



File I/O

The File Class

- ❑ The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
- ❑ The filename is a string.
- ❑ The File class is a wrapper class for the file name and its directory path.

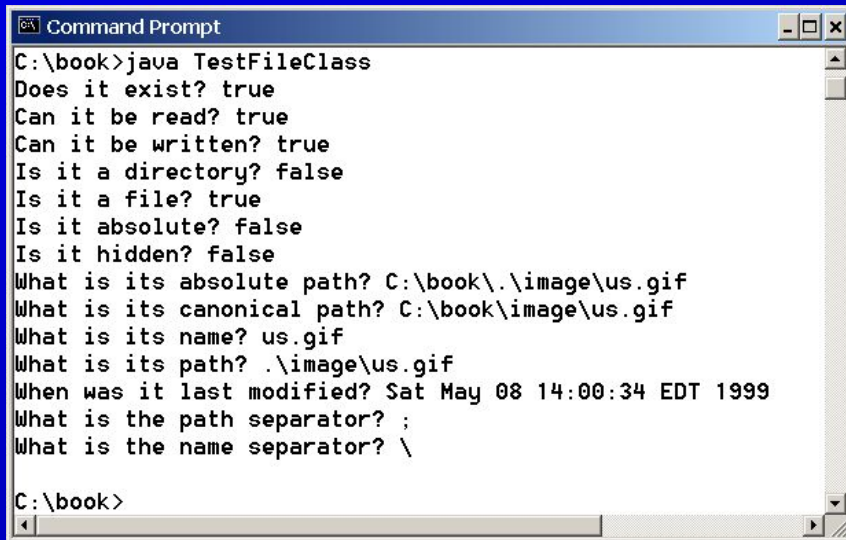


Obtaining file properties and manipulating file

java.io.File	
+File(pathname: String)	Creates a File object for the specified pathname. The pathname may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. child may be a filename or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of <i>hidden</i> is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period character '.'.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the pathname, resolves symbolic links (on Unix platforms), and converts drive letters to standard uppercase (on Win32 platforms).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\\book\\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\\book.
+lastModified(): long	Returns the time that the file was last modified.
+delete(): boolean	Deletes this file. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames this file. The method returns true if the operation succeeds.

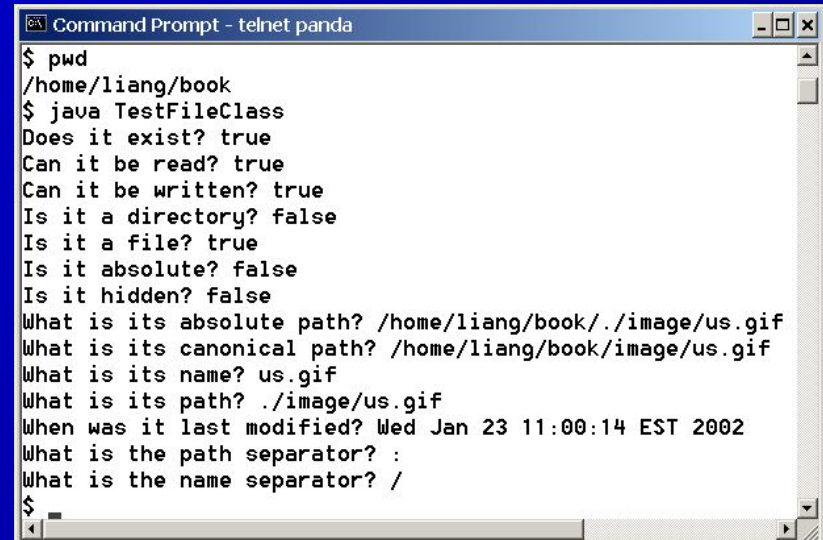
Problem: Explore File Properties

Objective: Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties. Figure 16.1 shows a sample run of the program on Windows, and Figure 16.2 a sample run on Unix.



```
Command Prompt
C:\book>java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? C:\book\image\us.gif
What is its canonical path? C:\book\image\us.gif
What is its name? us.gif
What is its path? image\us.gif
When was it last modified? Sat May 08 14:00:34 EDT 1999
What is the path separator? ;
What is the name separator? \

C:\book>
```



```
Command Prompt - telnet panda
$ pwd
/home/liang/book
$ java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? /home/liang/book/image/us.gif
What is its canonical path? /home/liang/book/image/us.gif
What is its name? us.gif
What is its path? image/us.gif
When was it last modified? Wed Jan 23 11:00:14 EST 2002
What is the path separator? /
What is the name separator? /

$
```

TestFileClass

Run

Text I/O

- 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.
- The objects contain the methods for reading/writing data from/to a file.



Reading Data Using Scanner

java.util.Scanner

+Scanner(source: File)

+Scanner(source: String)

+close()

+hasNext(): boolean

+next(): String

+nextByte(): byte

+nextShort(): short

+nextInt(): int

+nextLong(): long

+nextFloat(): float

+nextDouble(): double

+useDelimiter(pattern: String):
Scanner

Creates a Scanner that produces values scanned from the specified file.

Creates a Scanner that produces values scanned from the specified string.

Closes this scanner.

Returns true if this scanner has another token in its input.

Returns next token as a string.

Returns next token as a byte.

Returns next token as a short.

Returns next token as an int.

Returns next token as a long.

Returns next token as a float.

Returns next token as a double.

Sets this scanner's delimiting pattern.

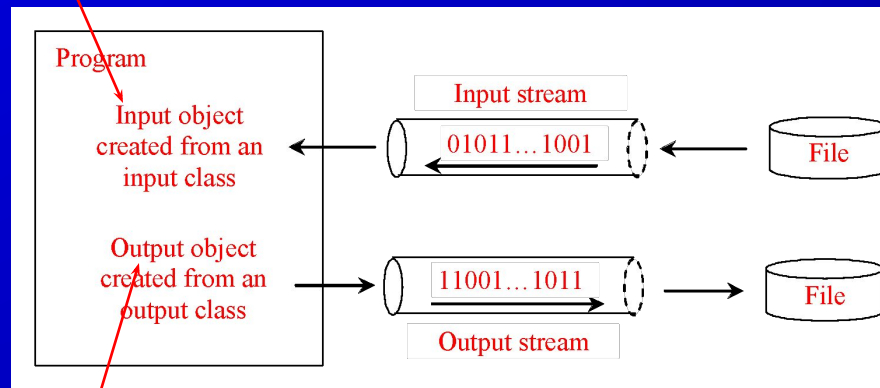
ReadData

Run

How is I/O Handled in Java?

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.

```
Scanner input = new Scanner(new File("temp.txt"));  
System.out.println(input.nextLine());
```



```
PrintWriter output = new PrintWriter("temp.txt");  
output.println("Java 101");  
output.close();
```

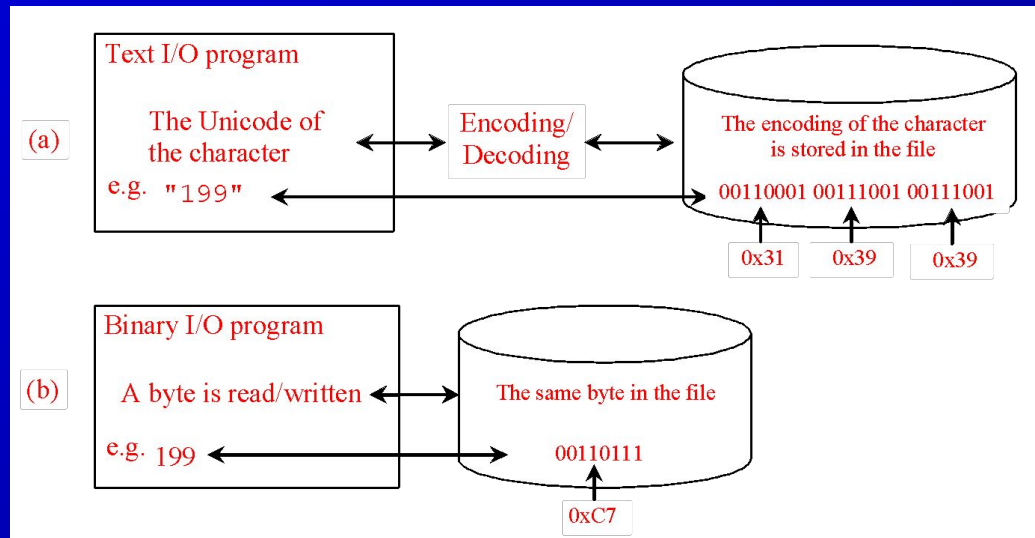


Text File vs. Binary File

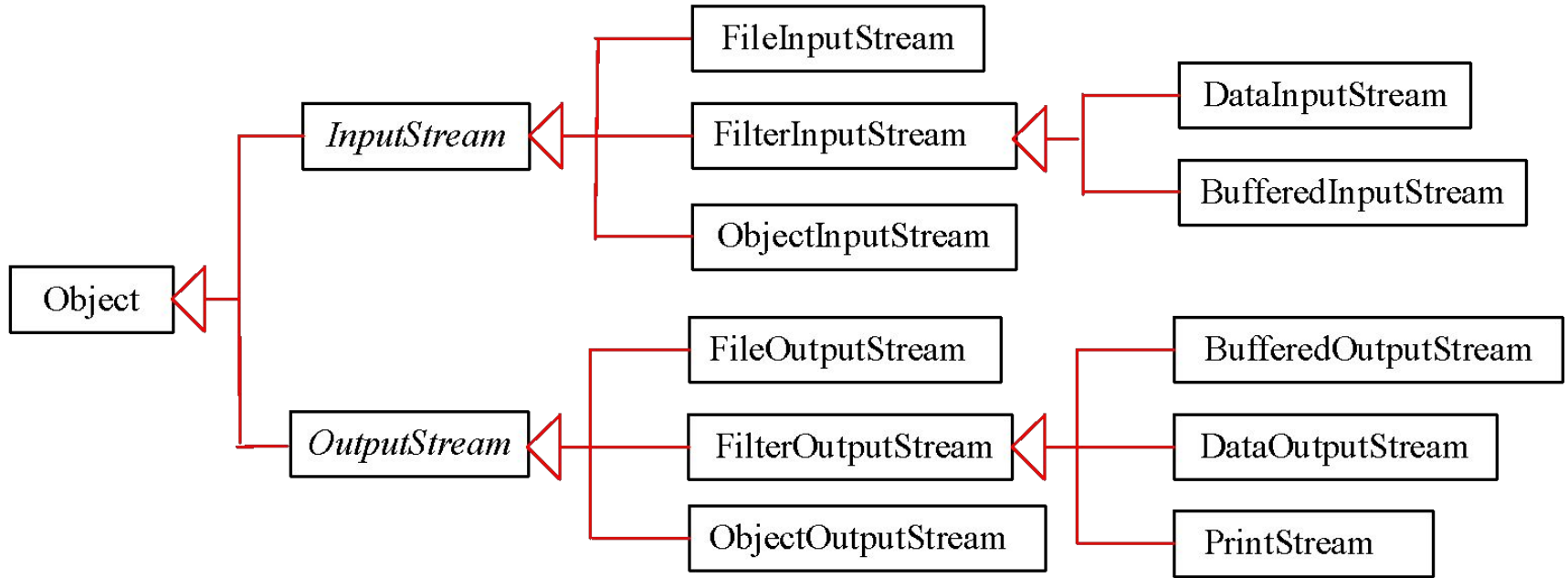
- Data stored in a text file are represented in human-readable form. Data stored in a binary file are represented in binary form. You cannot read binary files. Binary files are designed to be read by programs. For example, the Java source programs are stored in text files and can be read by a text editor, but the Java classes are stored in binary files and are read by the JVM. The advantage of binary files is that they are more efficient to process than text files.
- Although it is not technically precise and correct, you can imagine that a text file consists of a sequence of characters and a binary file consists of a sequence of bits. For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7.

Binary I/O

- Text I/O requires encoding and decoding. The JVM converts a Unicode to a file specific encoding when writing a character and converts a file specific encoding to a Unicode when reading a character.
- Binary I/O does not require conversions. When you write a byte to a file, the original byte is copied into the file. When you read a byte from a file, the exact byte in the file is returned.



Binary I/O Classes



InputStream

The value returned is a byte as an int type.

java.io.InputStream

+read(): int

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.

+read(b: byte[]): int

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.

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

Reads bytes from the input stream and stores into 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.

+available(): int

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

+close(): void

Closes this input stream and releases any system resources associated with the stream.

+skip(n: long): long

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

+markSupported(): boolean

Tests if this input stream supports the mark and reset methods.

+mark(readlimit: int): void

Marks the current position in this input stream.

+reset(): void

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

OutputStream

The value is a byte as an int type.

java.io.OutputStream

+*write(int b): void*

+*write(b: byte[]): void*

+*write(b: byte[], off: int, len: int): void*

+*close(): void*

+*flush(): void*

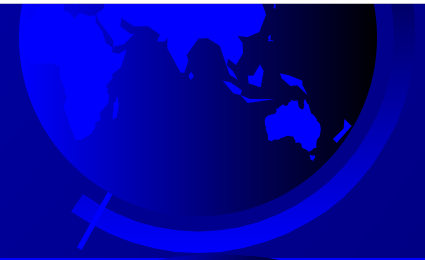
Writes the specified byte to this output stream. The parameter *b* is an int value. (byte)*b* is written to the output stream.

Writes all the bytes in array *b* to the output stream.

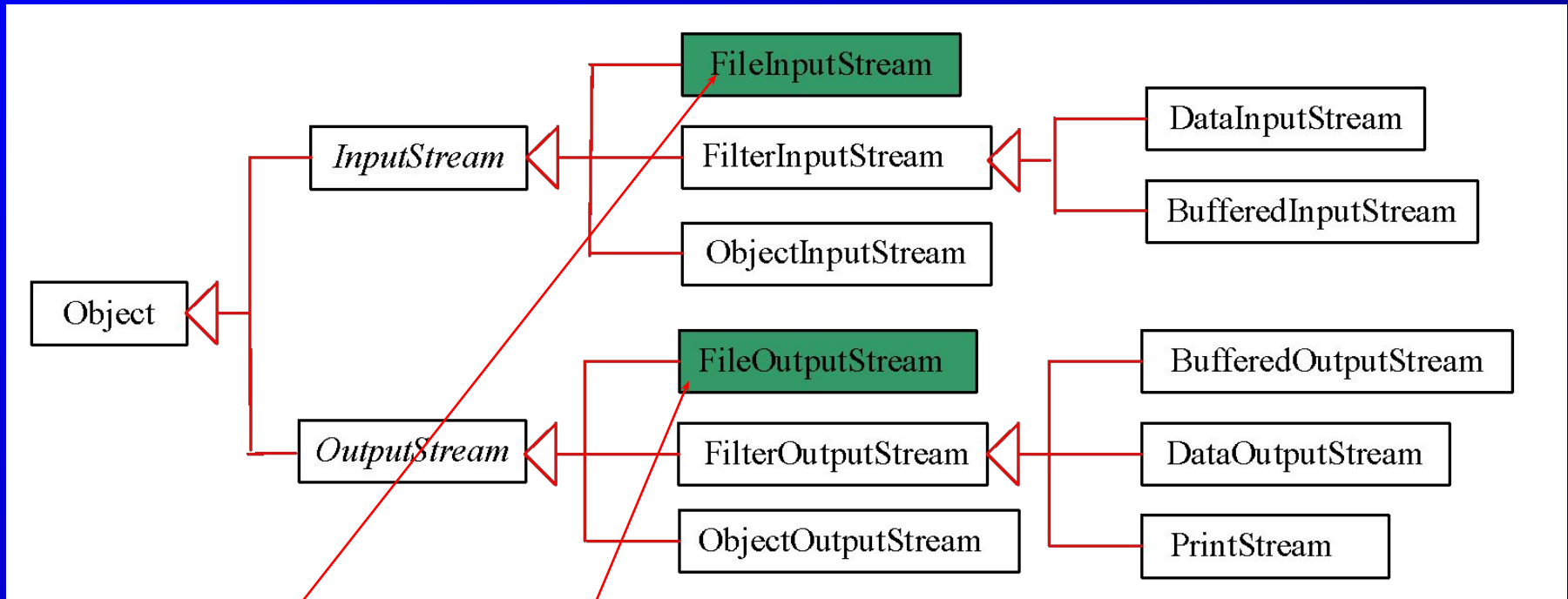
Writes *b[off]*, *b[off+1]*, ..., *b[off+len-1]* into the output stream.

Closes this input stream and releases any system resources associated with the stream.

Flushes this output stream and forces any buffered output bytes to be written out.



FileInputStream/FileOutputStream



`FileInputStream/FileOutputStream` associates a binary input/output stream with an external file. All the methods in `FileInputStream/FileOutputStream` are inherited from its superclasses.



FileInputStream

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.



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.



TestFileStream

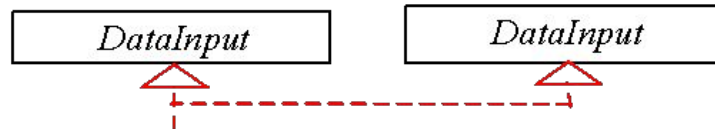
Run

Random Access Files

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 cannot be updated without creating a new 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.



RandomAccessFile



`java.io.RandomAccessFile`

- +RandomAccessFile(file: File, mode: String)
- +RandomAccessFile(name: String, mode: String)
- +close(): void
- +getFilePointer(): long
- +length(): long
- +read(): int
- +read(b: byte[]): int
- +read(b: byte[], off: int, len: int) : int
- +seek(long pos): void
- +setLength(newLength: long): void
- +skipBytes(int n): int
- +write(b: byte[]): void
- +write(byte b[], int off, int len)
- +write(b: byte[], off: int, len: int): void

Creates a RandomAccessFile stream with the specified File object and mode.

Creates a RandomAccessFile stream with the specified file name string and mode.

Closes the stream and releases the resource associated with the stream.

Returns the offset, in bytes, from the beginning of the file to where the next read or write occurs.

Returns the length of this file.

Reads a byte of data from this file and returns -1 at the end of stream.

Reads up to b.length bytes of data from this file into an array of bytes.

Reads up to len bytes of data from this file into an array of bytes.

Sets the offset (in bytes specified in pos) from the beginning of the stream to where the next read or write occurs.

Sets a new length of this file.

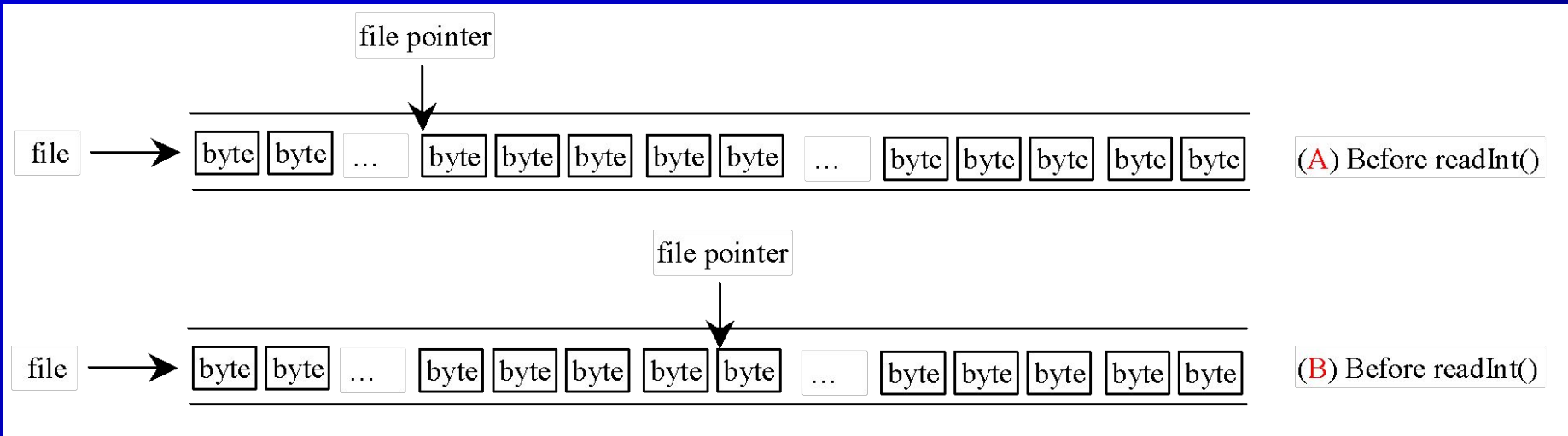
Skips over n bytes of input discarding the skipped bytes.

Writes b.length bytes from the specified byte array to this file, starting at the current file pointer.

Writes len bytes from the specified byte array starting at offset off to this 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.



RandomAccessFile Methods

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.



RandomAccessFile Methods, cont.

- `void seek(long pos) throws IOException;`

Sets the offset from the beginning of the `RandomAccessFile` stream to where the next read or write occurs.

- `long getFilePointer() throws IOException;`

Returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.



RandomAccessFile Constructor

```
RandomAccessFile raf =  
    new RandomAccessFile("test.dat", "rw");  
    //allows read and write
```

```
RandomAccessFile raf =  
    new RandomAccessFile("test.dat", "r");  
    //read only
```



- For further reading please explore
 - <https://www.javatpoint.com/java-io>

