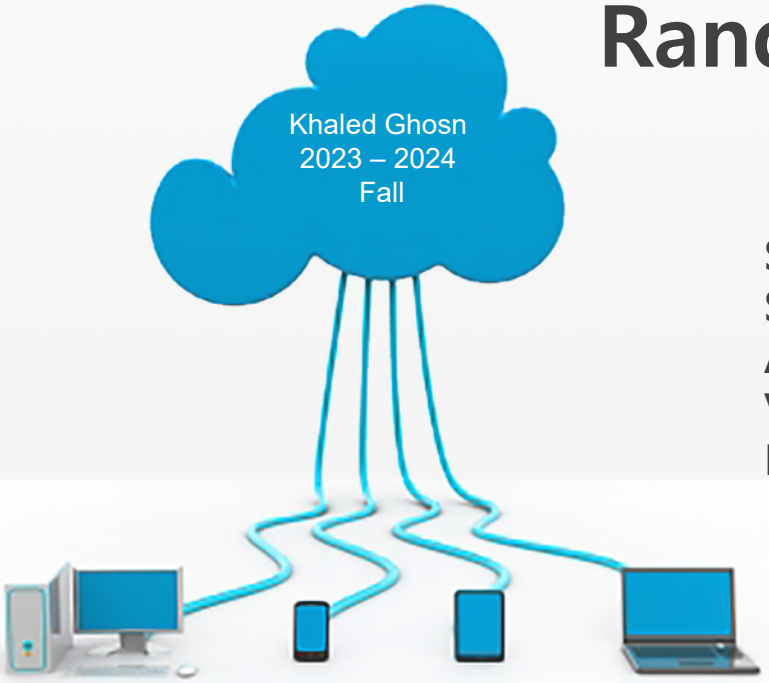
ARTS, SCIENCES & TECHNOLOGY
UNIVERSITY IN LEBANON
AUL

Binary Files, Streams, Random Access Files



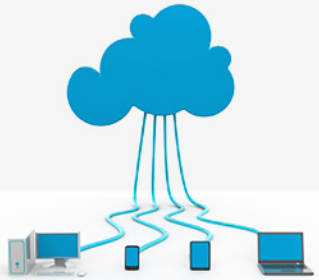
Khaled Ghosn
2023 – 2024
Fall

- Streams
- Serialization
- ArrayList
- Vector
- Random Access Files

Java Streams

Streams

- Byte-level communication is represented in Java by data streams, which are conduits through which information – **bytes of data** – is sent and received
- I/O in Java is built on **streams** (a stream is a sequence of data)
- In Java, streams are the sequence of data that are read from the source and written to the destination.
- Streams support many 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.

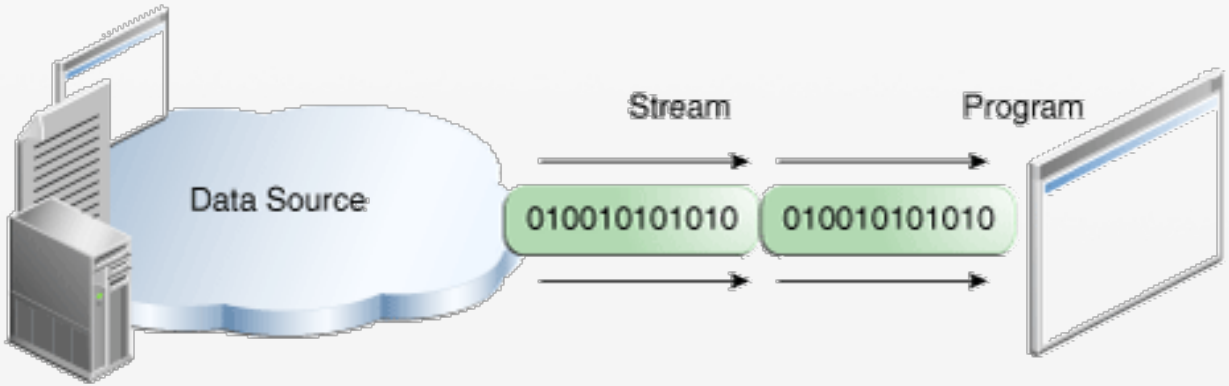


Java Streams

Streams

- A program uses an **input stream** to read data from a source, one item at a time

- **Input streams** move data into a Java program usually from an external source

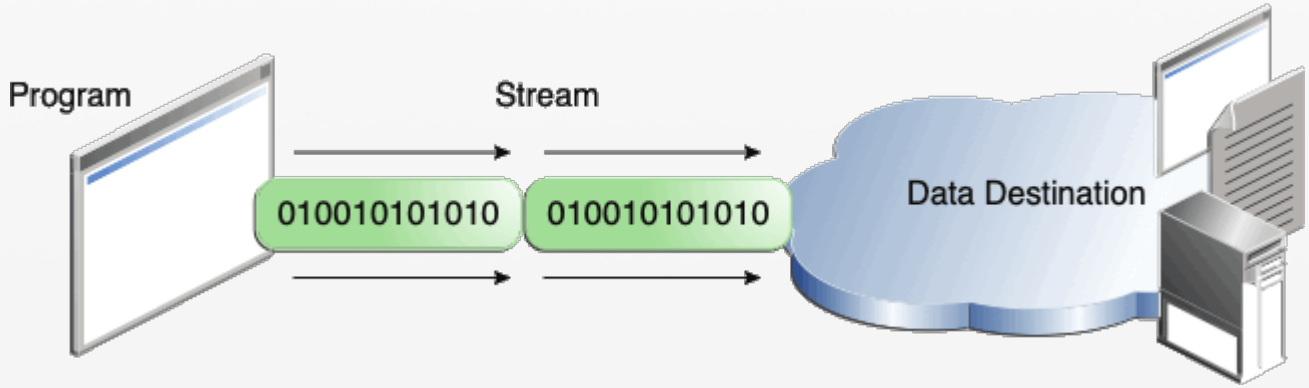


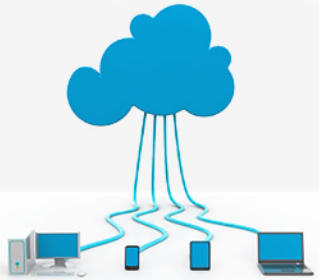
Java Streams

Streams

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

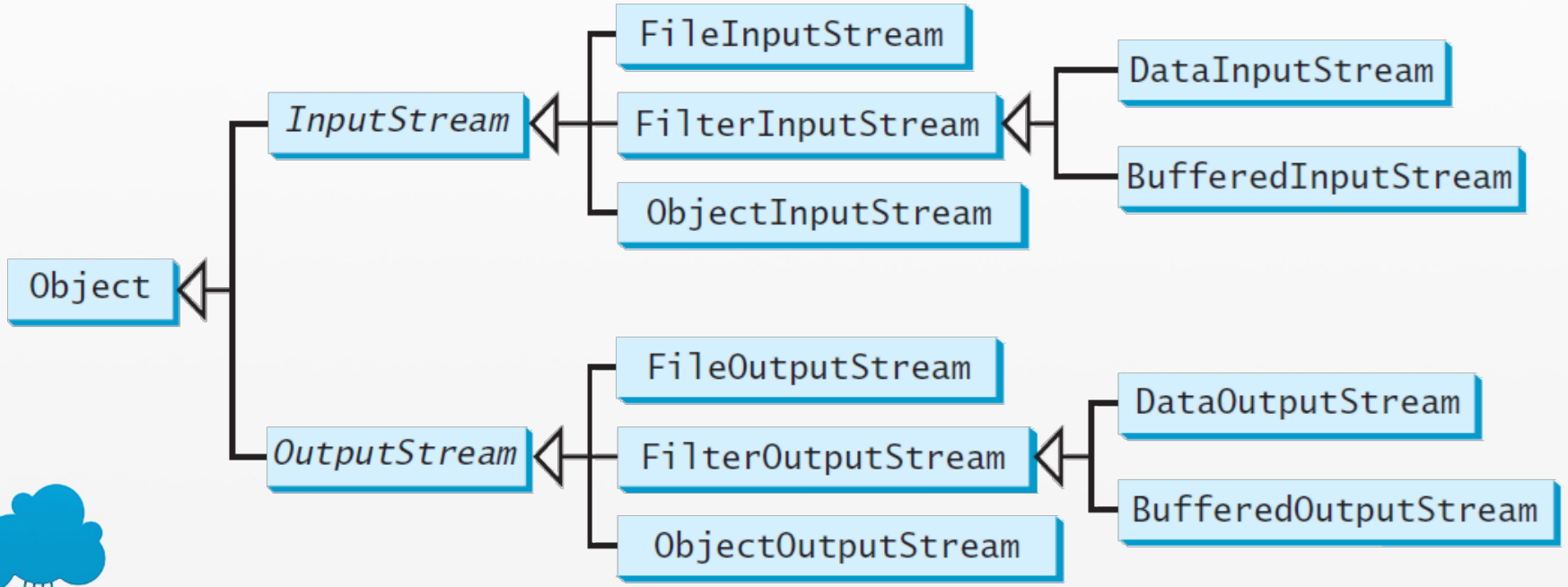
- **Output streams** move data from a Java program to an external targ et (File, memory, network)

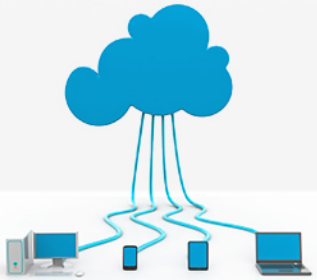




Java Streams

Streams



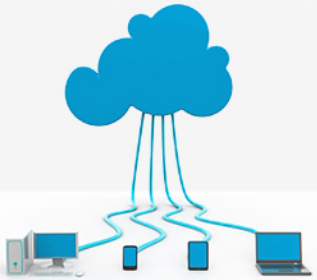


InputStream, OutputStream, and their subclasses are for performing binary I/O.

Java Streams

InputStream / OutputStream

- The abstract class **InputStream** is the root class for reading binary data.
- The abstract class **OutputStream** is the root class for writing binary data.
- All the methods in the binary I/O classes are declared to throw java.io.IOException or a subclass of java.io.IOException.



Java Streams

InputStream

java.io.InputStream

```
+read(): int

+read(b: byte[]): int

+read(b: byte[], off: int,
 len: int): int

+available(): int
+close(): void
+skip(n: long): long

+markSupported(): boolean
+mark(readlimit: int): void
+reset(): void
```

Reads the next byte of data from the input stream. The value byte is returned as an `int` value in the range `0` to `255`. If no byte is available because the end of the stream has been reached, the value –1 is returned.

Reads up to `b.length` bytes into array `b` from the input stream and returns the actual number of bytes read. Returns –1 at the end of the stream.

Reads bytes from the input stream and stores them in `b[off]`, `b[off+1]`, . . ., `b[off+len-1]`. The actual number of bytes read is returned. Returns –1 at the end of the stream.

Returns an estimate of the number of bytes that can be read from the input stream.

Closes this input stream and releases any system resources occupied by it.

Skips over and discards `n` bytes of data from this input stream. The actual number of bytes skipped is returned.

Tests whether this input stream supports the `mark` and `reset` methods.

Marks the current position in this input stream.

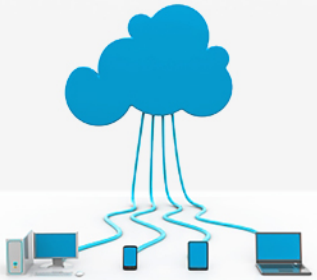
Repositions this stream to the position at the time the `mark` method was last called on this input stream.

The abstract **InputStream** class defines the methods for the input stream of bytes.

Java Streams

OutputStream

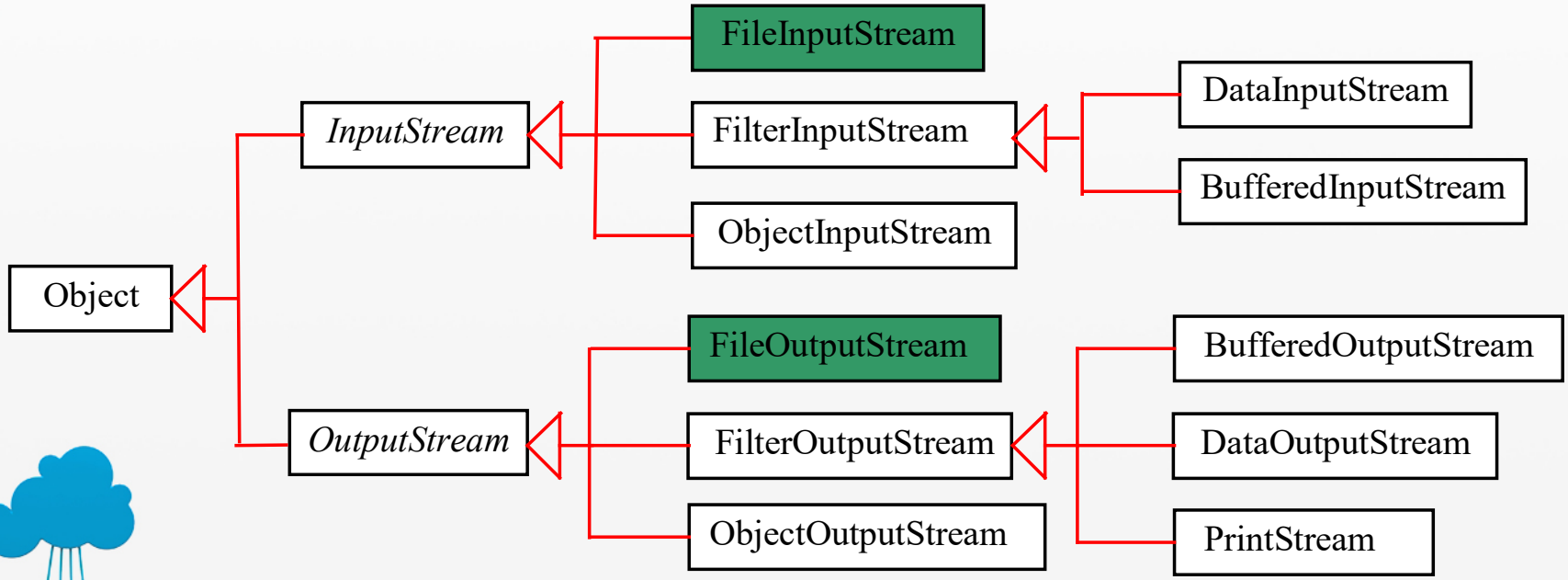
java.io.OutputStream	
+write(int b): void	Writes the specified byte to this output stream. The parameter `b` is an `int` value. `(byte)b` is written to the output stream.
`+write(b: byte[]): void`	Writes all the bytes in array `b` to the output stream.
`+write(b: byte[], off: int, len: int): void`	Writes `b[off]`, `b[off+1]`, . . ., `b[off+len-1]` into the output stream.
`+close(): void`	Closes this output stream and releases any system resources occupied by it.
`+flush(): void`	Flushes this output stream and forces any buffered output bytes to be written out.

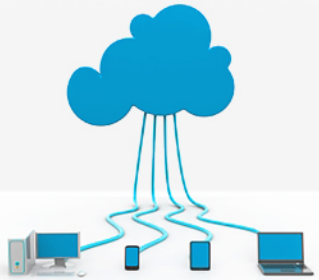


The abstract **OutputStream** class defines the methods for the output stream of bytes.

Java Streams

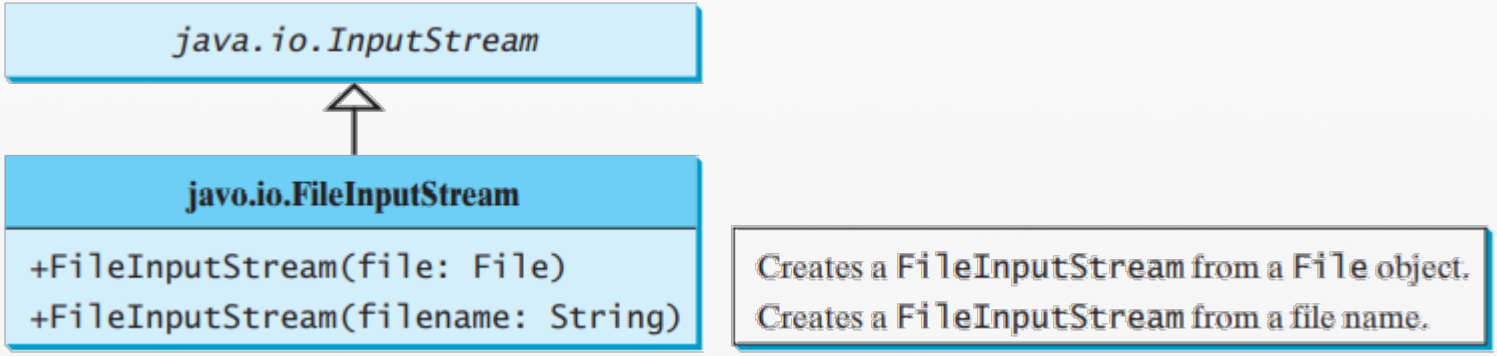
FileInputStream / FileOutputStream

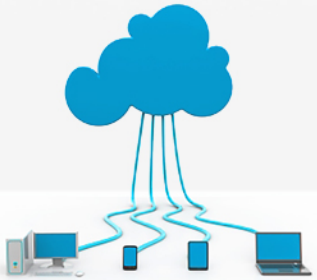




Java Streams

FileInputStream

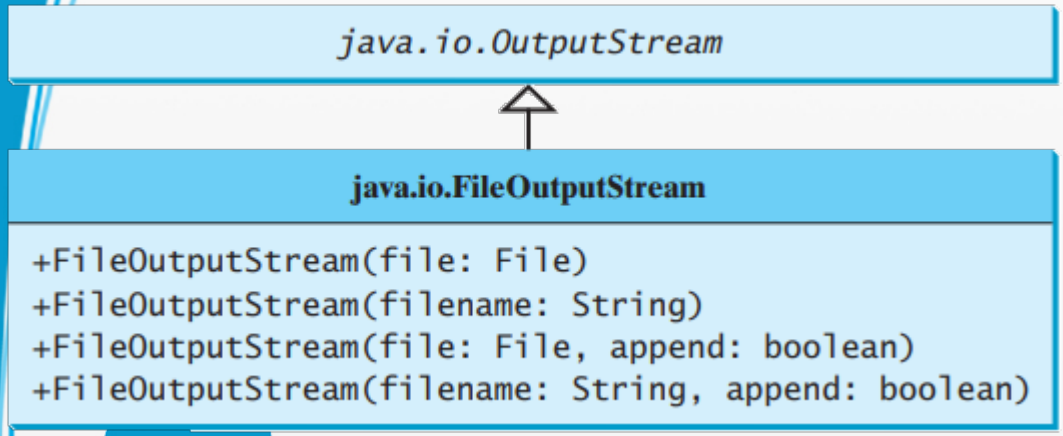




FileInputStream inputs a stream of bytes from a file.

Java Streams

FileOutputStream



Creates a `FileOutputStream` from a `File` object.
Creates a `FileOutputStream` from a file name.
If `append` is true, data are appended to the existing file.
If `append` is true, data are appended to the existing file.

FileOutputStream outputs a stream of bytes to a file.

Java Streams

FileInputStream / FileOutputStream

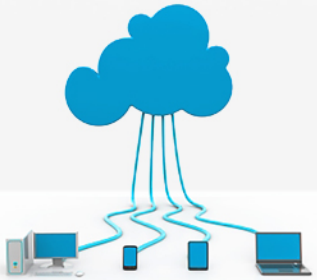
- FileInputStream / FileOutputStream is for reading / writing bytes from / to files.
  - associates a binary input / output stream with an external file.

- All the methods in these classes are inherited from InputStream and OutputStream.

- To construct a FileInputStream, use the following constructors

  public FileInputStream(String filename)
  public FileInputStream(File file)

- A java.io.FileNotFoundException would occur if you attempt to create a FileInputStream with a nonexistent file.



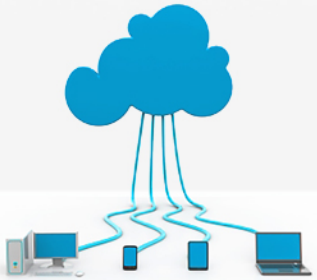
Java Streams

FileInputStream / FileOutputStream

- To construct a FileOutputStream, use the following constructors

```
public FileOutputStream(String filename)
public FileOutputStream(File file)
public FileOutputStream(String filename, boolean append)
public FileOutputStream(File file, boolean append)
```

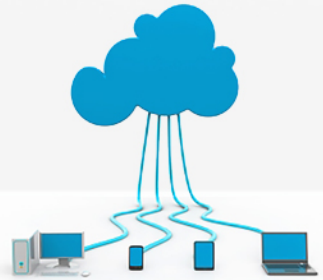
- If the file does not exist, a new file would be created
- If the file already exists, the first two constructors would delete the current contents in the file
- To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter



Java Streams

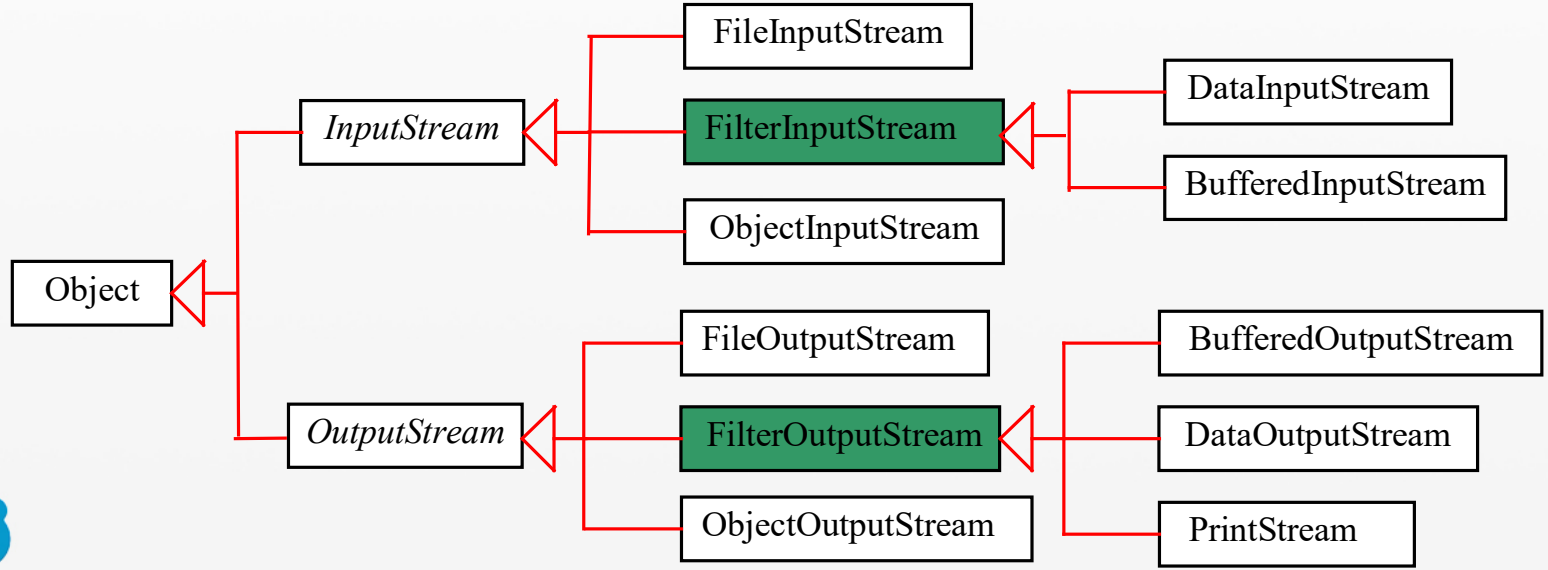
FileInputStream / FileOutputStream

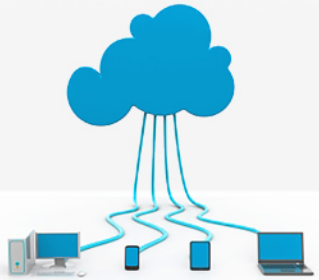
- An instance of FileInputStream can be used as an argument to c onstruct a Scanner
- An instance of FileOutputStream can be used as an argument to construct a PrintWriter.
- You can create a PrintWriter to append text into a file using
- e.g. PrintWriter p = new PrintWriter (new FileOutputStream ("temp.txt", true));
  - If temp.txt does not exist, it is created. If temp.txt already exists, new data are appended to the file.



Java Streams

FilterInputStream / FilterOutputStream

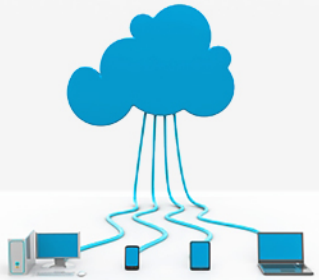




Java Streams

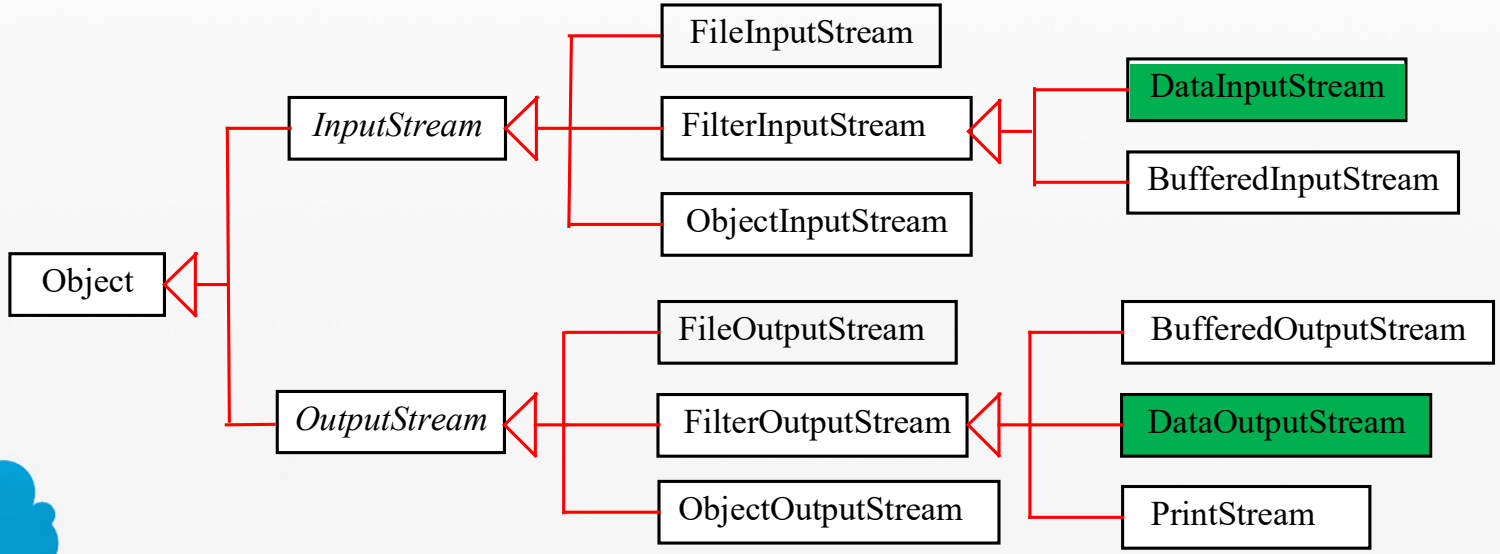
FilterInputStream / FilterOutputStream

- Filter streams are streams that filter bytes
- The basic byte input stream provides a read method that can only be used for reading bytes
- FilterInputStream and FilterOutputStream are the base classes for filtering data.



Java Streams

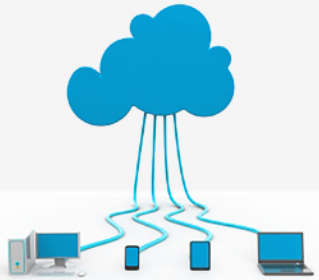
DataInputStream / DataOutputStream



Java Streams

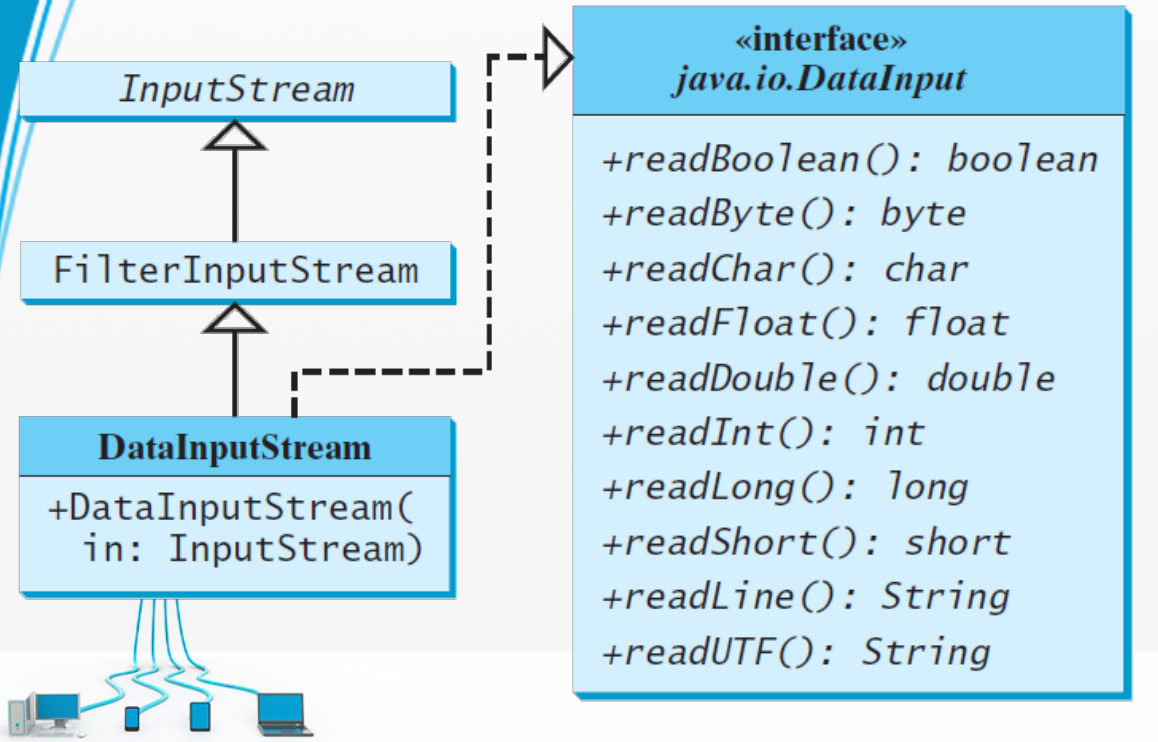
DataInputStream

- DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.
- DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.
- DataInputStream extends FilterInputStream and implements the DataInput interface
- DataOutputStream extends FilterOutputStream and implements the DataOutput interface



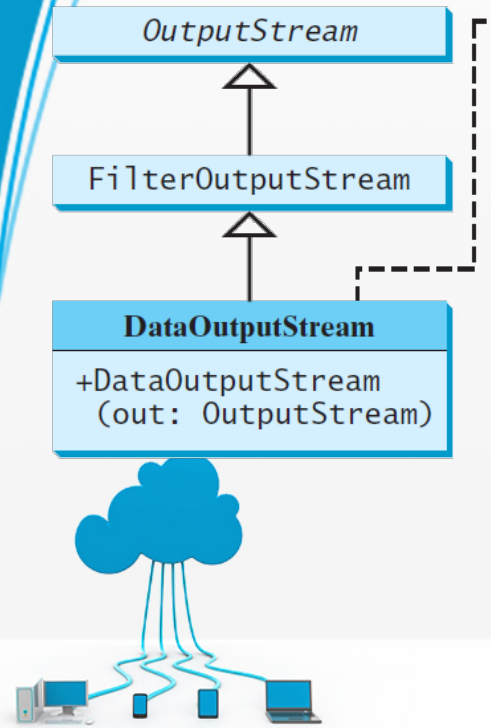
Java Streams

DataInputStream



Java Streams

DataOutputStream



«interface»
java.io.DataOutput

+writeBoolean(b: boolean): void	Writes a Boolean to the output stream.
+writeByte(v: int): void	Writes the eight low-order bits of the argument `v` to the output stream.
+writeBytes(s: String): void	Writes the lower byte of the characters in a string to the output stream.
+writeChar(c: char): void	Writes a character (composed of 2 bytes) to the output stream.
+writeChars(s: String): void	Writes every character in the string `s` to the output stream, in order, 2 bytes per character.
+writeFloat(v: float): void	Writes a `float` value to the output stream.
+writeDouble(v: double): void	Writes a `double` value to the output stream.
+writeInt(v: int): void	Writes an `int` value to the output stream.
+writeLong(v: long): void	Writes a `long` value to the output stream.
+writeShort(v: short): void	Writes a `short` value to the output stream.
+writeUTF(s: String): void	Writes `s` string in UTF format.

Java Streams

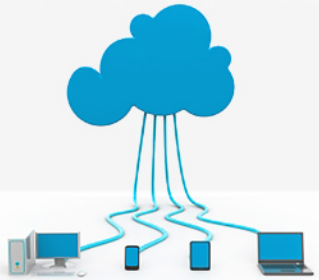
DataInputStream / DataOutputStream

- Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

```
public DataInputStream (InputStream instream)
public DataOutputStream (OutputStream outstream)
```

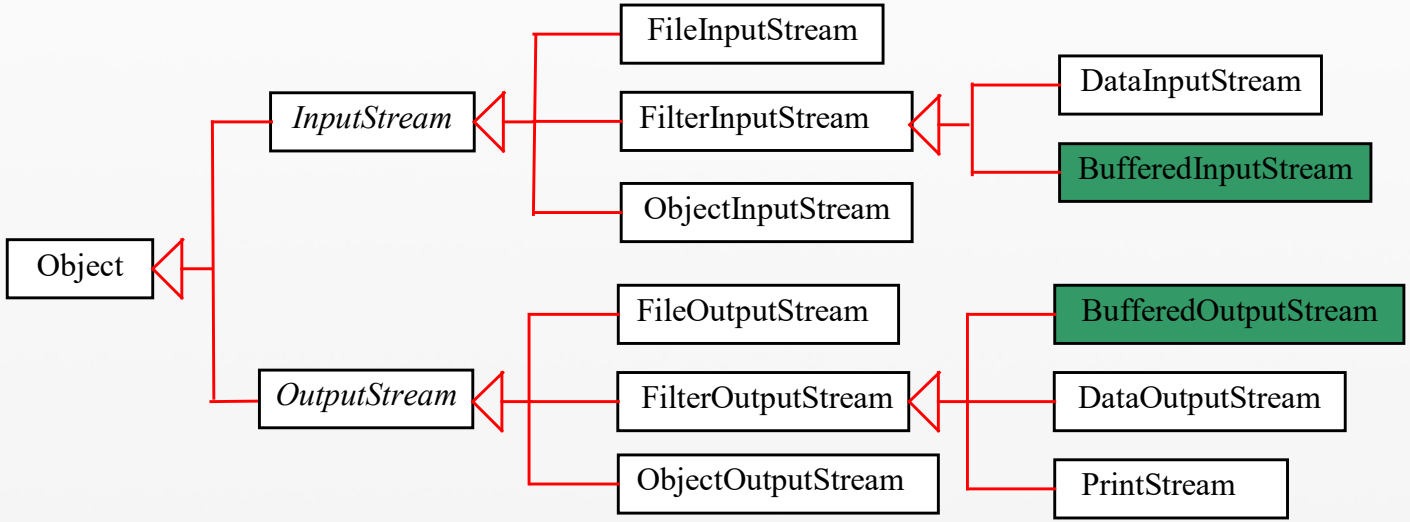
- The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**

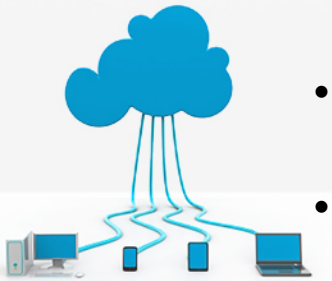
```
DataInputStream infile =
            new DataInputStream (new FileInputStream("in.dat"));

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



Java Streams

BufferedInputStream / BufferedOutputStream





- BufferedInputStream / BufferedOutputStream does not contain new methods.
- All the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes.

Java Streams

BufferedInputStream / BufferedOutputStream

// Create a BufferedInputStream

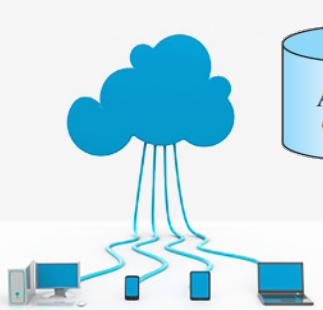
public BufferedInputStream(InputStream in)

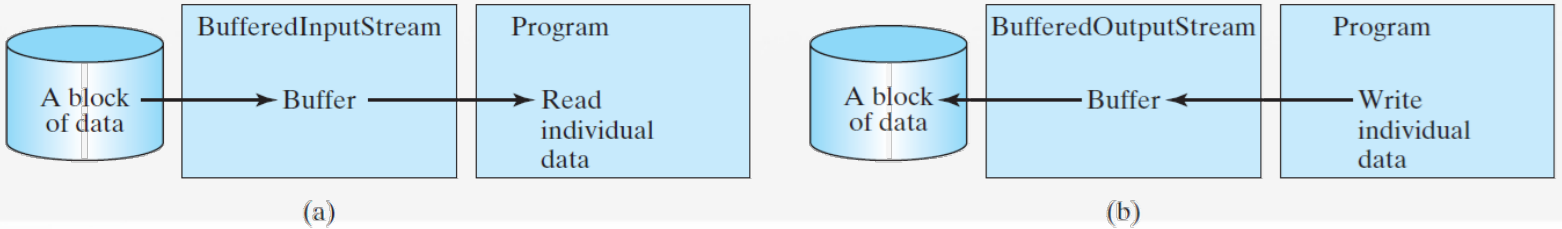
public BufferedInputStream(InputStream in, int bufferSize)

// Create a BufferedOutputStream

public BufferedOutputStream(OutputStream out)

public BufferedOutputStream(OutputStream out, int bufferSize)



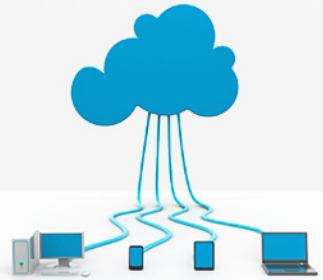


Java Streams

Object I/O

ObjectInputStream / ObjectOutputStream

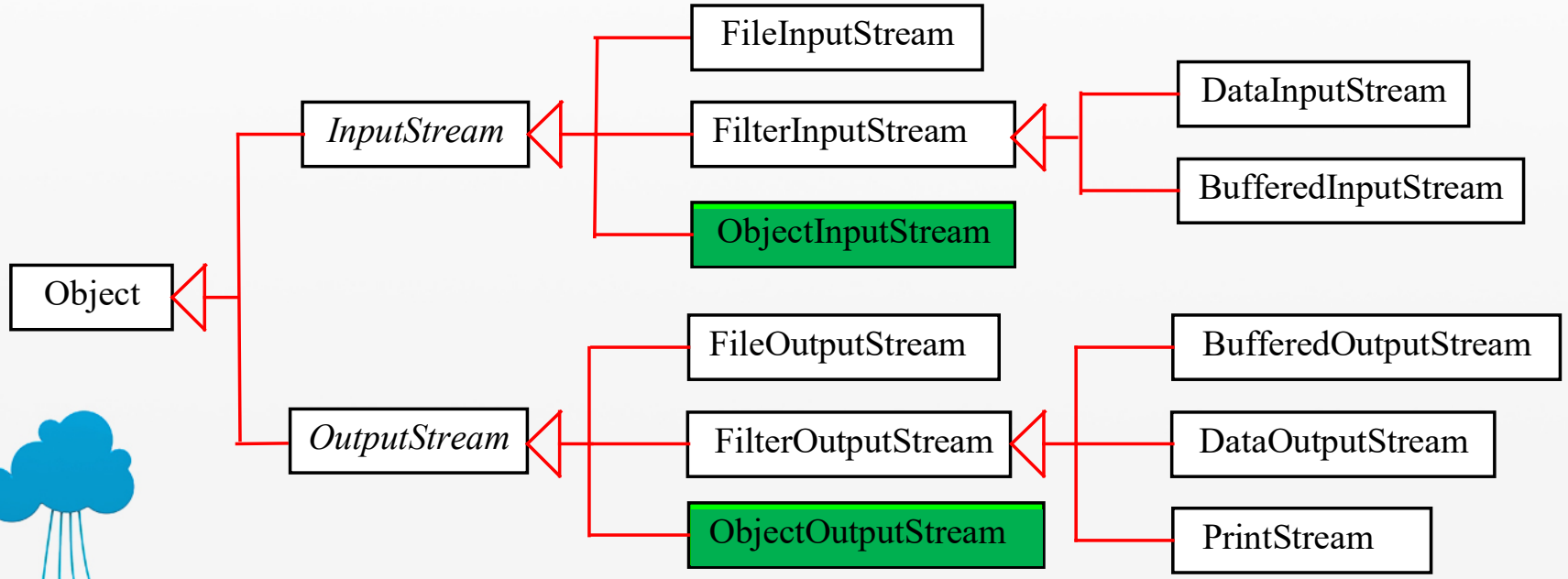
- enables you to perform I/O for primitive type values and strings.
- enables you to perform I/O for objects in addition for primitive type values and strings.
- can be used to read / write serializable objects.

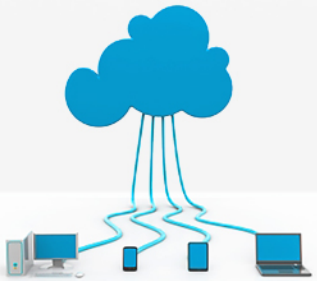


Java Streams

Object I/O

ObjectInputStream / ObjectOutputStream

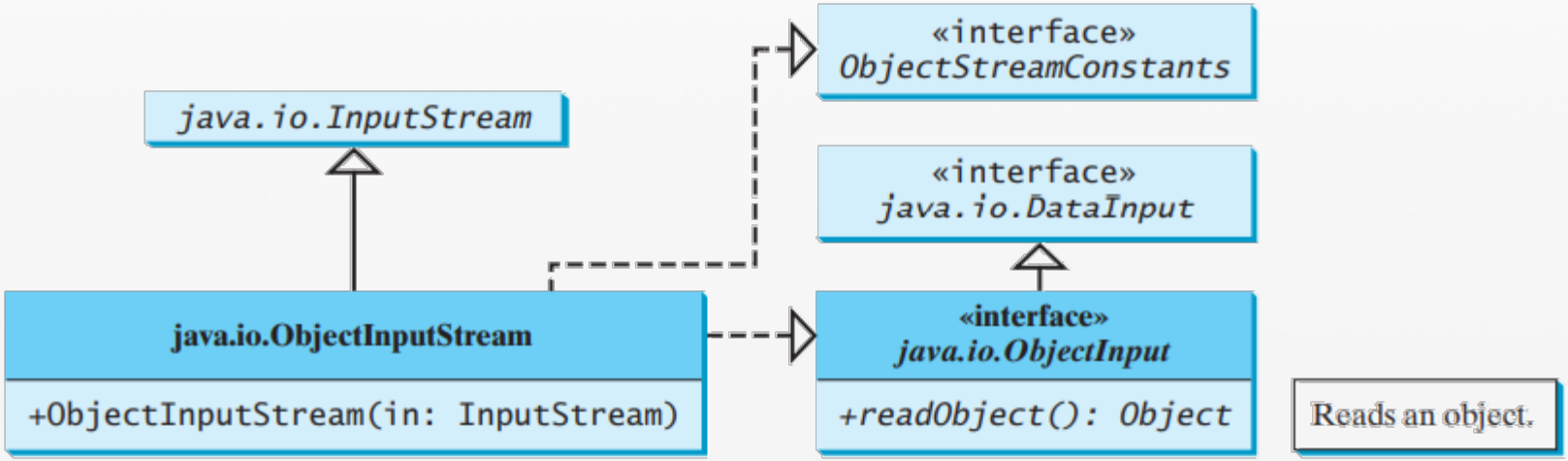


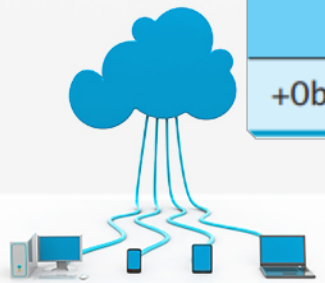


Java Streams

ObjectInputStream

ObjectInputStream extends InputStream and implements ObjectInput and ObjectStreamConstants.

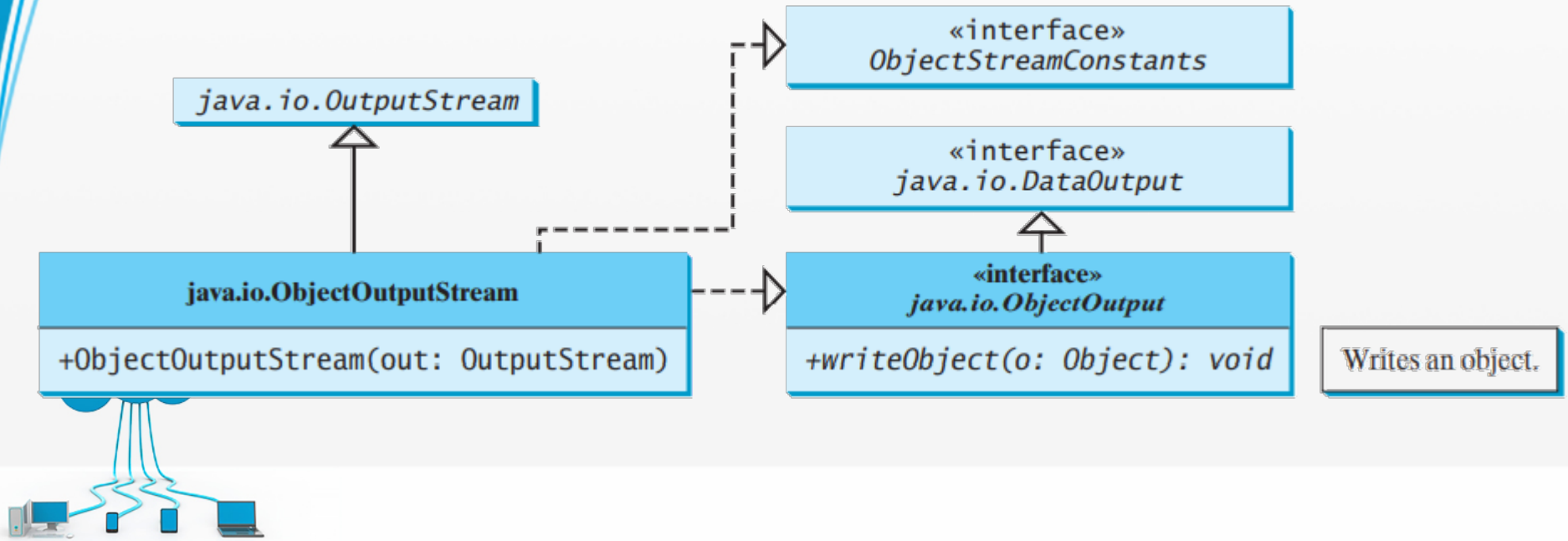




Java Streams

ObjectOutputStream

ObjectOutputStream extends OutputStream and implements ObjectOutput and ObjectStreamConstants.

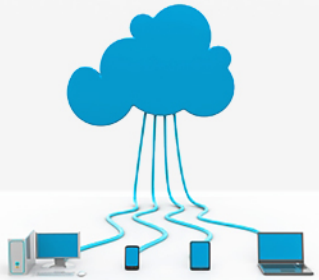


Java Streams

Using Object Streams

You may wrap an ObjectInputStream/ObjectOutputStream on any InputStream/OutputStream using the following constructors:

```
// Create an ObjectInputStream
ObjectInputStream ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```

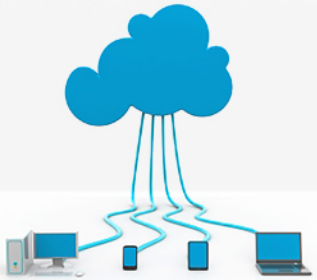


Serialization

To represent an object in a byte-encoded format that can be stored and passed to a stream, and in need can be reconstructed

Objects of any class that implements the Serializable interface may be transferred to and from disc files as whole objects, with no need for decomposition of those objects

The Serializable interface is nothing more than a marker to tell Java that objects of this class may be transferred on an object stream to and from files



Serialization

Serialization can be used in:

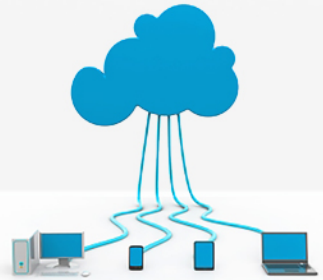
- Remote Method Invocation (RMI), communication between objects via sockets
- Archival of an object for use in a later invocation of the same program

Objects to be serialized:

- Must implement *Serializable* interface
- Non-persistent fields can be marked with *transient* keyword

The following is written and read during serialization

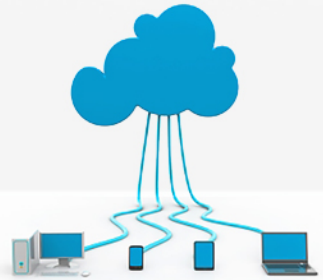
- Class of the object
- Class signature
- Values of all non-transient and non-static members



Serialization

ObjectOutputStream & ObjectInputStream

- ✓ Works like other input-output streams
- ✓ They can write and read Objects.
- ✓ *ObjectOutputStream*: Serializes Java Objects into a byte-encoded format, and writes them onto an OutputStream
- ✓ *ObjectInputStream*: Reads and reconstructs Java Objects from a byte-encoded format read from InputStream



Serialization

ObjectOutputStream & ObjectInputStream

- *ObjectOutputStream* :is used to save entire objects directly to disc

```
ObjectOutputStream outStream = new ObjectOutputStream (
                                    new FileOutputStream ("personnel.dat"));
```

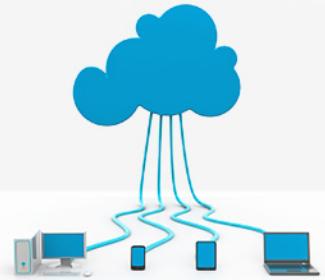
  - *writeObject*

- *ObjectInputStream* : is used to read them back from disc

```
ObjectInputStream inStream = new ObjectInputStream (
                                    new FileInputStream ("personnel.dat"));
```

  - *readObject*

```
Personnel person = (Personnel) inStream.readObject ( );
```



Serialization

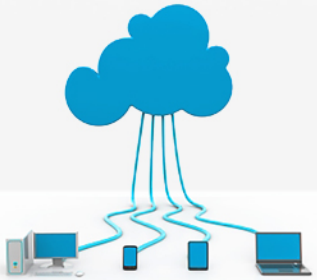
ObjectOutputStream & ObjectInputStream

To Write into an *ObjectOutputStream*:

```
FileOutputStream out = new FileOutputStream ( "FileURL" );
ObjectOutputStream oos = new ObjectOutputStream ( out );
oos.writeObject ("Today");
oos.writeObject (new Date ( ));
oos.flush ( );
```

To Read from an *ObjectInputStream*:

```
FileInputStream in = new FileInputStream ( "FileURL" );
ObjectInputStream ois = new ObjectInputStream ( in );
String today = (String) ois.readObject ( );
Date date = (Date) ois.readObject ( );
```



Serialization

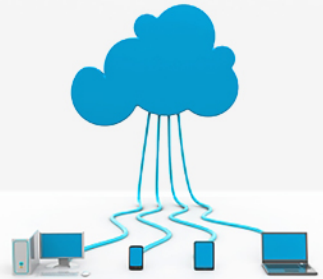
How to detect end-of-file?

use a for loop to read back the number of objects we believe that the file holds

➔ but this would be very bad practice in general

Recommended:

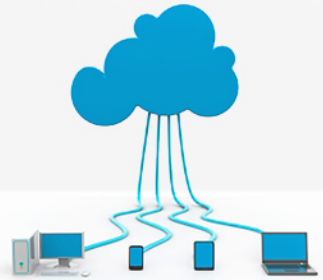
catch the *EOFException* that is generated when we read past the end of the file



Serialization

ArrayList

- An object of class *ArrayList* is like an array
- can dynamically increase or decrease in size according to an application's changing storage requirements
- can hold only references to objects, not values of primitive types
- E.g:
  - ArrayList<String> stringArray = new ArrayList<String> ( );
  - ArrayList<String> nameList = new ArrayList <> ( );

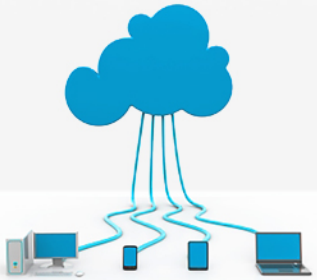


Serialization

ArrayLists and Serialisation

- Saving ArrayList is more efficient than saving a series of individual objects
- With binary files, pass an ArrayList instead of a series of strings
- Do not use PrintWriter
- Create an *ObjectOutputStream* object

```
ObjectOutputStream out = new ObjectOutputStream (
                         new FileOutputStream ("d://Khaled.dat"));
```



Serialization

Vectors Versus ArrayLists

A *Vector is* an alternative to using an *ArrayList*

Difference:

- Use *elementAt* method (instead of **get** method)

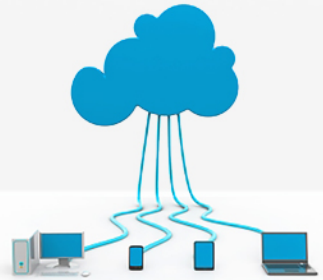
```
Vector<String> stringVector = new Vector<>( );
stringVector.add ( "Example" );
//Next step retrieves this element
String word = stringVector.elementAt ( 0 );
```

Serialization

Vectors Versus ArrayLists

- The *ArrayList* is faster, but is not **thread-safe**
- *Vector* is thread-safe
- if thread-safety is important to your program
  ➔ use a *Vector*

  otherwise
  ➔ use an **ArrayList**



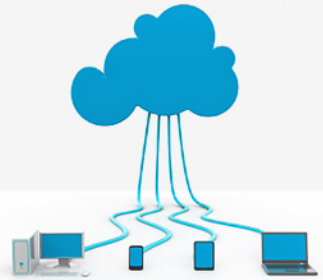
Random Access Files

- **RandomAccessFile** class allows data to be read from and written to any locations in a file.

- A file that is opened using the RandomAccessFile class is known as a random-access file.

- **RandomAccessFile** class implements **DataInput** and **DataOutput** interfaces.

- When creating a RandomAccessFile (in java), you can specify one of two modes: **r** (read-only) or **rw** (read and write).

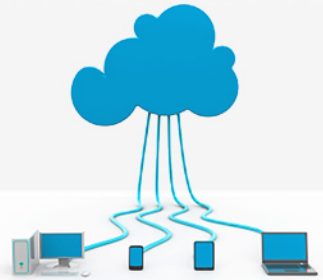


Random Access Files

- meaningfully called **direct access** files
- overcome problems of Serial Access Files
- fast and flexible

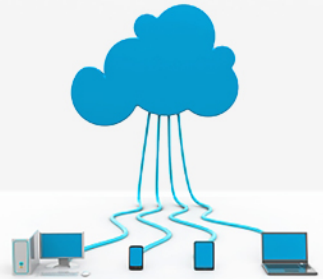
Problems:

1- all the (logical) records must be of the same length
2- given string field must be of the same length for all records on the file
3- numeric data is not in human-readable form

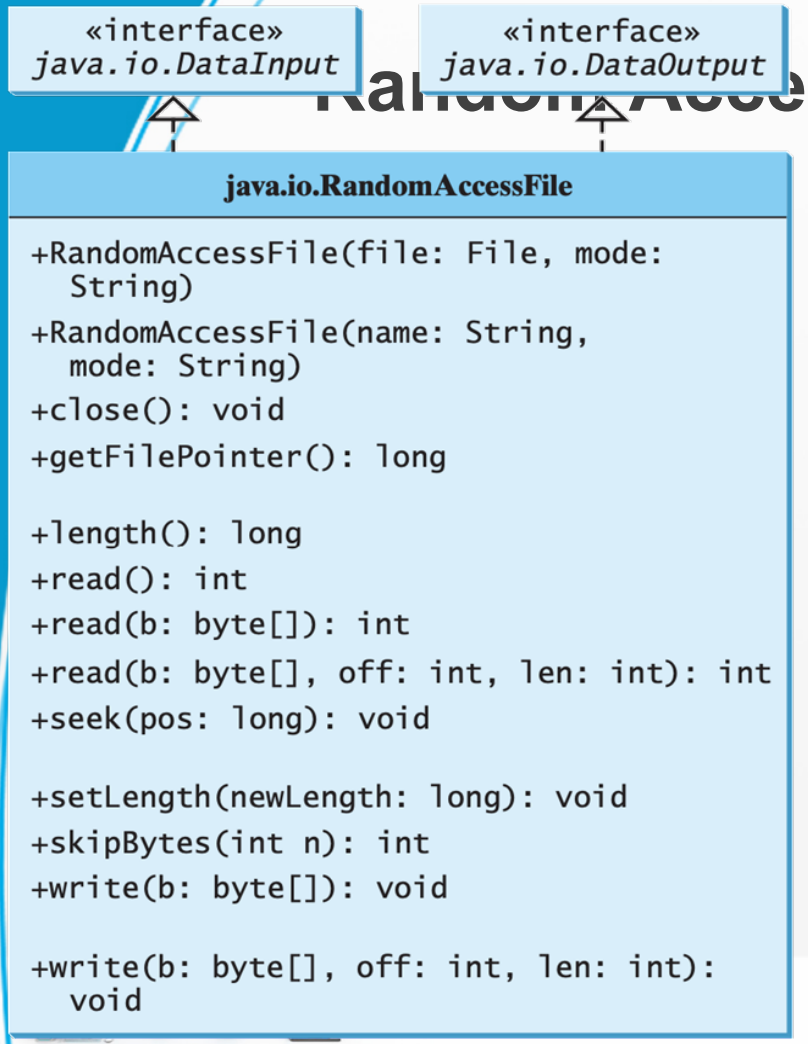


Random Access Files

- A random-access file consists of a sequence of bytes.
- A special marker called a file pointer is positioned at one of these bytes.
- A read or write operation takes place at the location of the file pointer.
- When a file is opened, the file pointer is set at the beginning of the file.
- When you read or write data to the file, the file pointer moves forward to the next data item.
- You can use the **.seek(**position**)** method to move the file pointer to a specified position.
- **.seek (0)** moves it to the beginning of the file
- **.seek (raf.length( ))** moves it to the end of the file



Random Access Files



`+RandomAccessFile(file: File, mode: String)`	Creates a `RandomAccessFile` stream with the specified `File` object and mode.
`+RandomAccessFile(name: String, mode: String)`	Creates a `RandomAccessFile` stream with the specified file name string and mode.
`+close(): void`	Closes the stream and releases the resource associated with it.
`+getFilePointer(): long`	Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.
`+length(): long`	Returns the number of bytes in this file.
`+read(): int`	Reads a byte of data from this file and returns –1 at the end of stream.
`+read(b: byte[]): int`	Reads up to `b.length` bytes of data from this file into an array of bytes.
`+read(b: byte[], off: int, len: int): int`	Reads up to `len` bytes of data from this file into an array of bytes.
`+seek(pos: long): void`	Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.
`+setLength(newLength: long): void`	Sets a new length for this file.
`+skipBytes(int n): int`	Skips over `n` bytes of input.
`+write(b: byte[]): void`	Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.
`+write(b: byte[], off: int, len: int): void`	Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

Random Access Files

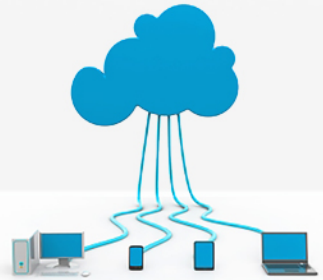
In order to move to the correct position for a particular record, we need to know two things:

- the size of records on the file
- the algorithm for calculating the appropriate position

Field size:

- String fields: decided by you — *readChar , writeChar*
  (in java, each character occupies 2 bytes)
- Numeric fields:

int	4 bytes	*readInt , writeInt*
long	8 bytes	*readLong , writeLong*
float	4 bytes	*readFloat , writeFloat*
double	8 bytes	*readDouble , writeDouble*



Random Access Files

Java:

1- create *RandomAccessFile* object

```
RandomAccessFile ranFile = new RandomAccessFile ("accounts.dat","rw");
```

"r" (for read-only access) or **"rw"** (for read-and-write access)

2- position the **file pointer** : specify the byte position within the file

```
ranFile.seek (500); //Move to byte 500 (the 501st byte)
```

The formula for calculating the position of any record on the file is

(Record No. –1) × 48 // record no * record size

3- method length returns the number of bytes in a file

➔ **number of records** in a file **= length / record size**

```
nbOfRecords = ranFile.length ( ) / 48;
```