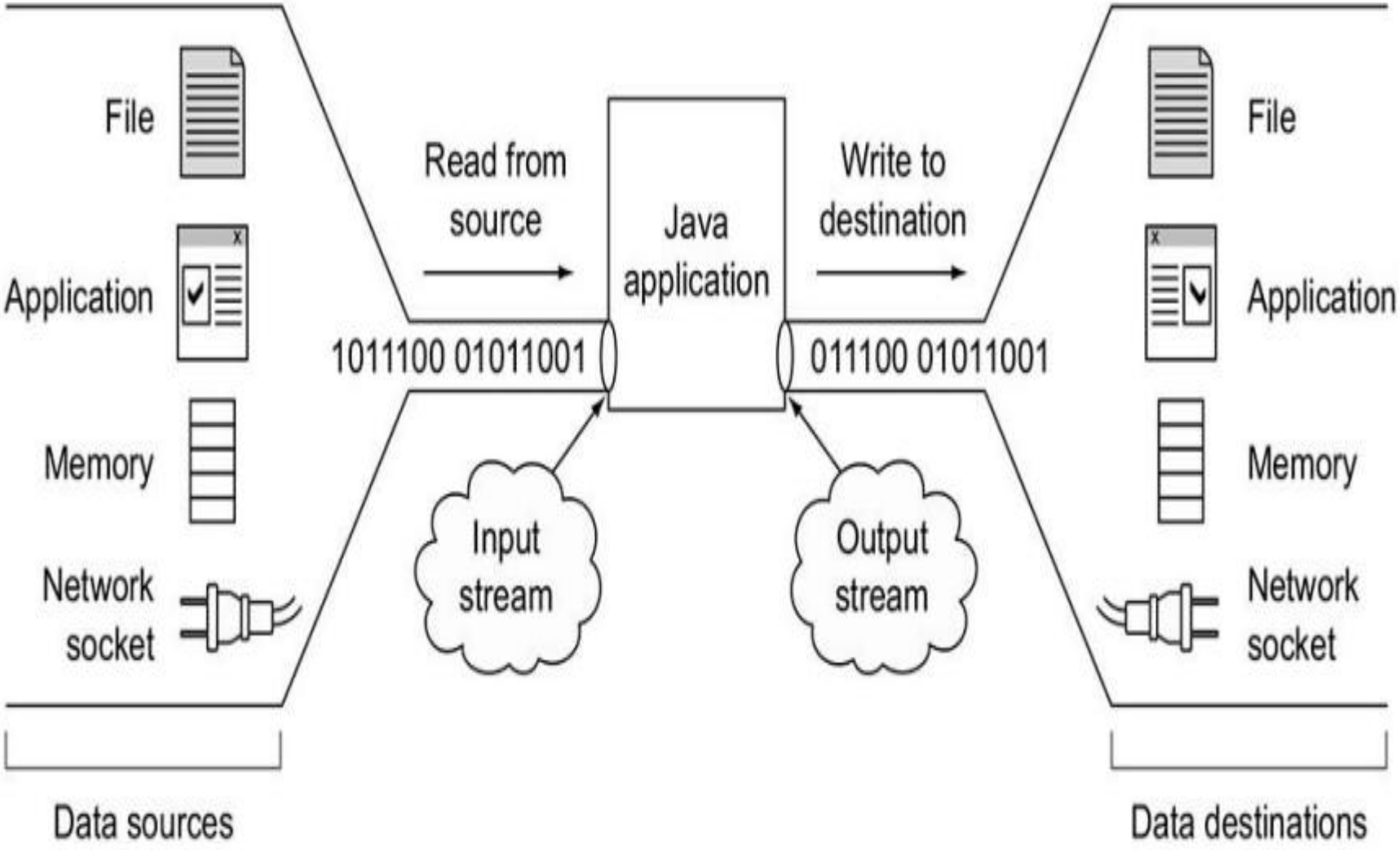
I/O streams

Introduction

- So far we have used variables and arrays for storing data inside the programs. This approach poses the following limitations:
  - The data is lost when variable goes out of scope or when the program terminates. That is data is stored in temporary/mail memory is released when program terminates.
  - It is difficult to handle large volumes of data.
- We can overcome this problem by storing data on secondary storage devices such as floppy or hard disks.
- The data is stored in these devices using the concept of Files and such data is often called persistent data.

- An I/O stream can represent different sources & destinations
  - e.g., disk files, devices, other programs, & memory arrays



Java I/O streams vs Stream

- **Java I/O** (Input and Output) is used to process the input and produce the output based on the input.
- Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- Java supports two streams-raw bytes and unicodes
- Java io, nio,nio.2 packages (Java 8)
- Java IO Streams vs Streams (Java 8)
- **Java 8 Streams API**, which provides a mechanism for processing a set of data in various ways that can include filtering, transformation, or any other way that may be useful to an application.(Collections)

Streams

- A *pipes-and-filters* based API for collections

```
This may be familiar...
ps -ef | grep java | cut -c 1-9 | sort -n | uniq
```

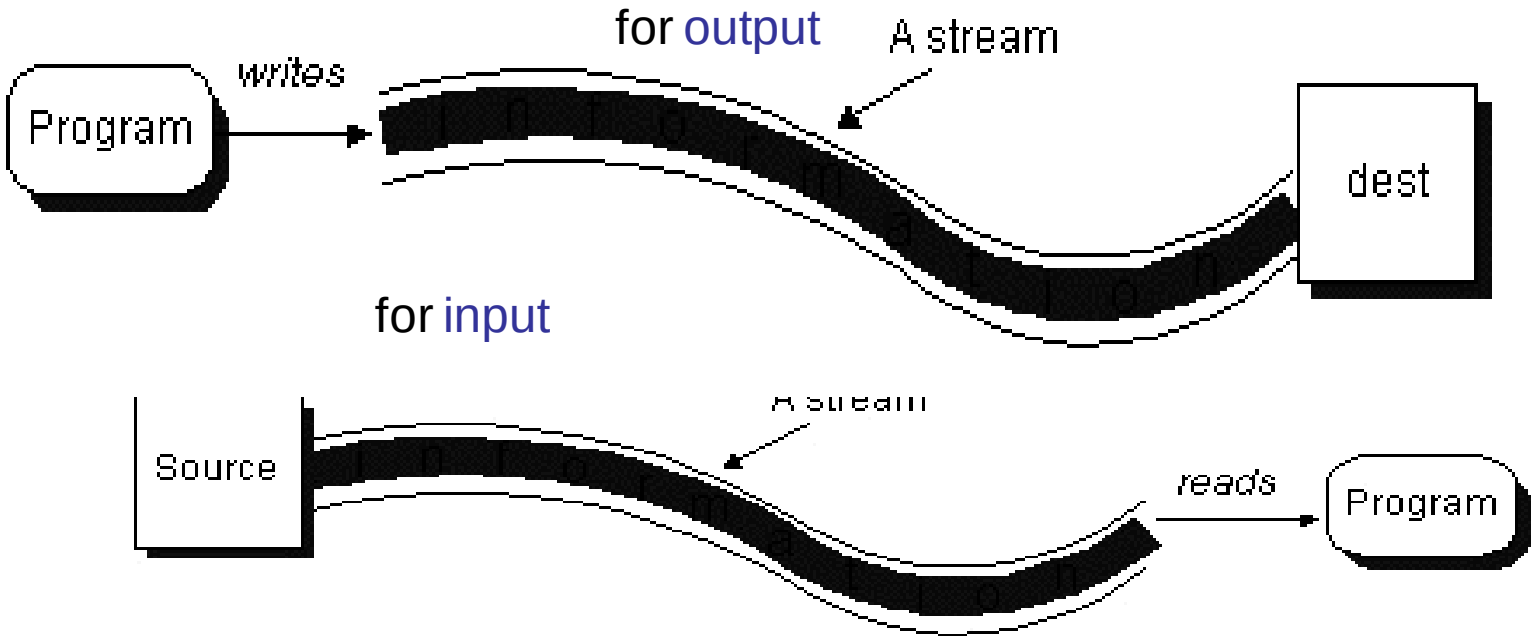
- A **Stream** is an abstraction that represents zero or more values (not objects)
- **Pipelines**
  - A stream source
  - Zero or more intermediate operations
  - a terminal operations
  - A pipeline can be executed in parallel
- `interface java.util.stream.Stream<T>`
  - `forEach()`
  - `filter()`
  - `map()`
  - `reduce()`
  - …
- `java.util.Collection<T>`
  - `Stream<T> stream()`
  - `Stream<T> parallelStream()`

Java Streams(EX)

```
List<Book> myBooks = …;

Stream<Book> books = myBooks.stream();

Stream<Book> goodBooks =
       books.filter(b -> b.getStarRating() > 3);

goodBooks.forEach(b -> System.out.println(b.toString()));
```

I/O Streams

- A stream is a sequence of bytes (or data or objects) that flow from a source to a destination
- In a program, we read information from an input stream and write information to an output stream



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 between the data source and destination

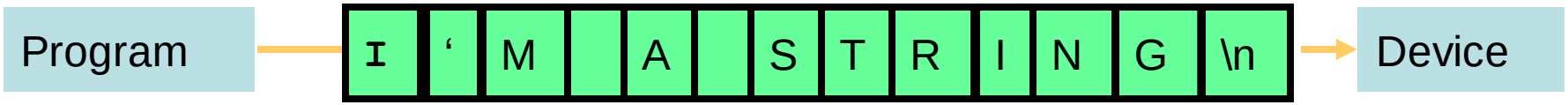
- ***Input stream***: a stream that provides input to a program
  - `System.in` is an input stream
- ***Output stream***: a stream that accepts output from a program
  - `System.out` is an output stream

- 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

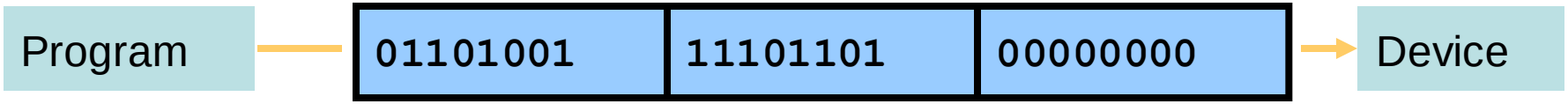
Streams

- JAVA distinguishes between 2 types of streams:

- Text – streams, containing ‘characters‘ (16 bit)

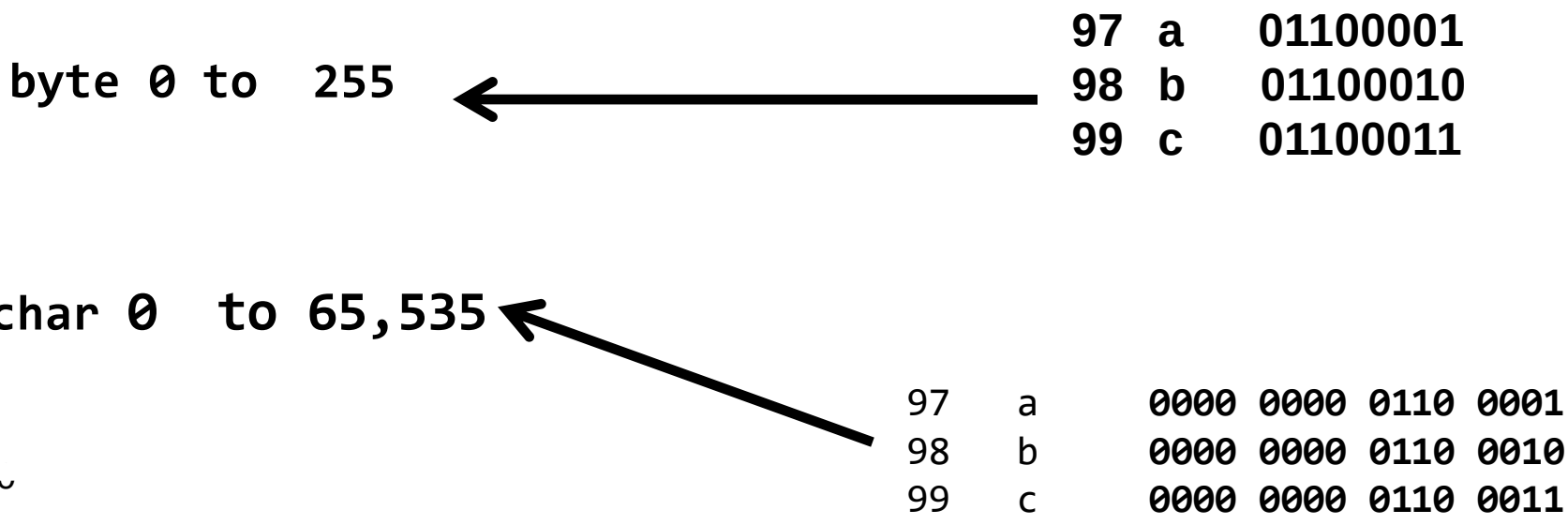


- Binary Streams, containing 8 – bit information



Streams

Byte Streams	Character streams
Operated on 8 bit (1 byte) data.	Operates on 16-bit (2 byte) unicode characters.
Input streams/Output streams	Readers/ Writers



Streams

- Byte stream-InputStream,OutputStream
- Character stream-Reader and Writer
- other primitive data(int, long, float,…) => DataInputStream /DataOutputStream
- Objects=> ObjectInputStream /ObjectOutputStream

InputStream

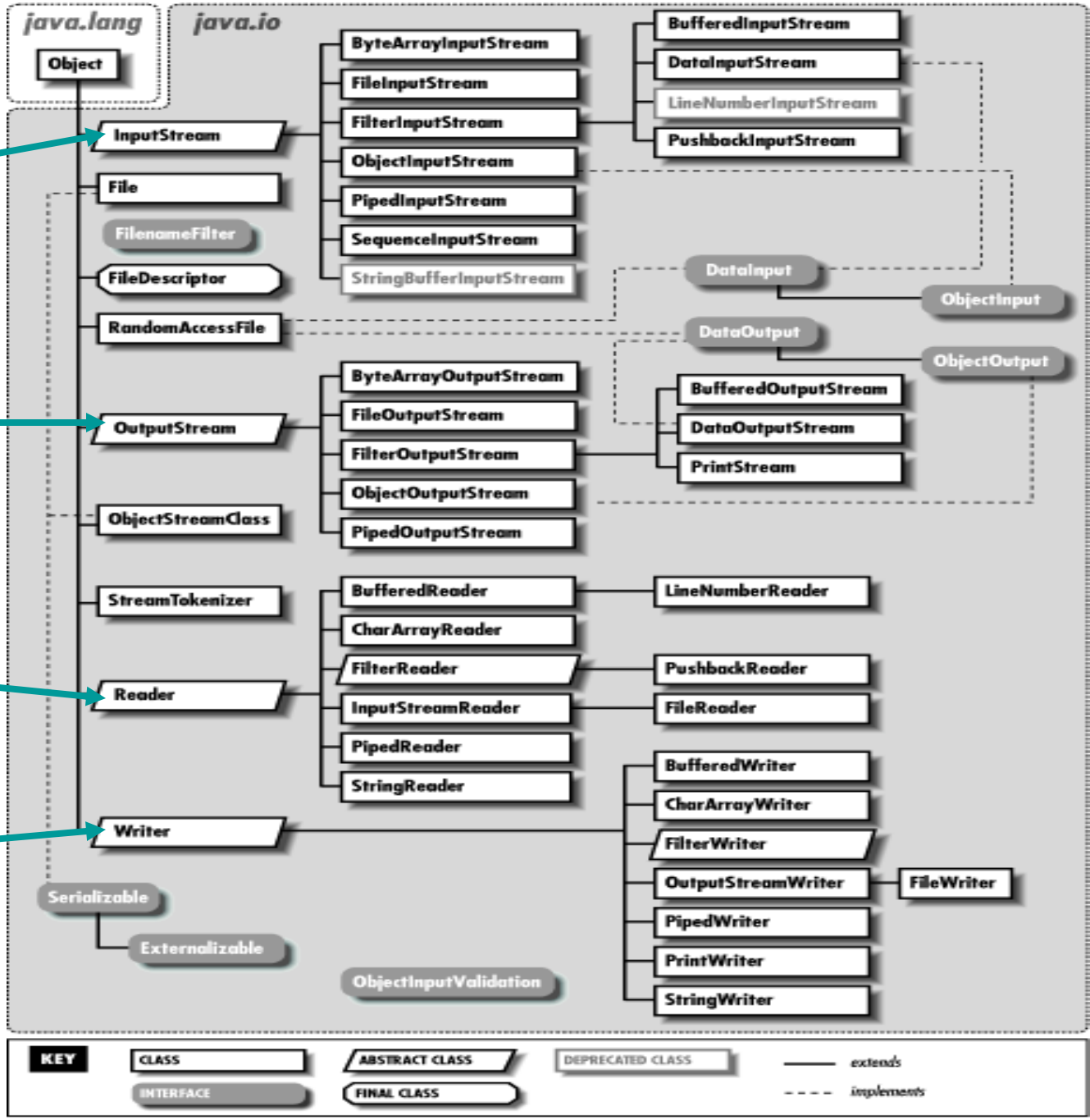
OutputStream

binary

Reader

Writer

text



I/O Streams		
Type of I/O	Streams	Description
Memory	CharArrayReader CharArrayWriter ByteArrayInputStream ByteArrayOutputStream	Use these streams to read from and write to memory. You create these streams on an existing array and then use the read and write methods to read from or write to the array.
	StringReader StringWriter StringBufferInputStream N/A for bytes	Use StringReader to read characters from a String in memory. Use StringWriter to write to a String. StringWriter collects the characters written to it in a StringBuffer, which can then be converted to a String. StringBufferInputStream is similar to StringReader, except that it reads bytes from a StringBuffer.
Pipe	PipedReader PipedWriter PipedInputStream PipedOutputStream	Implement the input and output components of a pipe. Pipes are used to channel the output from one thread into the input of another.

File	`FileReader`
	`FileWriter`
	
	`FileInputStream`
	`FileOutputStream`
		Collectively called file streams, these streams are used to read from or write to a file on the native file system.
		uses `FileReader` and `FileWriter` to copy the contents of one file into another.

Concatenation	*N/A char* `SequenceInputStream`	Concatenates multiple input streams into one input stream..

Object Serialization	*N/A text*
	
	`ObjectInputStream`
	`ObjectOutputStream`
		Used to serialize objects..

Data Conversion	`N/A` text
	
	`DataInputStream`
	`DataOutputStream`
		Read or write primitive data types in a machine-independent format..

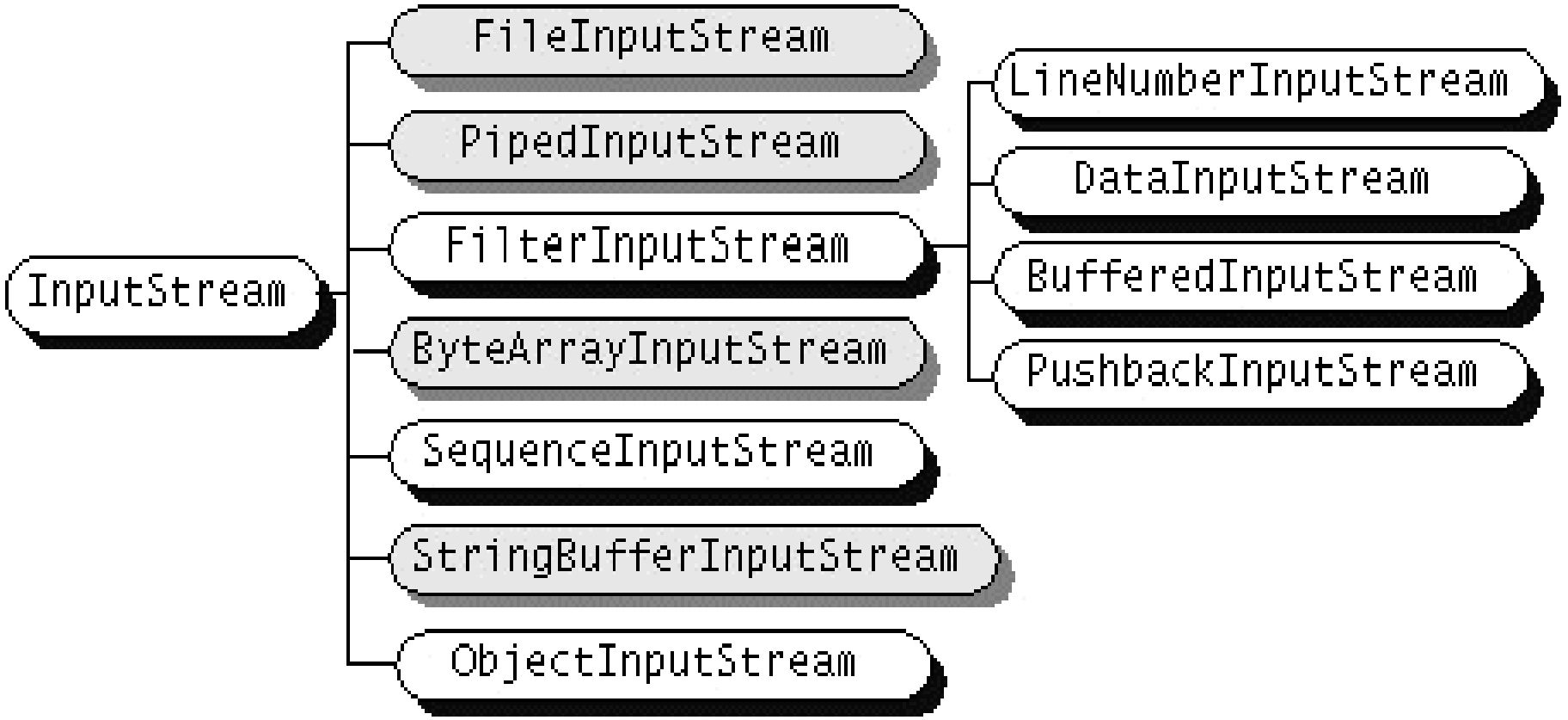
Counting	`LineNumberReader`
	
	`LineNumberInputStream`
		Keeps track of line numbers while reading.

Peeking Ahead	`PushbackReader`
	
	`PushbackInputStream`
		These input streams each have a pushback buffer. When reading data from a stream, it is sometimes useful to peek at the next few bytes or characters in the stream to decide what to do next.

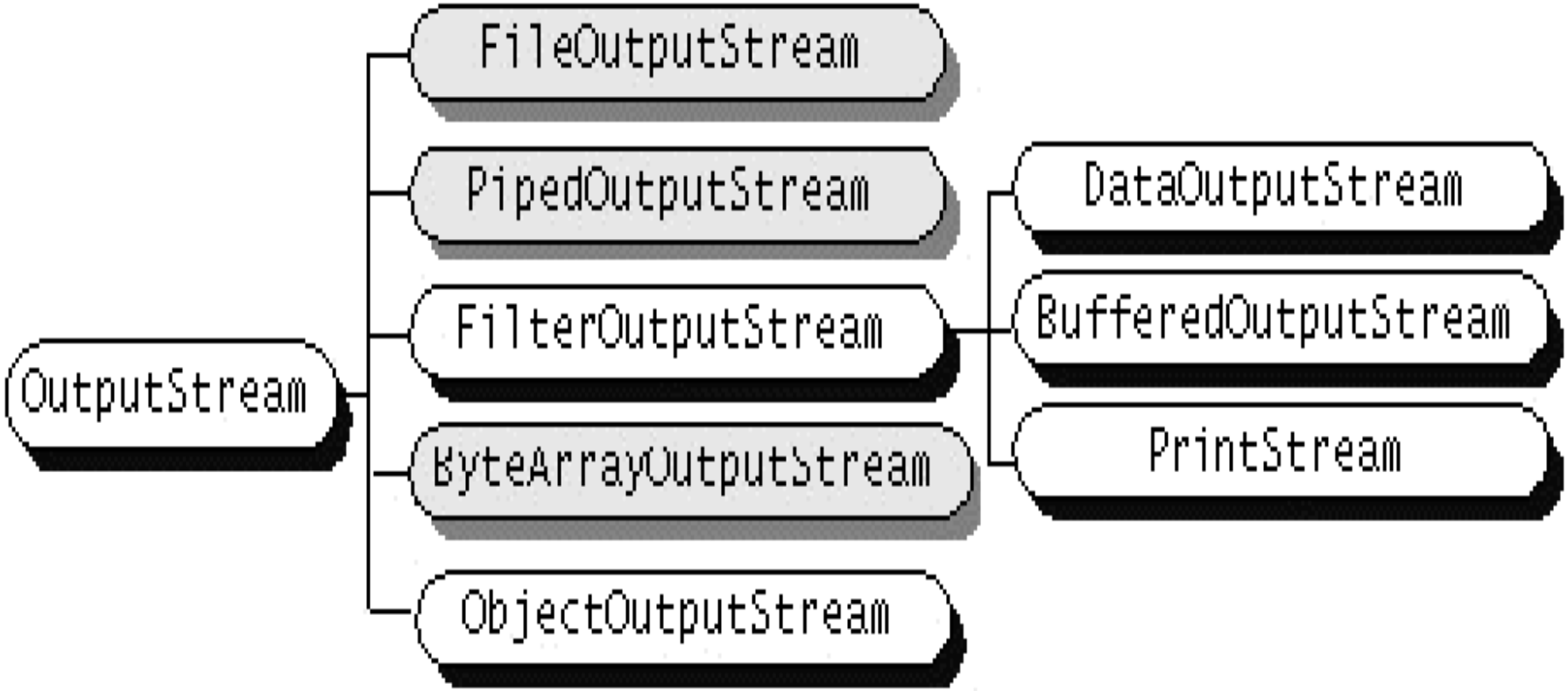
Printing	`PrintWriter`
	
	`PrintStream`
		Contain convenient printing methods. These are the easiest streams to write to, so you will often see other writable streams wrapped in one of these.

Buffering	`BufferedReader`
	`BufferedWriter`
	
	`BufferedInputStream`
	`BufferedOutputStream`
		Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source. Buffered streams are typically more efficient than similar nonbuffered streams and are often used with other streams.

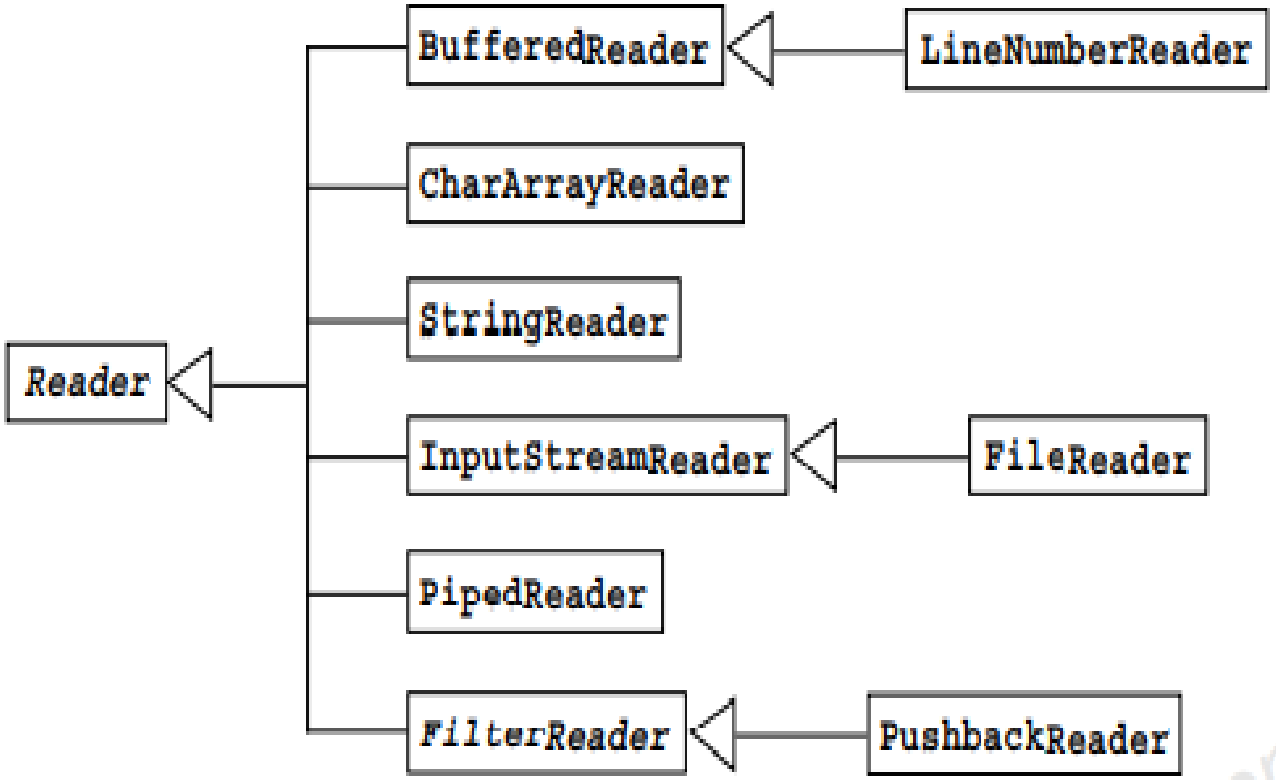
java.io.InputStream and its subclasses



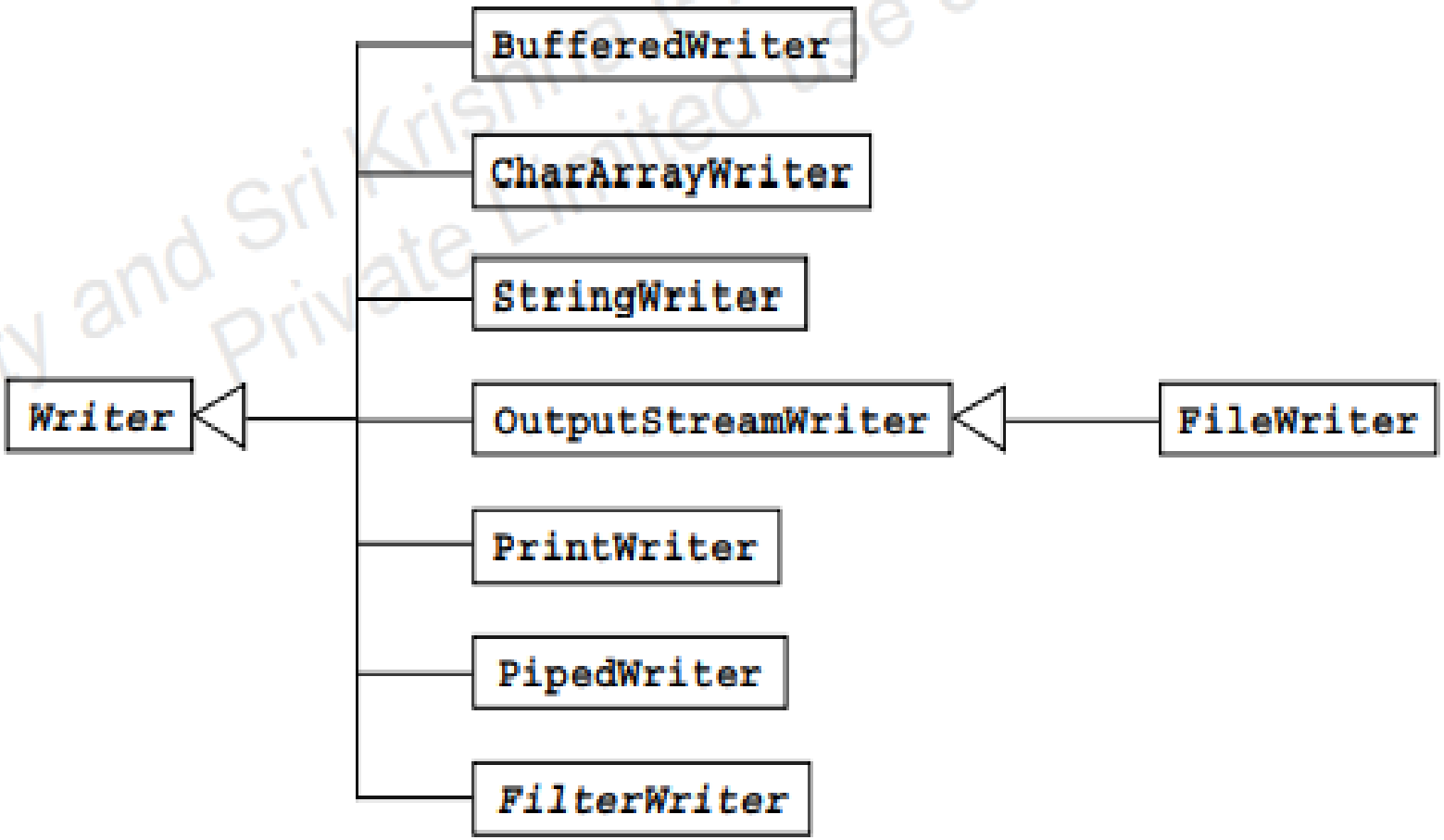
The Java.io.OutputStream and its subclasses



Reader Class



Writer Class



Reading data from keyboard

- There are many ways to read data from the keyboard. For example:
- InputStreamReader
- Console
- Scanner
- DataInputStream etc.

InputStreamReader class

InputStreamReader class can be used to read data from keyboard.It performs two tasks:

- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream

BufferedReader class

BufferedReader class can be used to read data line by line by readLine() method.

example

```
class G5{
public static void main(String args[])throws Exception{

InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);

System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Welcome "+name);
 }
}
```

I/O Stream Chaining

A program rarely uses a single stream object. Instead, it chains a series of streams together to process the data. Figure 10-1 demonstrates an example input stream; in this case, a file stream is *buffered* for efficiency and then converted into data (Java primitives) items.

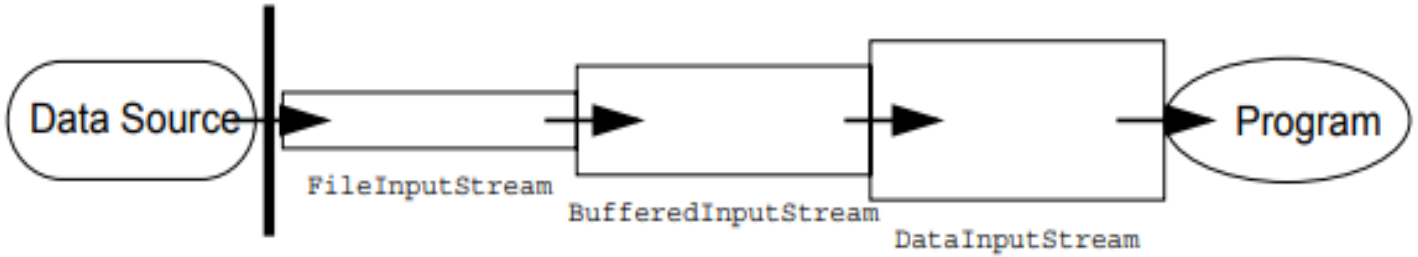
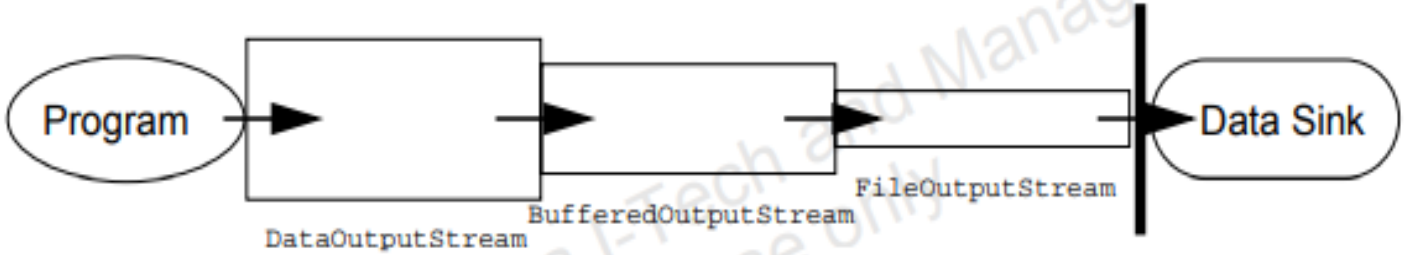
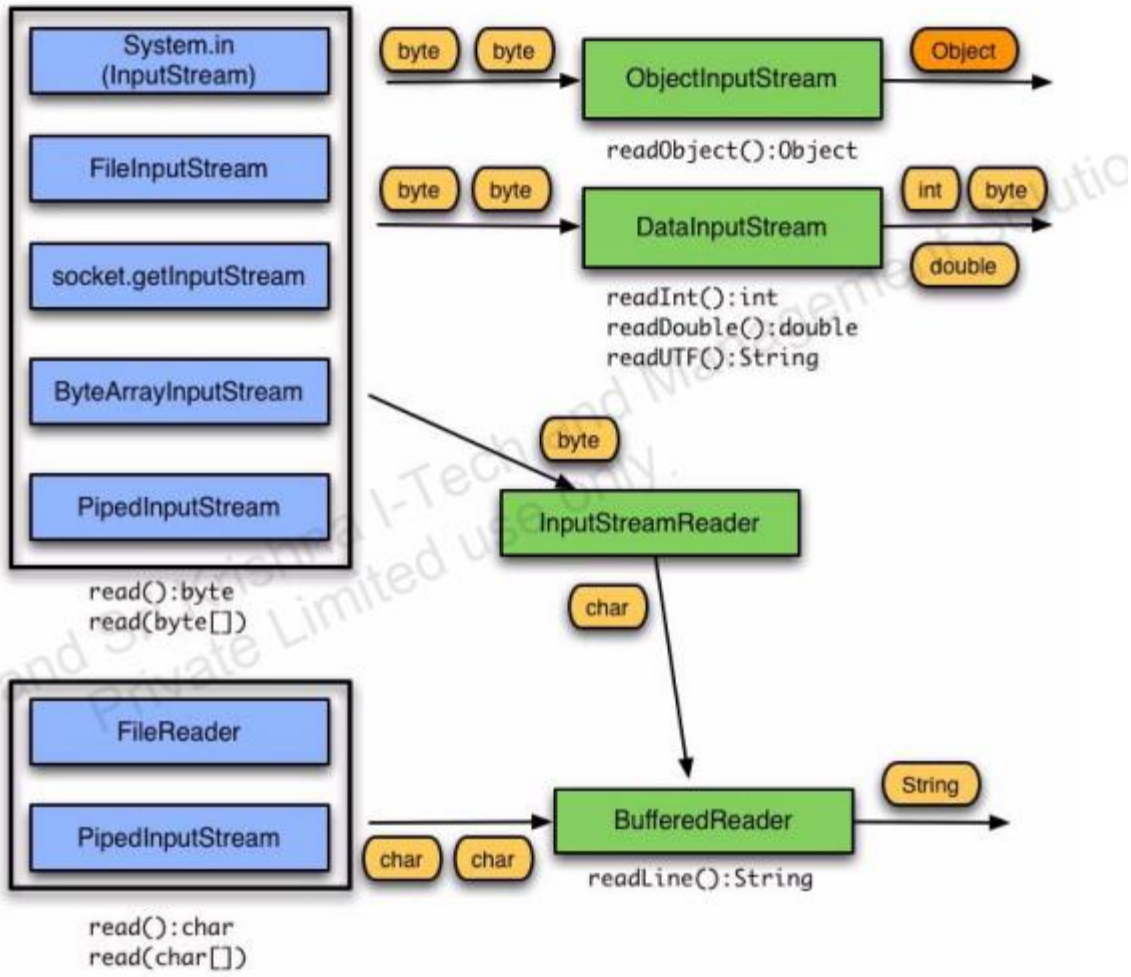


Figure 10-1 An Input Stream Chain Example

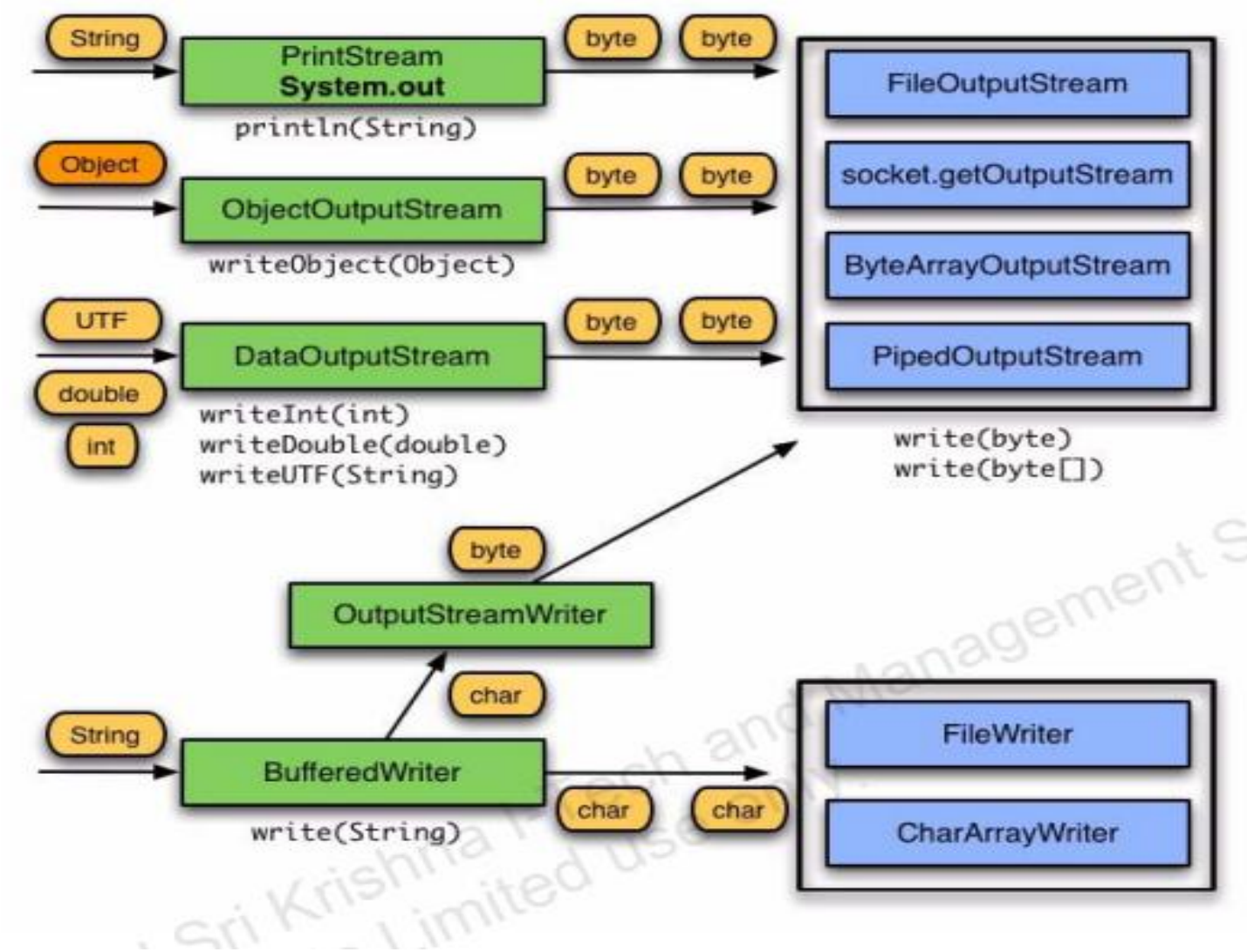
Figure 10-2 demonstrates an example output stream; in this case, data is written, then buffered, and finally written to a file.



Possible input stream and reader class that can be chained



Output Chaining-review



Java Console class

- The Java Console class is be used to get input from console. It provides methods to read text and password.
- If you read password using Console class, it will not be displayed to the user.

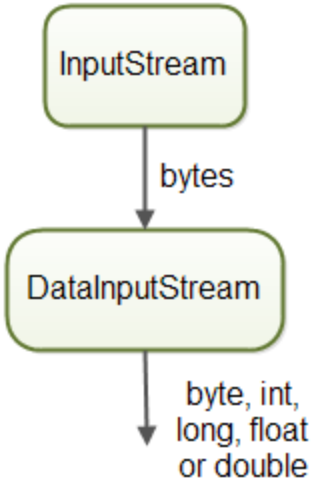
Example

```
import java.io.*;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);
}
}
```

```
System.out.print("Enter your password: ");

    char[] password = console.readPassword();
```

Data Input stream

- The *Java DataInputStream* class, java.io.DataInputStream, enables you to read Java primitives (int, float, long etc.) from an InputStream instead of only raw bytes.
- You wrap in InputStream in DataInputStream and then you can read Java primitives via ' the DataInputStream.
- That is why it is alled *DataInputStream* - because it reads data (numbers) instead of just bytes.



Example

```
import java.io.*;
public class DataInputStreamExample {
 public static void main(String[] args) throws IOException {
DataOutputStream dataOutputStream = new
    DataOutputStream( new
    FileOutputStream("data/data.bin"));
dataOutputStream.writeInt(123);
dataOutputStream.writeFloat(123.45F);
dataOutputStream.writeLong(789);
dataOutputStream.close();
```

```
DataInputStream dataInputStream = new DataInputStream( new
    FileInputStream("data/data.bin"));

 int int123 = dataInputStream.readInt();

 float float12345 = dataInputStream.readFloat();
long long789 = dataInputStream.readLong();
 dataInputStream.close();

 System.out.println("int123 = " + int123);
System.out.println("float12345 = " + float12345);
System.out.println("long789 = " + long789); } }
```

Java Scanner

- The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace bydefault. It provides many methods to read and parse various primitive values.

Example

```
class ScannerTest{
 public static void main(String args[]){
   Scanner sc=new Scanner(System.in);

   System.out.println("Enter your rollno");
   int rollno=sc.nextInt();
   System.out.println("Enter your name");
   String name=sc.next();
   System.out.println("Enter your fee");
   double fee=sc.nextDouble();
   System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
   sc.close();
 }
}
```

FILES

Java: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
- Java binary files are portable
  - they can be used by Java on different machines
  - Reading and writing binary files is normally done by a program
  - text files are used only to communicate with humans

Java Text Files

- Source files
- Occasionally input files
- Occasionally output files

Java Binary Files

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text file: an example

[unix: od –w8 –bc <file>]

[http://www.muquit.com/muquit/software/hod/hod.html for a Windows tool]

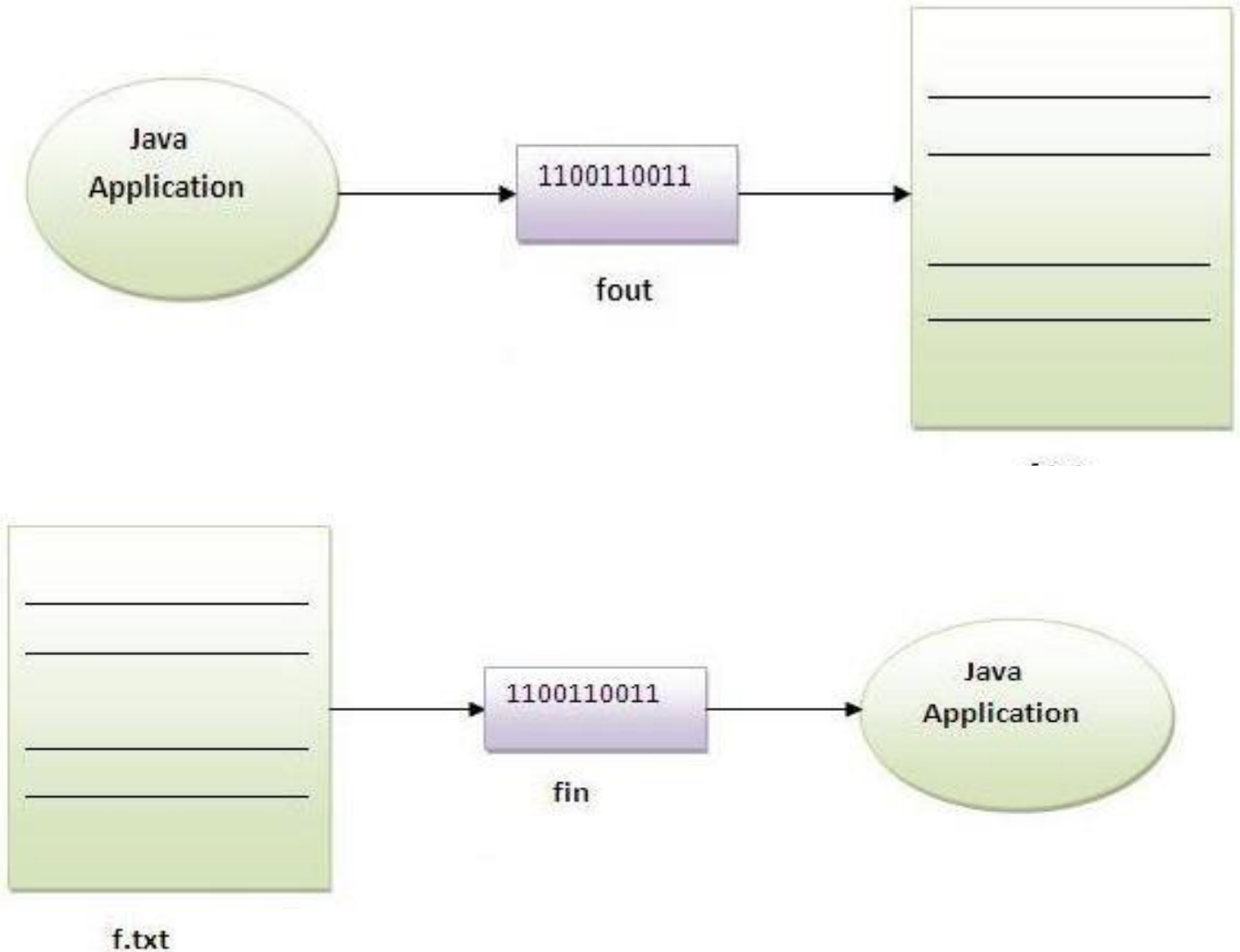
```
127     smiley
faces



0000000 061 062 067 011 163 155 151 154
          1   2   7  \t   s   m   i   l
0000010 145 171 012 146 141 143 145 163
          e   y  \n   f   a   c   e   s
0000020 012
         \n
```

Binary file: an example [a .class file]

```
0000000 312 376 272 276 000 000 000 061
        312 376 272 276  \0  \0  \0   1
0000010 000 164 012 000 051 000 062 007
         \0   t  \n  \0   )  \0   2  \a
0000020 000 063 007 000 064 010 000 065
         \0   3  \a  \0   4  \b  \0   5
0000030 012 000 003 000 066 012 000 002
         \n  \0 003  \0   6  \n  \0 002

...
0000630 000 145 000 146 001 000 027 152
         \0   e  \0   f 001  \0 027   j
0000640 141 166 141 057 154 141 156 147
          a   v   a   /   l   a   n   g
0000650 057 123 164 162 151 156 147 102
          /   S   t   r   i   n   g   B
0000660 165 151 154 144 145 162 014 000
          u   i   l   d   e   r  \f  \0
```

Binary file-File output stream

```
class Test{
  public static void main(String args[]){
   try{
     FileOutputstream fout=new FileOutputStream("abc.txt");
     String s="Sachin Tendulkar is my favourite player";
     byte b[]=s.getBytes();//converting string into byte array
     fout.write(b);
     fout.close();
     System.out.println("success...");
    }catch(Exception e){system.out.println(e);}
  }
}
```


Java
Application
1100110011
fout
f.txt
1100110011
fin
Java
Application

File Input Stream

```
import java.io.*;
class SimpleRead{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("abc.txt");
    int i=0;
    while((i=fin.read())!=-1){
     System.out.println((char)i);
    }
    fin.close();
  }catch(Exception e){system.out.println(e);}
 }
}
```

One file to another file

```
import java.io.*;
class C{
public static void main(String args[])throws Exception{
FileInputStream fin=new FileInputStream("C.java");
FileOutputStream fout=new FileOutputStream("M.java");
int i=0;
while((i=fin.read())!=-1){
fout.write((byte)i);
}
fin.close();
}
}
```

FileWriter and FileReader

```
class ReadTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   BufferedReader br = new BufferedReader(new FileReader(fl)) ;
   String str;
   while ((str=br.readLine())!=null)
   {
    System.out.println(str);
   }
   br.close();
   fl.close();
  }  catch (IOException   e)  { e.printStackTrace(); } }}
```

```
class WriteTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   String str="Write this string to my file";
   FileWriter fw = new FileWriter(fl) ;
   fw.write(str);
   fw.close();
   fl.close();
  }
  catch (IOException   e)  { e.printStackTrace(); } }}
```

```
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();      }}}}
```

Java NEW File I/O

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
public class WriteClass {
public static void write() throws IOException {
Path path = Paths.get("src/main/resources/question.txt");
String question = "To be or not to be?";
Files.write(path, question.getBytes());  }}
```

```
private static void copyFileUsingChannel(File source, File dest)
    throws IOException
{
FileChannel sourceChannel = null;
FileChannel destChannel = null;
try {
    sourceChannel = new FileInputStream(source).getChannel();
    destChannel = new FileOutputStream(dest).getChannel();
    destChannel.transferFrom(sourceChannel, 0,
    sourceChannel.size());
}finally
{
sourceChannel.close();
 destChannel.close(); } }
```

Files and Exceptions

- When creating files and performing I/O operations on them, the systems generates errors. The basic I/O related exception classes are given below:
  - EOFException – signals that end of the file is reached unexpectedly during input.
  - FileNotFoundException – file could not be opened
  - InterruptedIOException – I/O operations have been interrupted
  - IOException – signals that I/O exception of some sort has occurred – very general I/O exception.

Syntax

- Each I/O statement or a group of I/O statements much have an exception handler around it/them as follows:

```
try {
…// I/O statements – open file, read, etc.
}
catch(IOException e) // or specific type exception
{
    …//message output statements
}
```

File writer class

```
import java.io.*;
public class FileCopy {

    public static void main (String[] args)
    {
    try {
            FileReader srcFile = new FileReader("file1.txt");
              FileWriter destFile = new FileWriter("file2.txt");

               int ch;
               while((ch=srcFile.read()) != -1)
                     destFile.write(ch);
               srcFile.close();
               destFile.close();
          }
          catch(IOException e)
          {
                System.out.println(e);
                System.exit(-1);
          }
    }
}
```

Serialization

Saving an object to some type of permanent storage is called *persistence*. An object is said to be *persistent-capable* when you can store that object on a disk or tape or send it to another machine to be stored in memory or on disk. The non-persisted object exists only as long as the Java Virtual Machine is running.

Serialization is a mechanism for saving the objects as a sequence of bytes and later, when needed, rebuilding the byte sequence back into a copy of the object.

For objects of a specific class to be serializable, the class must implement the `java.io.Serializable` interface. The `Serializable` interface has no methods and only serves as a *marker* that indicates that the class that implements the interface can be considered for serialization.

When an object is serialized, only the fields of the object are preserved; methods and constructors are not part of the serialized stream. When a field is a reference to an object, the fields of that referenced object are also serialized if that object's class is serializable. The tree, or structure of an object's fields, including these sub-objects, constitutes the object *graph*.

Serializable Vs Transient

- The Serializable interface is a marker interface. Since it has no methods, you don't need to add additional code in your class that implements Serializable.
- In a Serializable class, every instance variable must be Serializable.
- Static variables are not serialized
- Non-Serializable instance variables must be declared transient to indicate that they should be ignored during the serialization process.
- By default, all primitive-type variables are serializable.
- For reference-type variables, you must check the class's documentation (and possibly its superclasses) to ensure that the type is Serializable.
- For example, Strings are Serializable.
- By default, arrays are serializable; however, in a reference-type array, the referenced objects might not be.
- Class Employee contains private data members all of which are Serializable.

Object Serialization

- The process of reading and writing objects is called object serialization
- ObjectInputStream and ObjectOutputStream are streams used to read/write objects.
- when to use object serialization:
  - Remote Method Invocation (RMI)--communication between objects via sockets
  - Lightweight persistence--storage of an object for use in a later invocation of the same program
- What a Java programmer need to know about object serialization:
  - How to serialize objects by writing them to an ObjectOutputStream and reading them in again using an ObjectInputStream.
  - how to write a class so that its instances can be serialized.

NotSerializableException

- **When a class implements the** Serializable **interface, all its sub-classes are serializable as well.**
- **But when an object has a reference to another object, these objects must implement the** Serializable **interface separately.**
- **If our class is having even a single reference to a non** Serializable **class then JVM will throw** NotSerializableException**.**

How to write objects

SYNTAX

```
ObjectOutputStream objectOut =

 objectOut.writeObject(serializableObject);
```

```
public class Employee implements Serializable
{
 private String empId;
    private String Name;
    private String Address;
    private String Dept;

   public Employee(String empId, String name, String address, String
     dept)

 {       this.empId = empId;
     this.Name = name;
  this.Address = address;
     this.Dept = dept;
    }

}
```

```
public class WriteObject
 {
 public static void main(String args[])
{

 try{

     Employee employee = new Employee("0001","Robert","Newyork City","IT");

    FileOutputStream fos = new FileOutputStream("employee.dat");

   ObjectOutputStream oos = new ObjectOutputStream(fos);

  oos.writeObject(employee);

           oos.close();

   }
catch(IOException ex){  ex.printStackTrace();  }
    }}
```

Deserialization

- Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization

```
import java.io.*;
class Depersist{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(
                                        new FileInputStream(" employee.dat "));
  Employee s=(Employee)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.empId+" "+s.Name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 } }
```

Random Access Files

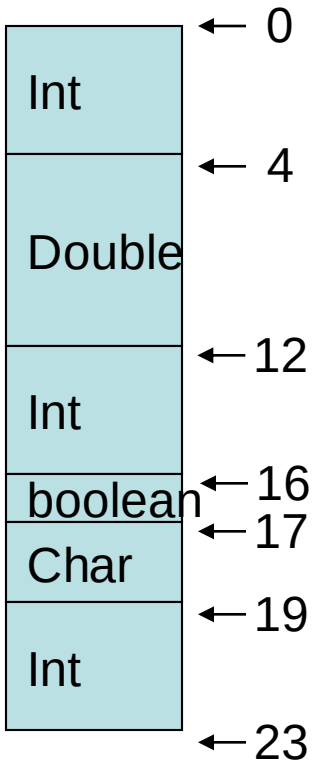
- So for we have discussed sequential files that are either used for storing data and accessed (read/write) them in sequence.
- Java IO package supports **RandomAccessFile** class that allow us to create files that can be used for reading and/or writing with random access.

- The file can be open either in read mode (“r”) or read-write mode (“rw”) as follows:
  - myFileHandleName = new RandomAccessFile (“filename”, “mode”);

- The file pointer can be set to any to any location (measured in bytes) using seek() method prior to reading or writing.

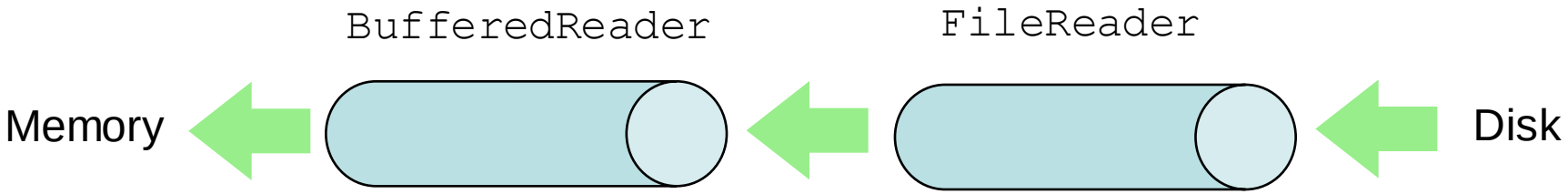
Random Access Example

```
import java.io.*;
public class RandomAccess  {
   public static void main(String args[]) throws IOException {
      // write primitive data in binary format to the "mydata"  file
      RandomAccessFile myfile = new RandomAccessFile("rand.dat", "rw");
      myfile.writeInt(120);
      myfile.writeDouble(375.50);
      myfile.writeInt('A'+1);
      myfile.writeBoolean(true);
      myfile.writeChar('X');

      myfile.seek(0);
       System.out.println(myfile.readInt());
       System.out.println(myfile.readDouble());

       myfile.seek(16);

       System.out.println(myfile.readBoolean());

      myfile.seek(myfile.length());
       myfile.writeInt(2003);

      myfile.seek(17);
       System.out.println(myfile.readChar());
       System.out.println(myfile.readInt());
       System.out.println("File length: "+myfile.length());
       myfile.close();
   }
}
```



Input File Streams

```
FileReader s = new FileReader(“file.txt");
BufferedReader br = new BufferedReader(s);
String t=“”;
t=br.readLine();
```



Methods for BufferedReader

- **readLine**: read a line into a String
- no methods to read numbers directly, so read numbers as Strings and then convert them (StringTokenizer later)
- read: read a char at a time
- close: close BufferedReader stream

BufferedReader - example

- Use a BufferedReader to read a file one line at a time and print the lines to standard output

```
import java.io.*;

class ReadTextFile {
        public static void main(String[] args)
                throws FileNotFoundException, IOException
        {
                BufferedReader in;
                in = new BufferedReader( new FileReader("Command.txt"));
                String line;
                while (( line = in.readLine()) != null )
            {
                System.out.println(line);
            }
        }
}
```