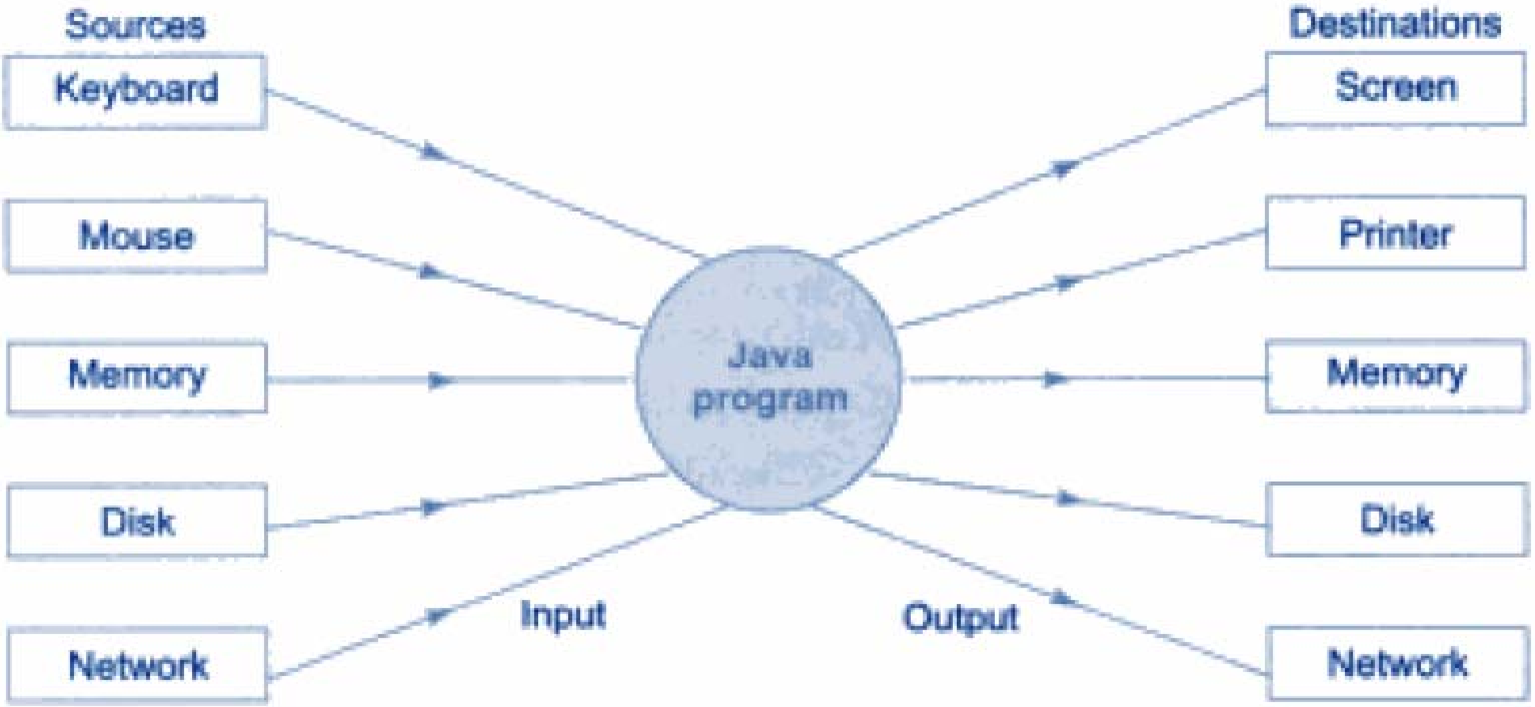
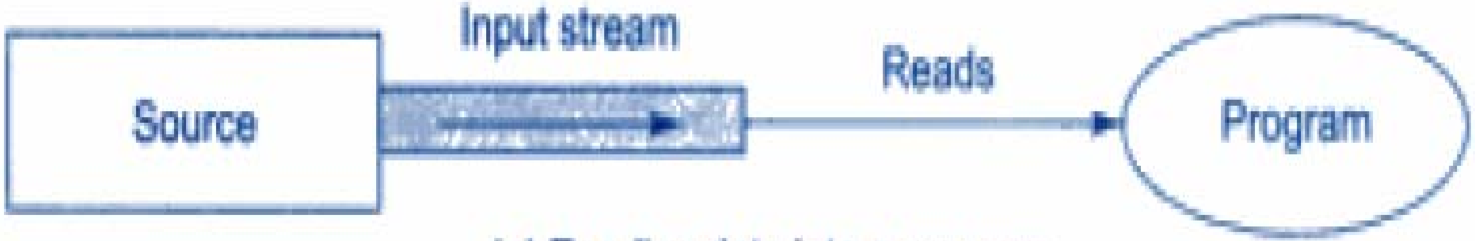
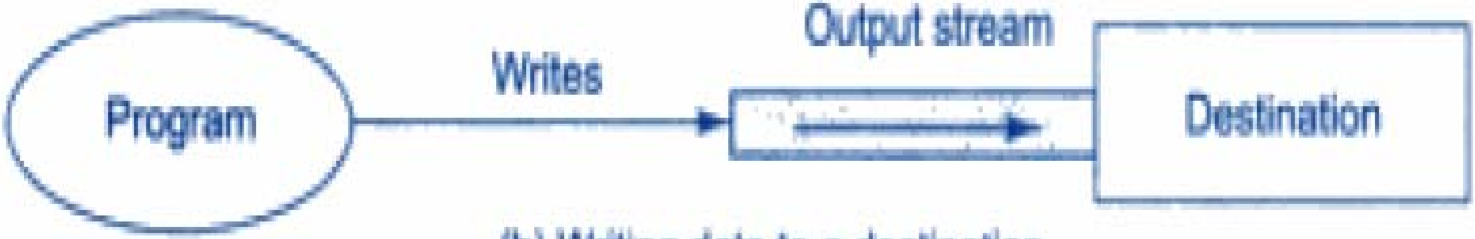
Streams

- ***Stream***: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer (path) between the data source and destination
- ***Input stream***: a stream that provides input to a program
  - `System.in` is an input stream
- ***Output stream***: a stream that accepts output from a program
  - `System.out` is an output stream
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `System.in` connects a program to the keyboard

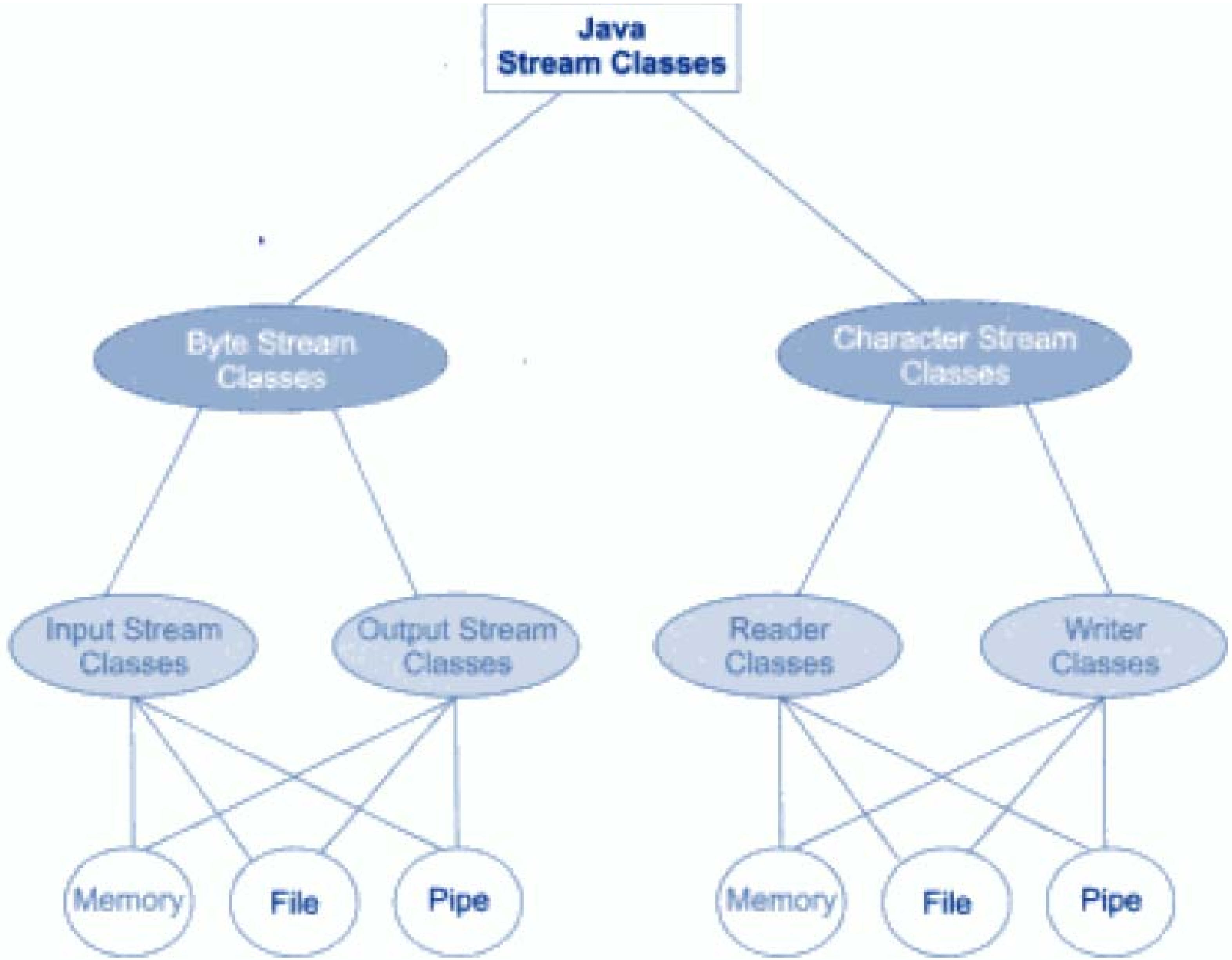

Sources
Keyboard
Mouse
Memory
Disk
Network
Input
Java program
Output
Destinations
Screen
Printer
Memory
Disk
Network



(a) Reading data into a program



(b) Writing data to a destination


Java Stream Classes
Byte Stream Classes
Character Stream Classes
Input Stream Classes
Output Stream Classes
Reader Classes
Writer Classes
Memory
File
Pipe
Memory
File
Pipe

Byte Streams and Character Streams

Java 2 defines two types of streams: byte and character.

Byte streams **provide a convenient means for handling input and output of bytes.**

Byte streams are used, for example, when reading or writing binary data.

Character streams **provide a convenient means for handling input and output of characters.**

They use Unicode and, therefore, can be internationalized.

Also, in some cases, character streams are more efficient than byte streams.

1.The Byte Stream Classes

➢Byte streams are defined by using two class hierarchies.

➢At the top are two abstract classes: InputStream and OutputStream.

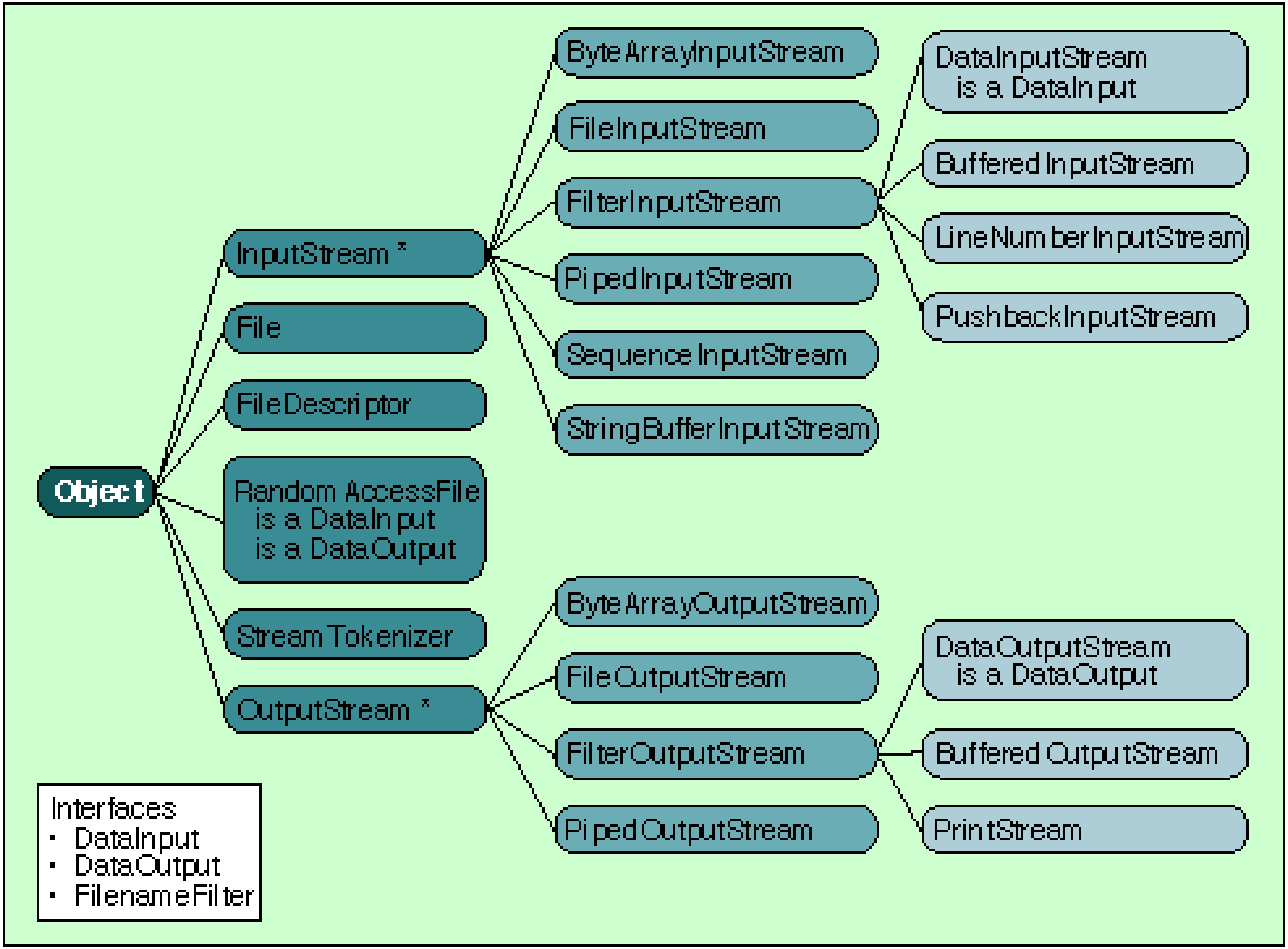
➢Each of these abstract classes has several concrete subclasses, that handle the differences between various devices, such as disk files, network connections, and even memory buffers

➢The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement.

➢Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data.

➢Both methods are declared as abstract inside InputStream and OutputStream.

➢They are overridden by derived stream classes.


Object
InputStream *
File
FileDescriptor
RandomAccessFile
is a DataInput
is a DataOutput
StreamTokenizer
OutputStream *
ByteArrayInputStream
FileInputStream
FilterInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
DataInputStream
is a DataInput
BufferedInputStream
LineNumberInputStream
PushbackInputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
PipedOutputStream
DataOutputStream
is a DataOutput
BufferedOutputStream
PrintStream
Interfaces
▪ DataInput
▪ DataOutput
▪ FilenameFilter

2. The Character Stream Classes

- Character streams are defined by using two class hierarchies.

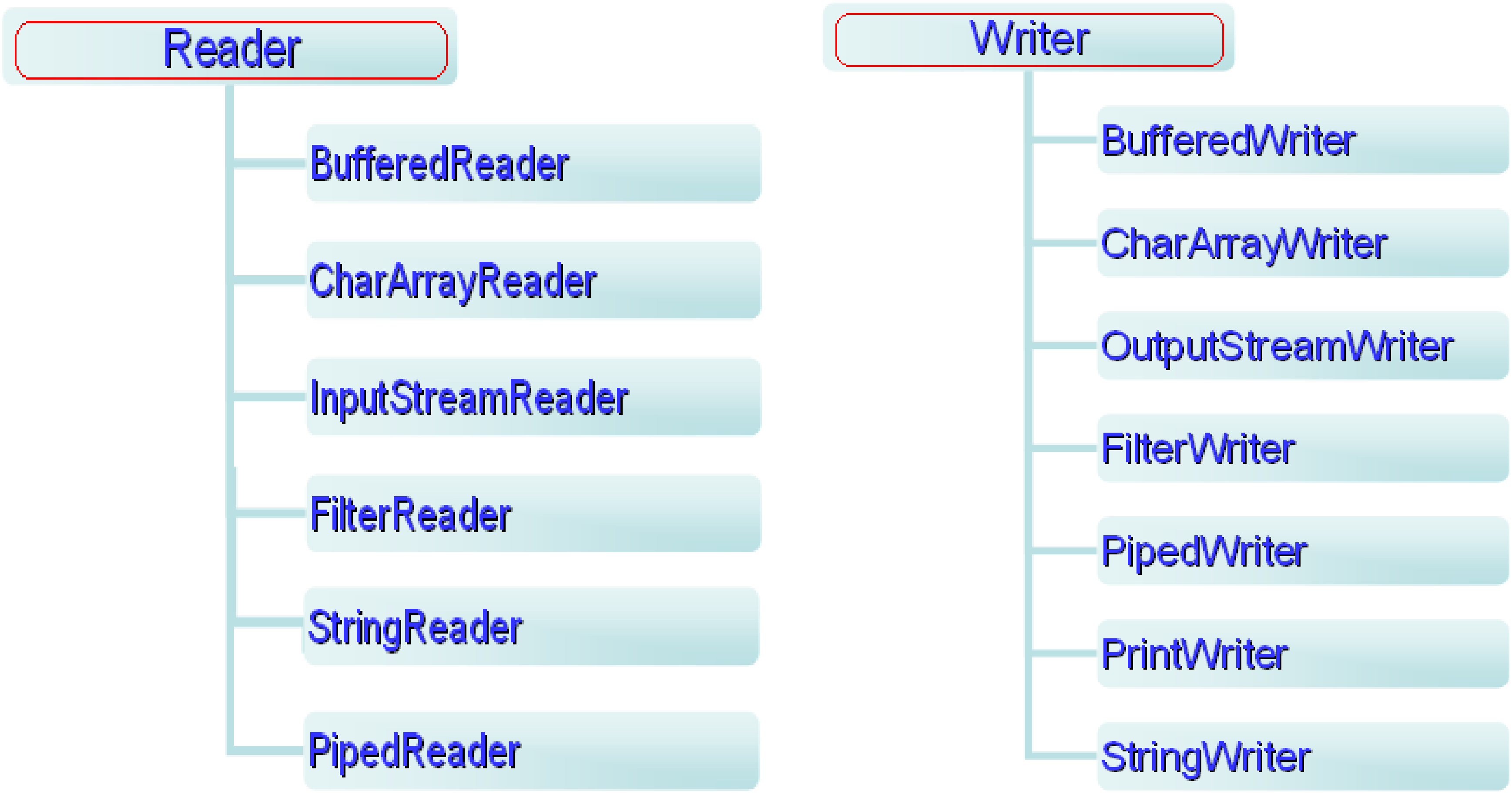
- At the top are two abstract classes, Reader and Writer.

- These abstract classes handle Unicode character streams.

- The abstract classes Reader and Writer define several key methods that the other stream classes implement.

- Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively.

- These methods are overridden by derived stream classes.



Reader

- BufferedReader
- CharArrayReader
- InputStreamReader
- FilterReader
- StringReader
- PipedReader

Writer

- BufferedWriter
- CharArrayWriter
- OutputStreamWriter
- FilterWriter
- PipedWriter
- PrintWriter
- StringWriter

```
import java.io.*;

public class CopyFile {
private static void copyfile() {
try {
File f1 = new File("D:/Fall 2010-11/MSSoftwareEngineering/Myjava/Files/file.txt");
File f2 = new File("D:/Fall 2010-11/MS Software Engineering/Myjava/Files/new.txt");
      InputStream in = new FileInputStream(f1);
      OutputStream out = new FileOutputStream(f2, true);
      byte[] buf = new byte[1024];
      int len;
      while ((len = in.read(buf)) > 0) {
        out.write(buf, 0, len);
      }
      in.close();
      out.close();
      System.out.println("File copied.");
    } catch (Exception ex) {
      System.out.println(ex);
    }
  }
```

```
public static void main(String[] args) {
    copyfile();
  }
}
```

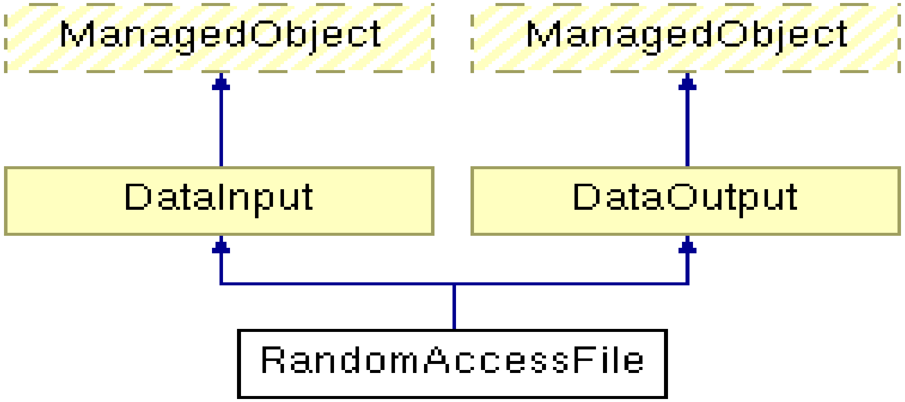
```
// Read a character from console(command prompt) using a BufferedReader.


import java.io.*;
class BRRead {
public static void main(String args[])
throws IOException
{
char c;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");

// read characters
do {
c = (char) br.read();   // prototype: int read() throws IOException ; -1 at eof
System.out.println(c);
} while(c != 'q');
}
}
```

RandomAccessFile

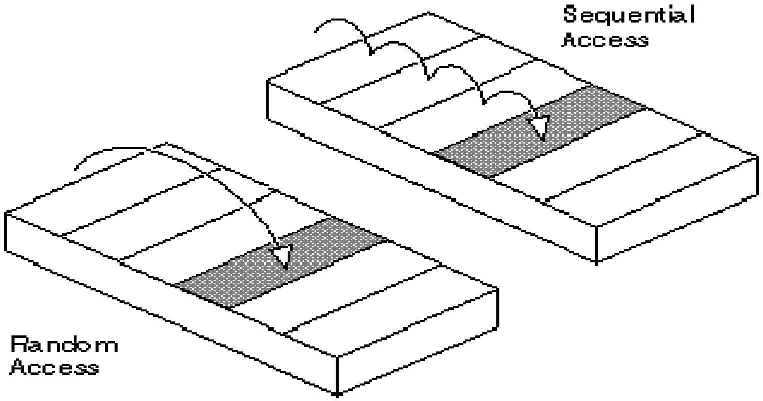
RandomAccessFile encapsulates a random-access file. It is not derived from InputStream or OutputStream.

Random access files **permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.**

Instead, it implements the interfaces DataInput and DataOutput, which define the basic I/O methods.



How it is different from a sequential file?



- ➢A random-access data file enables you to read or write information anywhere in the file.

- ➢In a sequential-access file, you can only read and write information sequentially, starting from the beginning of the file.

- ➢If you are always accessing information in the same order, a sequential-access file is faster.

- ➢If you tend to access information randomly, random access is better.

- ➢Random access is sometimes called *direct access.*

- ➢.Disks are random access media, whereas tapes are sequential access medi

Creating a RandomAccessFile

Before working with the RandomAccessFile class you must instantiate it. Here is how that looks:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw);
```

Notice the second input parameter to the constructor: "rw". This is the mode you want to open file in. "rw" means read/write mode.

Moving Around a RandomAccessFile

- To read or write at a specific location in a RandomAccessFile you must first position the file pointer at the location to read or write.

- This is done using the seek() method.

- The current position of the file pointer can be obtained by calling the getFilePointer() method.

- Here is a simple example:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw);
file.seek(200);
long pointer = file.getFilePointer();
file.close();
```

Reading from a RandomAccessFile

- Reading from a RandomAccessFile is done using one of its many read() methods.

- Here is a simple example:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw);
int aByte = file.read();
file.close();
```

- The read() method reads the byte located a the position in the file currently pointed to by the file pointer in the RandomAccessFile instance.

- The read() method increments the file pointer to point to the next byte in the file after the byte just read!

- This means that you can continue to call read() without having to manually move the file pointer.

Writing to a RandomAccessFile

- Writing to a RandomAccessFile can be done using one it its many write() methods.

- Here is a simple example:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw");
file.write("Hello World".getBytes());
file.close();
```

- Just like with the read() method the write() method advances the file pointer after being called.

- That way you don't have to constantly move the file pointer to write data to a new location in the file.

Advantages or The need of RandomAccessFile

- **File constructed in a manner in which records may be placed in a random order; also called *direct access file.***
- **Each record in a random access file has associated with it a relative index number.**
- **Whenever a record is read from a random access file, a computer program must produce a relative index number for this record in order to locate the record in the file.**
- **This type of file design offers the following advantages:**

(1) it provides rapid access to the desired information. In a decision-making environment where information is needed quickly, random access is a requisite to rapid retrieval;

(2) it is efficient for retrieving a relatively few records at a time; and

(3) it provides a method of keeping files up to date as transactions or events occur.

RandomAccessFile Example:

```
import java.io.File;
import java.io.RandomAccessFile;

public class FileSeek {
	public static void main(String[] args) throws Exception {
		File file = new
		    File("D:/Fall 2010-11/MS SoftwareEngineering/Myjava/Files/abc.txt");
		RandomAccessFile access = new RandomAccessFile(file, "rw");
		System.out.println(access.readLine());
		access.seek(file.length());
		access.writeBytes("Truth is more important than facts");
		access.close();
	}
}
```

Character Streams Example:

To pull successive characters from the stream, we then call the Reader's read() method:

```
InputStream in = new FileInputStream("charfile.txt");
Reader r = new InputStreamReader(in, "US-ASCII");
int intch;
while ((intch = r.read()) != -1)
{
 char ch = (char) intch;
// ...
 }
```

```
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("xanadu.txt");
outputStream = new FileWriter("characteroutput.txt");
int c;
while ((c = inputStream.read()) != -1)
{
outputStream.write(c);
}
}
```

```
finally {
 if (inputStream != null)
    {
       inputStream.close();
     }
if (outputStream != null)
     {
       outputStream.close();
          } } } }
```

INPUT/OUTPUT EXCEPTIONS

EOFException – Signals that an end of the file or end of stream has been reached unexpectedly during input

FileNotFoundException – Informs that a file could not be found

InterruptedIOException – Warns that an I/O operations has been interrupted

IOException – Signals that an I/O exception of some sort has occured

The FILE Class

- **It deals directly with files and the file system.**

- **The File class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.**

- **A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.**

- **A directory is also considered as a File**

The following constructors can be used to create File objects:

File(String *directoryPath*)
File(String *directoryPath,* String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

- ***directoryPath*** **is the path name of the file,**
- ***filename*** **is the name of the file,**
- ***dirObj*** **is a File object that specifies a directory, and**
- ***uriObj*** **is a URI object that describes a file.**

```java
// Demonstrate File.
import java.io.File;
class FileDemo {
static void p(String s) {
System.out.println(s);
}
public static void main(String args[]) {
File f1 = new File(“java/COPYRIGHT");
p("File Name: " + f1.getName());
p("Path: " + f1.getPath());
p("Abs Path: " + f1.getAbsolutePath());
p("Parent: " + f1.getParent());
p(f1.exists() ? "exists" : "does not exist");
p(f1.canWrite() ? "is writeable" : "is not writeable");
p(f1.canRead() ? "is readable" : "is not readable");
p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
p(f1.isFile() ? "is normal file" : "might be a named pipe");
p(f1.isAbsolute() ? "is absolute" : "is not absolute");
p("File last modified: " + f1.lastModified());
p("File size: " + f1.length() + " Bytes");
}
}
```

```
File Name: COPYRIGHT
Path: /java/COPYRIGHT
Abs Path: /java/COPYRIGHT
Parent: /java
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified: 812465204000
File size: 695 Bytes
```

```
// Read a string from console(command prompt) using a BufferedReader.

import java.io.*;
class BRReadLines {
public static void main(String args[ ]) throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(newInputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do {
str = br.readLine(); // String readLine() throws IOException
System.out.println(str);
} while(!str.equals("stop"));
}
}
```

Directories

- **A directory is a File that contains a list of other files and directories.**

- **When you create a File object and it is a directory, the isDirectory( ) method will return true.**

- **By calling list( ) on that object to extract the list of other files and directories inside.**

- **String[ ] list( )**

- **The list of files is returned in an array of String objects.**

```
// Using directories.
import java.io.File;
class DirList {
public static void main(String args[]) {
String dirname = "/java";
File f1 = new File(dirname);
if (f1.isDirectory()) {
System.out.println("Directory of " + dirname);
String s[] = f1.list();
for (int i=0; i < s.length; i++) {
File f = new File(dirname + "/" + s[i]);
if (f.isDirectory()) {
System.out.println(s[i] + " is a directory");
}
else {
System.out.println(s[i] + " is a file");
}
}
} else {
System.out.println(dirname + " is not a directory");
}
}
}
```