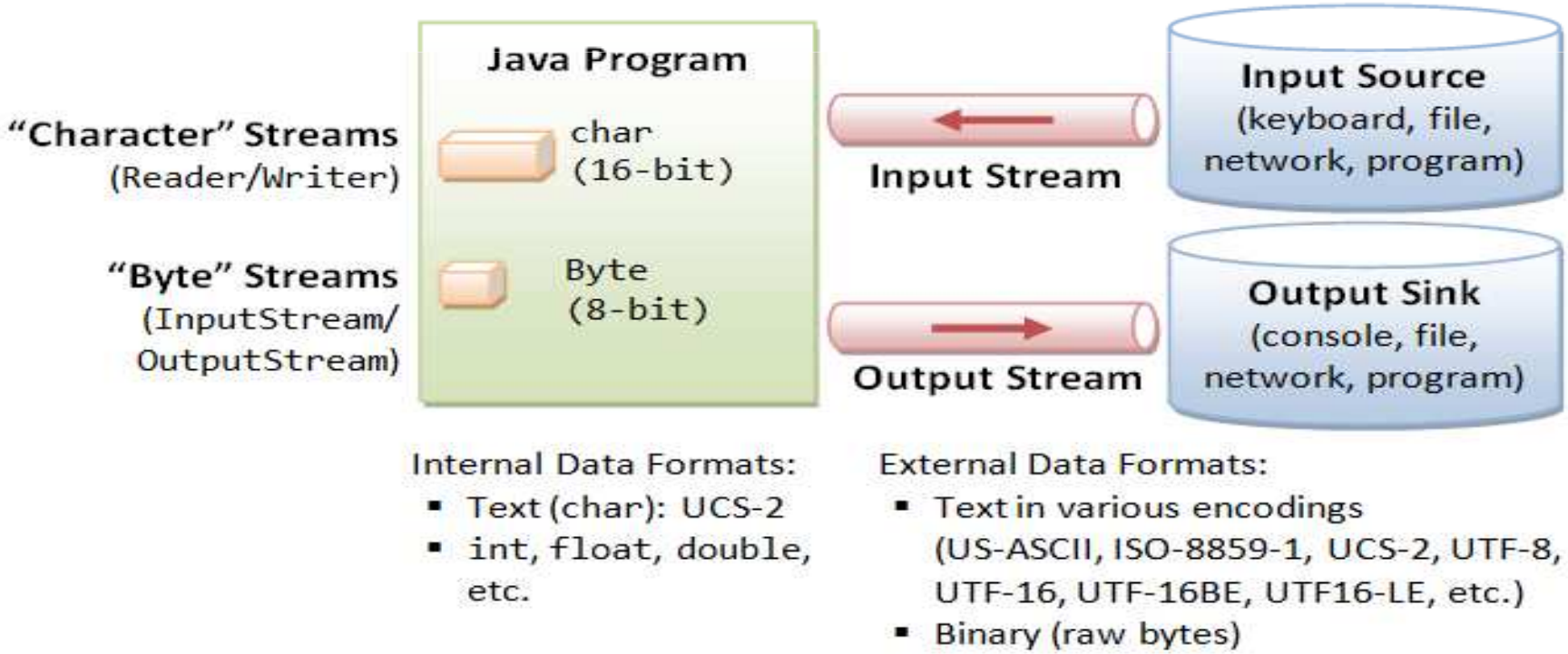
...Contd

When to use Character Stream over Byte Stream?

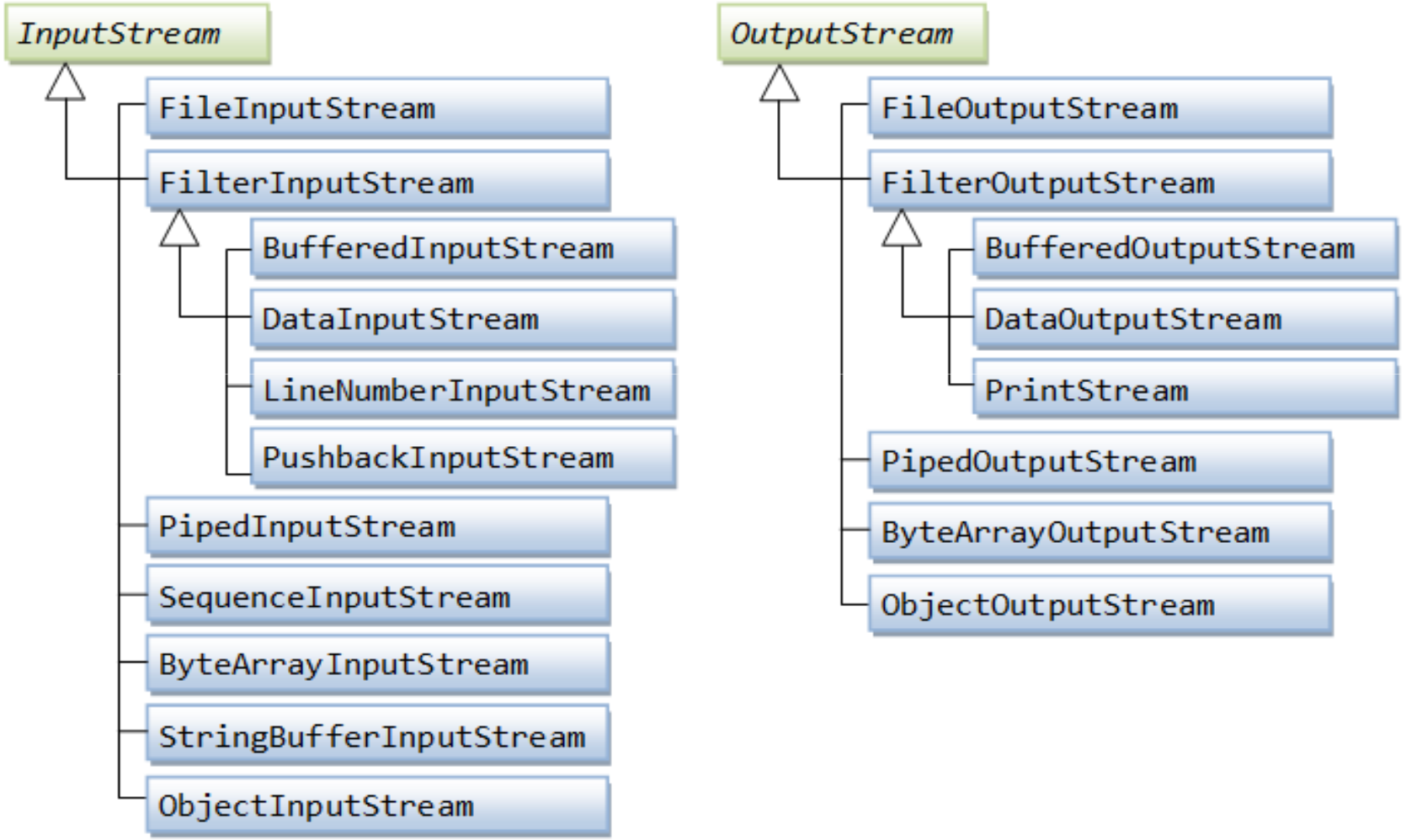
In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

When to use Byte Stream over Character Stream?

Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.



Hierarchy of I/O Streams



Example for character streams

```
import java.io.*;
public class GfG
{
    public static void main(String[] args) throws IOException
    {       FileReader sourceStream = null;
            try
            {
            sourceStream = new FileReader("test.txt");

            int temp;
            while ((temp = sourceStream.read()) != -1)
            System.out.println((char)temp);
            }
            finally
            {
            if (sourceStream != null)
            sourceStream.close();
            }
      }
}
```

Byte Streams

- Java byte streams are used to perform input and output of 8-bit bytes.
- All byte stream classes are descended from InputStream and OutputStream.
- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

Copying the content of an input file into an output file

```
public class CopyFile {
 public static void main(String args[]) throws
      IOException  {
       FileInputStream in = null;
      FileOutputStream out = null;
       try {
          in = new FileInputStream("input.txt");
          out = new
      FileOutputStream("output.txt");
          int c;
          while ((c = in.read()) != -1)
      out.write(c);
      } finally {
         if (in != null)  in.close();
         if (out != null) out.close();
      }
  }
}
```

The finally block will execute whether or not an exception is thrown.

Output:

Contents of “input.txt” is copied to “output.txt”

ByteArrayInputStream class

Why ByteArrayStream?

- ByteArrayInputStream is like wrapper which protects underlying array from external modification
- It has high order read ,mark ,skip functions
- A stream is advantageous because
  - Need not have all bytes in memory at the same time
  - Convenient if the size of the data is large and can easily be handled in small chunks.
- This stream utilizes a byte array as the source
- This class has two constructors
  - ByteArrayInputStream(byte array[])
  - ByteArrayInputStream(byte array[], int start, int numBytes)
- Creates stream from a sub set of array
- This stream implements both mark and skip functionalities

Example

Creating Streams

```
String tmp="abcdefghijklmnopqrstuvwxyz";
byte b[]=tmp.getBytes();
ByteArrayInputStream in1 =new
      ByteArrayInputStream(b);
ByteArrayInputStream in2 =new
      ByteArrayInputStream(b,0,5);
```

Reading from the stream

```
while(c=in.read()!=-1)
{
  System.out.println((char) c);
}
```

ByteArrayIOutputStream class

Creating Streams

```
ByteArrayOutputStream out1 =new
    ByteArrayOutputStream(b);
```

Writing into the stream

```
String tmp="abcdefghijklmnopqrstuvwxyz";
byte b[]=tmp.getBytes();
out1.write(b);
```

Buffered IO Streams

- Most of the examples we've seen so far use *unbuffered* I/O.
- Each read or write request is handled directly by the underlying OS.
- Makes the program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
- To reduce this kind of overhead, the Java platform implements *buffered* I/O streams.

…Contd

- Buffered input streams read data from a memory area known as a *buffer* and 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.
- The following code converts an unbuffered stream into a buffered stream using the wrapping

Example

```
ByteArrayInputStream f=new ByteArrayInputStream(b);
BufferedInputStream in=new BufferedIputStream(f);
```

Example for mark and reset in Buffered Stream

Creation

BufferedInputStream in=new BufferedIputStream(f);

BufferedInputStream in=new BufferedIputStream(f, 100);

Input

String s="This is a © copyright symbol"+" But this is &copy not";

byte b[]=s.getBytes();

…Contd

Output

This is a © copyright symbol But this is &copy not

- How do we mark?
  - f.mark(32), where f is the buffered stream
- Marks next 32 bytes read
- How do we reset?
  - f.reset()

PushbackInputStream

- One of the novel uses of buffering is the implementation of pushback.
- Pushback is used on an input stream to allow a byte to be read and then returned (that is, “pushed back”) to the stream.
- The PushbackInputStream class implements this idea.
- Pushback is used on an input stream to allow a byte to be read and then returned to the stream
- How do we create the stream?

  ByteArrayInputStream f=new ByteArrayInputStream(b);

  PushbackInputStream in=new PushbackInputStream(f);

Example

Input

- String s="if (a==4) a=0;"

Output

- If (a .eq. 4) a<-0;

Steps to implement

- Get byte array from the string
- Convert into pushback stream
- Read the char by char
- Print the character
- If the character read is =, read another char to see '=' if so print == by .eq.
- Otherwise print <- and return the char to the stream

DataInputStream & DataOutputStream

- It enables the user to write or read primitive data to or from a stream

Creating Stream

```
DataOutputStream out;
out = new DataOutputStream(new
   FileOutputStream("input.dat");
```

…Contd

- **Writing into the stream**
  - out.writeDouble(10.9);
  - out.writeInt(1000);
  - out.writeBoolean(true);
- **Reading from the stream**
  - DataInputStream in;
  - out = new DataInputStream(new FileInputStream(“input.dat”);

…contd

- double d=in.readDouble();
- int i=in.readInt();
- boolean b=in.readBoolean();

- System.out.println(“The values are:”+d+” “+ I +” “ +b);

SequenceInputStream

- Java SequenceInputStream class is used **to read data from multiple streams**. It reads data of streams one by one.
- Constructors of SequenceInputStream class:

  SequenceInputStream(InputStream s1, InputStream s2)
- Creates a new input stream by reading the data of two input streams in order, first s1 and then s2

  .

Example

```
Import java.io.*;
public class Simple
{
    public static void main(String args[]) throws Exception
    {
        FileInputStream fin1=new FileInputStream("f1.txt");
        FileInputStream fin2=new FileInputStream("f2.txt");
        SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
        int i;
        while((i=sis.read())!=-1)
        {
            System.out.println((char)i);
        }
        sis.close();
        fin1.close();
        fin2.close();
        }
}
```