EE3206 Java Programming and Applications

Lecture 7 File I/O

Intended Learning Outcomes

- To scan a string using the Scanner class.
- To discover file properties, delete and rename files using the File class.
- To distinguish between text I/O and binary I/O.
- To discover how I/O is processed in Java.
- To read/write files using various Java I/O classes
- To set character encoding for text I/O
- To understand how objects are serialized and what kind of objects can be serialized.
- To read/write the same file at random location using the RandomAccessFile class.

The File Class

- The File class provides a constructor to create a file handle. This file handle is then used by various file class methods to access the properties of a specific file.
  - To construct a File object:
    - File myFile = new File("/path/to/the/file");
  - Accessor methods:
    - getAbsolutePath(), getName(), getPath(), getParent(), lastModified(), length(), … etc
  - Mutator methods:
    - delete(), mkdir(), renameTo(), setReadOnly(), … etc
  - Boolean methods:
    - canRead(), canWrite(), exists(), isDirectory(), isFile(), isHidden(), … etc
- The File class also provides a platform independent abstraction for other I/O classes (see next part) to read/write from a specific file.

- Ex: Write a program that demonstrates how to use the File class to obtain the properties of a specific file.

TestFileClass

The Scanner Class

- A *Scanner* can be created to "scan" or tokenize a data source such as a string, a file or an input stream e.g. **System.in**.
  - breaks its input into tokens using a delimiter pattern, which by default matches whitespace (i.e. Space, Tab , Line Feed, Carriage Return).
  - token1 *delimiter* token2 *delimiter* token3 *delimiter* token4 … etc
- To change the delimiter, use the method useDelimiter()

```
String s = "Welcome to Java! Java is fun! Java is cool!";
Scanner scanner = new Scanner(s);
scanner.useDelimiter("Java");
while (scanner.hasNext())
  System.out.println(scanner.next());
```

```
Welcome to
!
 is fun!
 is cool!
```

- The hasNext() method returns true if there are more tokens from input.
- To read one character at a time, set the delimiter pattern to the empty string:
  - `sc.useDelimiter("");`
- Now each call to next returns a string consisting of a single character

Scanning File and Primitive Type Values

- To read some long numbers from a text file numbers.txt

```
Scanner sc = new Scanner(new File(“numbers.txt"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

- If a token is a primitive data type value, you can use the methods nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), or nextBoolean() to obtain it. For example, the following code adds all numbers in the string.

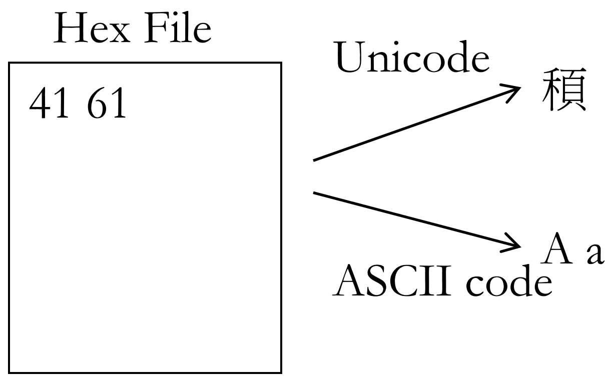
```
String s = "1 2 3 4";
Scanner scanner = new Scanner(s);

int sum = 0;
while (scanner.hasNext())
  sum += scanner.nextInt();

System.out.println("Sum is " + sum);
```

Text Data vs. Binary Data

- Computer data has its natural form in binary (1 and 0). To make it easy for human use, certain schemes are used to transform (encode/decode) binary values to text.
  - ASCII (single byte scheme, max 256 symbols)
  - Unicode UTF-16 (double byte scheme, max 65536 symbols)
    - Java system uses Unicode to represent characters
- For example, Java source files (*.java) are stored in text format and is readable to human, but after compilation Java bytecode files (*.class) are in binary format and are optimized for processing by the JVM.
- Binary format is a more efficient representation in terms of data size and processing time. Its major drawback is difficult for human to process.

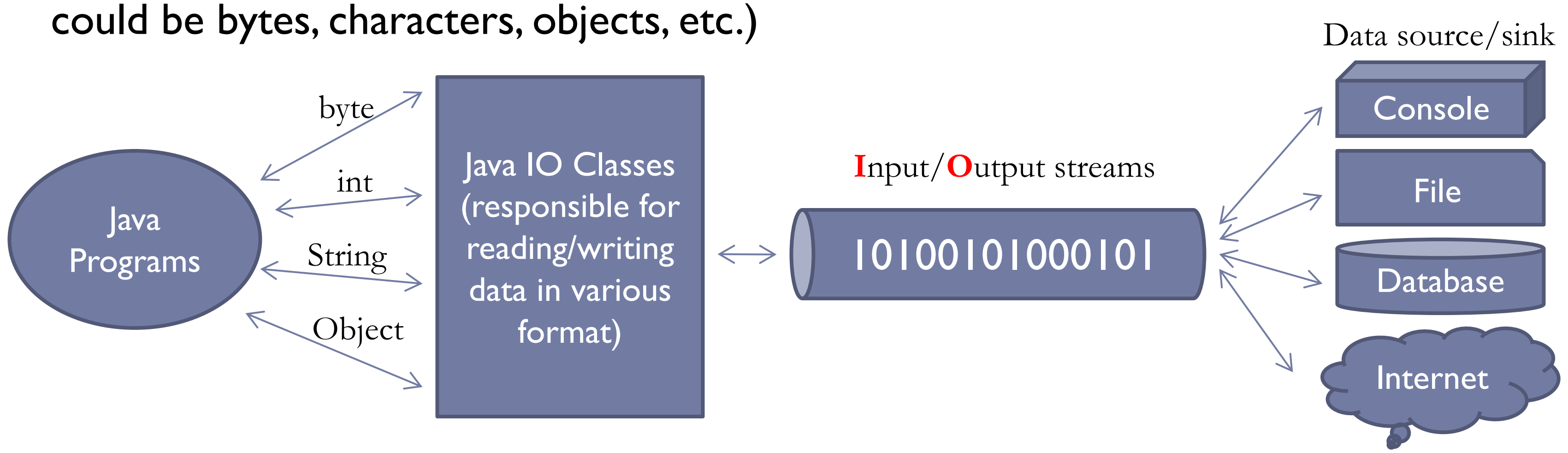
To represent 10 in binary format: 00001010
To represent 10 in text format using ASCII coding:

‘1’	’0’
0011 0001	0011 0000



How Is Data Being Read and Write?

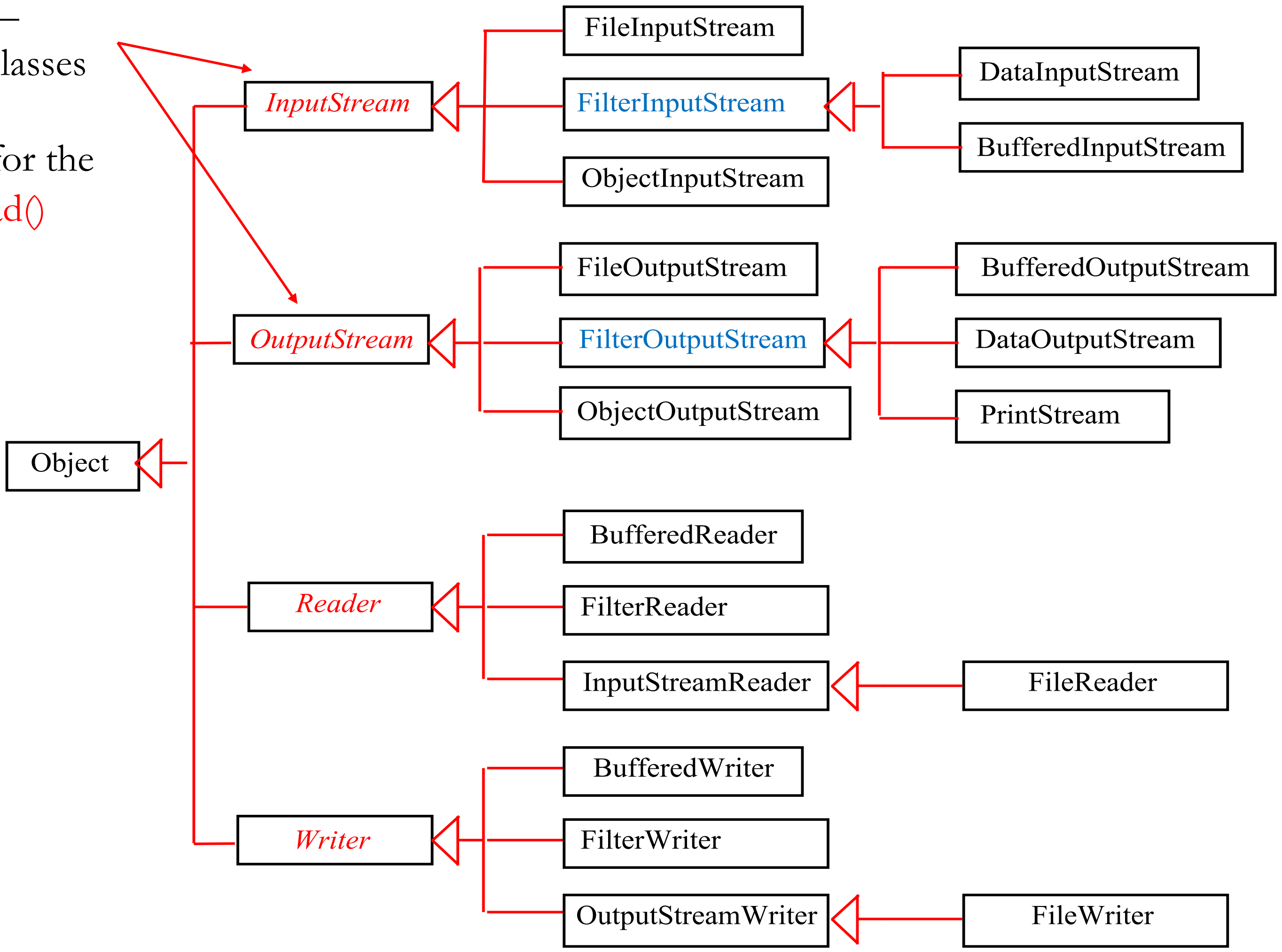
- I/O in Java is built on streams. A stream means an unbroken flow of data (which could be bytes, characters, objects, etc.)



- A **File** object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes.
  - FileReader, FileWriter // for text I/O, perform encoding/decoding implicitly using system default
  - FileInputStream, FileOutputStream // for binary I/O
  - DataInputStream, DataOutputStream // for binary I/O
  - BufferedInputStream, BufferedOutputStream // for binary I/O
  - ObjectInputStream, ObjectOutputStream // for binary I/O

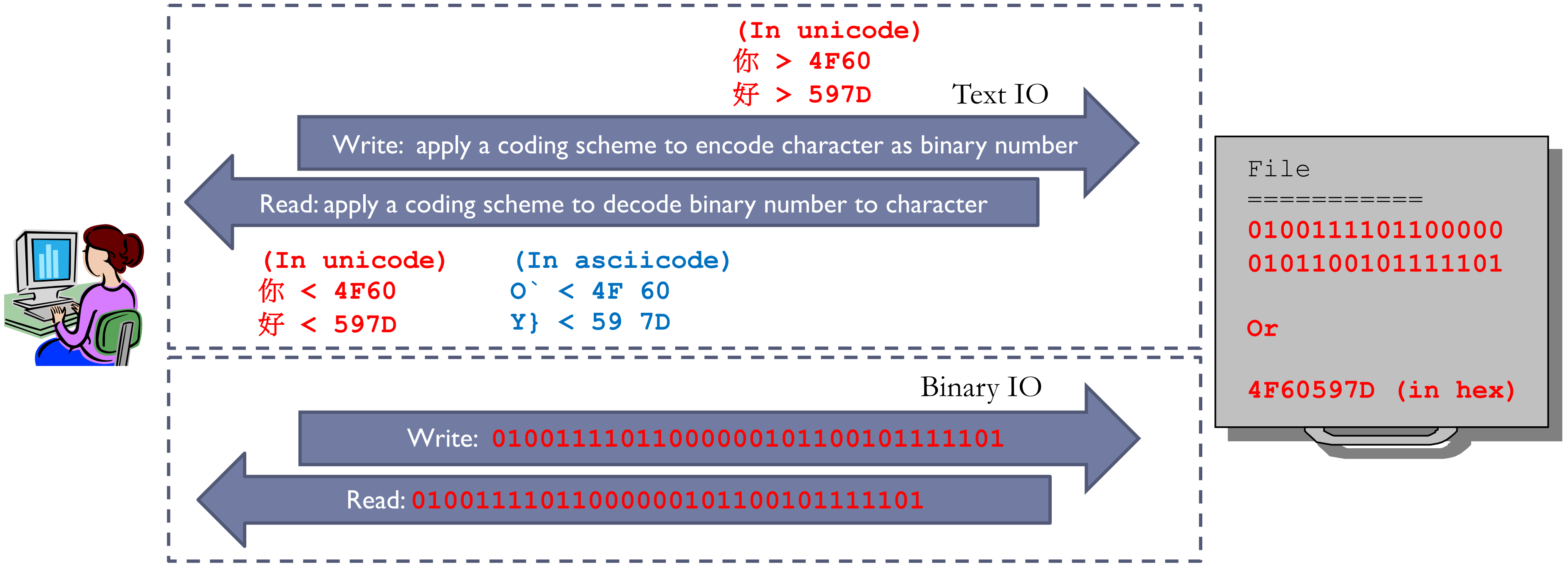
Summary of I/O Classes

Abstract Classes – requiring all subclasses to provide an implementation for the two methods: read() and write()



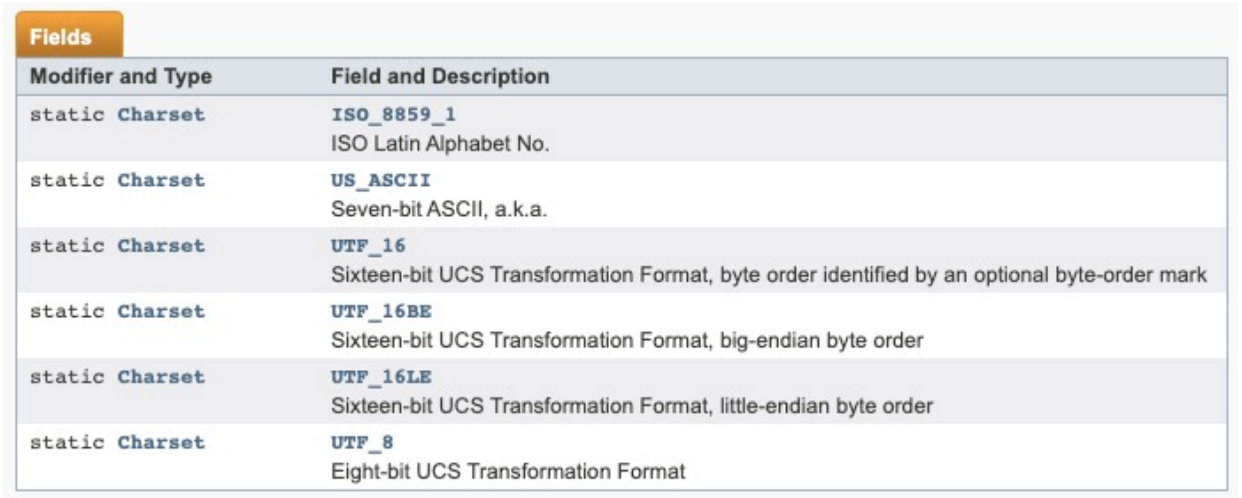
What Is the Difference of Text and Binary IO?

- **Reader** is a bridge from byte streams to character streams (same as **Writer**). It reads bytes and decodes them into characters using a specified charset (character encoding).
- The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted when not specified.
- Binary IO does not alter any data – raw read/write.


(In unicode)
你 > 4F60
好 > 597D
Text IO
Write: apply a coding scheme to encode character as binary number
Read: apply a coding scheme to decode binary number to character
(In unicode)
你 < 4F60
好 < 597D
(In asciicode)
O` < 4F 60
Y} < 59 7D
File
===========
0100111101100000
0101100101111101
Or
4F60597D (in hex)
Binary IO
Write: 01001111011000000101100101111101
Read: 01001111011000000101100101111101

Default Character Encoding

- Default Character encoding or Charset in Java is used by Java Virtual Machine (JVM) to convert bytes into a string of characters.
- During JVM start-up, Java gets character encoding by calling System.getProperty("file.encoding", "UTF-8"). In the absence of file.encoding attribute, Java uses "UTF-8" character encoding by default.
- `Charset` provides a convenient static method Charset.defaultCharset() which returns default character encoding in Java.
- `StandardCharsets` provides constant definitions for standard charsets that are available on every Java platform.



Fields

Modifier and Type	Field and Description
static Charset	ISO_8859_1 ISO Latin Alphabet No.
static Charset	US_ASCII Seven-bit ASCII, a.k.a.
static Charset	UTF_16 Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark
static Charset	UTF_16BE Sixteen-bit UCS Transformation Format, big-endian byte order
static Charset	UTF_16LE Sixteen-bit UCS Transformation Format, little-endian byte order
static Charset	UTF_8 Eight-bit UCS Transformation Format

Text I/O for Files

- **FileReader** and **FileWriter** are used for reading/writing streams of characters (16 bits) from/to a file.
- Read/write using system default encoding
- FileReader's common methods
  - FileReader(File file) // constructors
  - FileReader(String fileName)
  - public int read() // return the unicode of the next character in the stream
  - // or -1 if reach the end of the stream

~~FF FF~~ FF FF
(no use) (unicode)

- FileWriter's common methods
  - FileWriter(File file)
  - FileWriter(File file, boolean append) // append to the end of the file if set to true
  - FileWriter(String fileName)
  - FileWriter(String fileName, boolean append)
  - public void write(int c) // c is the unicode to be written
  - public void write(String str) // str is the String to be written

TestFileReaderWriter

Binary I/O for Files

- **FileInputStream** and **FileOutputStream** are used for reading/writing streams of raw bytes (8 bits) from/to a file.
- Read/write without any encoding/decoding (e.g. suitable for image data)
- FileInputStream's common methods
  - FileInputStream(String filename) // constructors
  - FileInputStream(File file)
  - public int read() // return the next byte of data,
  - // or -1 if the end of the file is reached

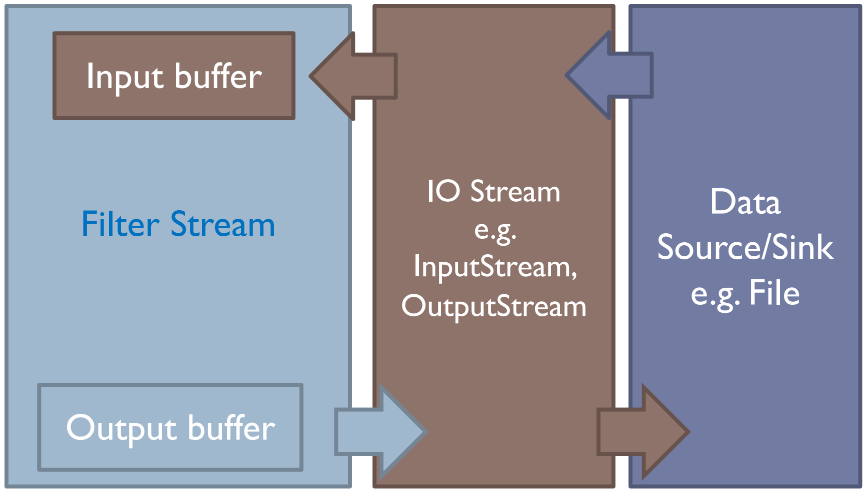
~~FF FF FF~~ FF
(no use) (lower order byte)

- FileOutputStream's common methods
  - FileOutputStream(String filename)
  - FileOutputStream(File file)
  - FileOutputStream(String filename, boolean append) // append to the end of the file if set to true
  - FileOutputStream(File file, boolean append)
  - public void write(int b) // b is the byte to be written

TestFileStreams

Filter Streams

- Filter streams are streams that filter bytes for some purpose. When they are constructed, an **InputStream** or **OutputStream** object is supplied. The filter streams basically acts like a wrapper and pass all requests to the contained stream object.
  - Adding functionalities to underlying streams
  - Have a buffer to temporarily cache the data from/to the underlying streams and convert or manipulate the data accordingly
- **FilterInputStream** and **FilterOutputStream** are the base classes for filtering data.
  - For example, the GZIPInputStreasm can be used to read compressed data files.


Input buffer
Filter Stream
Output buffer
IO Stream e.g. InputStream, OutputStream
Data Source/Sink e.g. File

```
String fname = “myFile.gz”;  // GZIP file

GZIPInputStream gs =
     new GZIPInputStream(new FileInputStream(fname));
InputStreamReader reader = new InputStreamReader(gs);
BufferedReader in = new BufferedReader(reader);

String line = in.readLine();
```

Data I/O Streams

- While file streams are primitive streams whose sources or destinations are files, data streams are streams whose sources and destinations are other streams!
- They are therefore known as wrappers because they can wrap other stream object mechanisms inside a more powerful one.
- **DataInputStream** and **DataOutputStream** provide extra methods for you to read/write primitive data types (more bytes instead of each a byte). The methods are named readXX() and writeXX() where XX is a primitive data type (e.g. Boolean, Byte, Char, Double, Float,… check API for more details).
- Constructors:
  - DataInputStream(InputStream instream)
  - DataOutputStream(OutputStream outstream)
- For example:
  - DataInputStream infile = new DataInputStream(new FileInputStream("in.dat"));
  - DataOutputStream outfile = new DataOutputStream(new FileOutputStream("out.dat"));

TestDataStreams

Buffered I/O Streams

- Another pair of wrappers is **BufferedInputStream** and **BufferedOutputStream**.
- Unlike data streams providing extra read/write methods, buffered streams create internal byte-buffer to speed up read/write operations. More bytes are read/write to fill up the buffer each time, and hence reduce the number of direct I/O operations to the underlying streams.
- Constructors:
  - BufferedInputStream(InputStream in)
  - BufferedInputStream(InputStream in, int bufferSize)
  - BufferedOutputStream(OutputStream out)
  - BufferedOutputStream(OutputStreamr out, int bufferSize)

The size of the internal buffer (byte array)

- For example:
  - BufferedInputStream infile = new BufferedInputStream (new FileInputStream("in.dat"));
  - BufferedOutputStream outfile = new BufferedOutputStream (new FileOutputStream("out.dat"));

TestBufferedStreams

Object I/O Streams

- **ObjectInputStream** and **ObjectOutputStream** is another pair of wrapper that can be used to read/write objects from/to stream.
- During the write process, an object is first flatten or serialized to become a stream of bytes. The same stream of bytes are read and de-serialize to re-construct the object during the read process.
- Similar to data streams, object streams provide extra methods in the form of readXX() and writeXX() where XX is a primitive data type or Object (check API for more details).
- Constructors:
  - ObjectInputStream(InputStream in)
  - ObjectOutputStream(OutputStream out)
- For example:
  - ObjectInputStream infile = new ObjectInputStream (new FileInputStream("in.dat"));
  - ObjectOutputStream outfile = new ObjectOutputStream (new FileOutputStream("out.dat"));

TestObjectStreams

The Serializable Interface

- Not all objects can be written to an output stream. Objects that can be written to an object stream is said to be serializable. A serializable object is an instance of the java.io.Serializable interface. So the class of a serializable object must implement java.io.Serializable.
- The Serializable interface is a marker interface which has no method but merely for denoting certain properties in the class. So you don't need to add additional code in your class that implements Serializable. Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays.

```
// instances of Foo cannot be written to object stream
public class Foo {
  private int v1;
}
```

⇩

```
// instances of Foo can be written to object stream now
public class Foo implements Serializable {
  private int v1;
}
```

The transient Keyword

- If an object is an instance of Serializable, but it contains data fields that are non-serializable, can the object be written to object streams?
- The answer is NO. In this case, you may mark these non-serializable data fields with the keyword transient. This modifier tells the JVM to ignore these fields when writing the object to an object stream.

```
public class Foo implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}
class A { }                  // A is not serializable
```

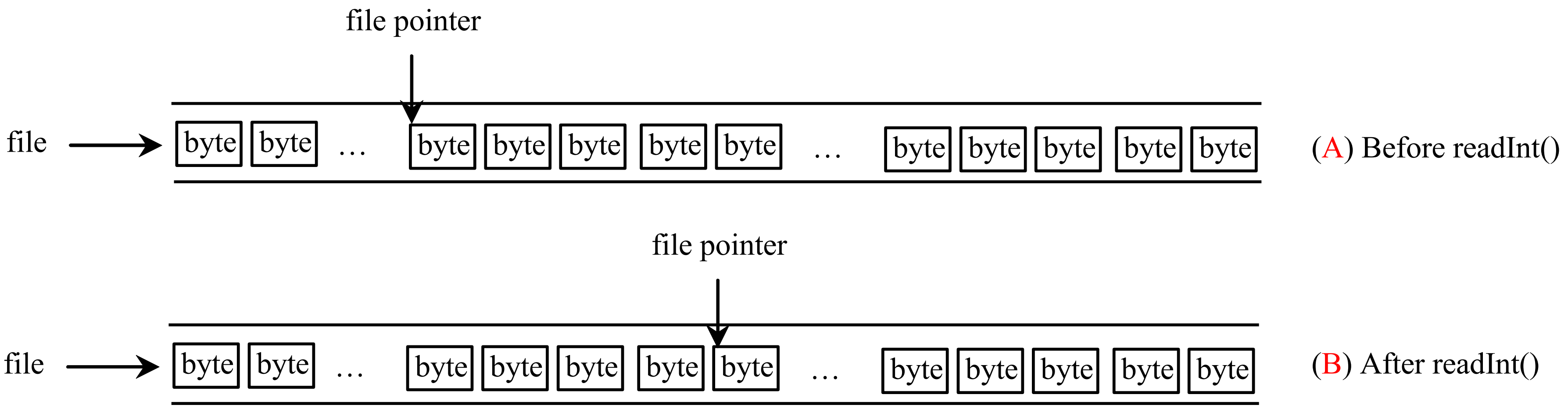
- When an object of the Foo class is serialized, only variable v1 is serialized. Variable v2 is not serialized because it is a static variable, and variable v3 is not serialized because it is marked transient. If v3 were not marked transient, the java.io.NotSerializableException would occur.

Random Access File

- All of the streams you have used so far are known as read-only or write-only streams.
- The external files of these streams are sequential files that data cannot be inserted in the middle of the file.
- It is often necessary to modify files or to insert new records into files. Java provides the **RandomAccessFile** class to allow a file to be read from and write to at random locations.
- Many methods in RandomAccessFile are the same as those in DataInputStream and DataOutputStream. For example, readInt(), readLong(), writeDouble(), readLine(), writeInt(), and writeLong() can be used in data input stream or data output stream as well as in RandomAccessFile streams.

Random Access File - File Pointer

- A random access file consists of a sequence of bytes. There is a special marker called file pointer that 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 sets at the beginning of the file. When you read or write data to the file, the file pointer moves forward to the next data.
- For example, if you read an int value using readInt(), the JVM reads four bytes from the file pointer and now the file pointer is four bytes ahead of the previous location.



Random Access File - Common Methods

- To construct RandomAccessFile:
  - RandomAccessFile raf = new RandomAccessFile("test.dat", "rw");   //allows read and write
  - RandomAccessFile raf = new RandomAccessFile("test.dat", "r");   //read only
- To returns the current file pointer offset, in bytes, from the beginning of the file to where the next read or write occurs.
  - long getFilePointer()
- To set the file pointer offset, measured from the beginning of this file, at which the next read or write occurs.
  - void seek(long pos)
- To return the length of the file.
  - long length()

TestRandomAccessFile