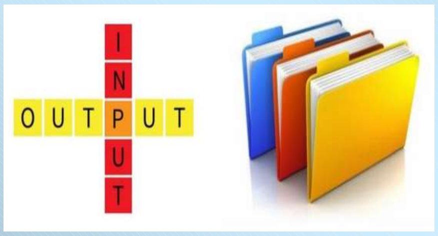
File Processing



Objectives

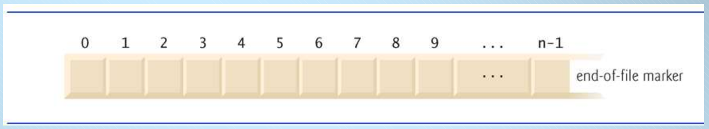
- At the end of this topic, you should be able to
  - Understand the concept and types of files
  - Use File object to get info about a file
  - Use JFileChooser object to let user select a file
  - Understand the basic of I/O streams
  - Distinguish between text I/O and binary I/O
  - Read and write to binary files.
  - Read and write to text files.

Introduction

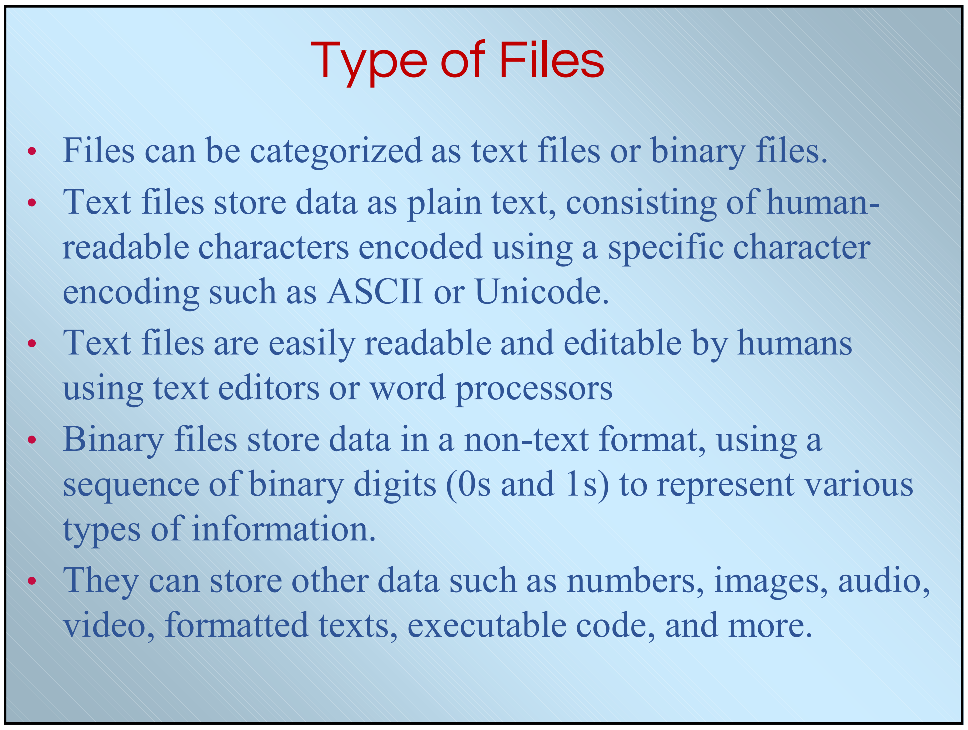
- Data stored in variables and arrays is temporary
  - It’s lost when the program terminates
- Computers use **files** for long-term retention of data
- Files are stored on **secondary storage devices**
  - hard disks, SSD, flash drives, optical discs and more.
- Data maintained in files is **persistent data** because it exists beyond the duration of program execution.

File

- File is regarded as a sequence of bytes:

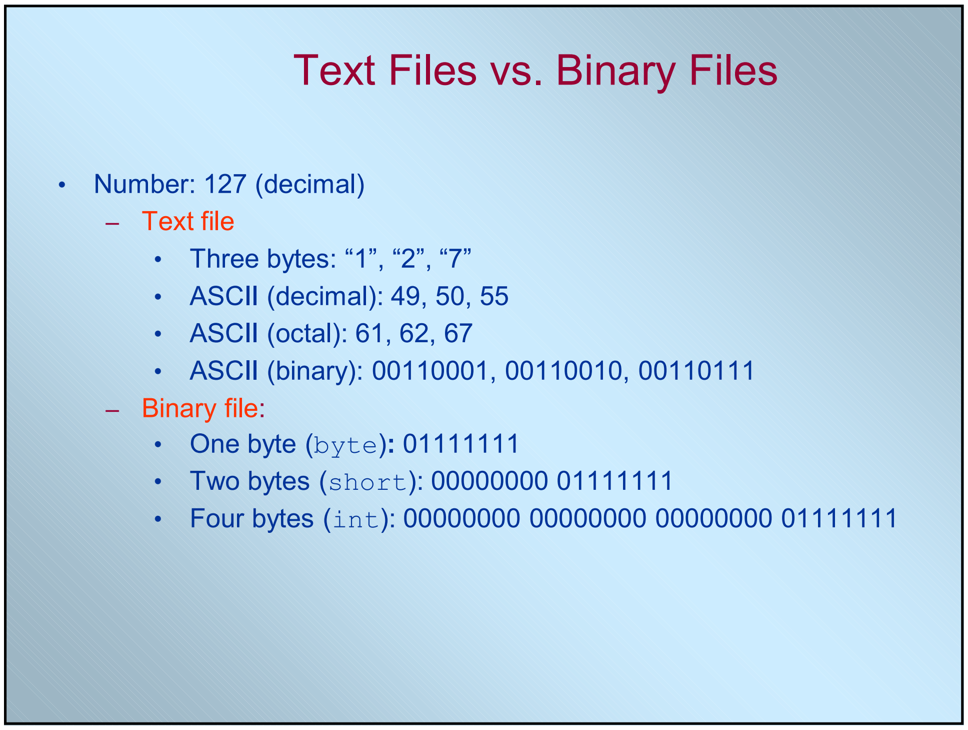


- When a file is read, computer delivers some of those bytes to the program.
- When a file is written, computer accepts some bytes from the program and saves them in part of the file.
- Computer makes no distinction between eg. image files and text files. Its all bytes to the hardware. What those bytes are used for is up to the software.



Type of Files

- Files can be categorized as text files or binary files.
- Text files store data as plain text, consisting of human-readable characters encoded using a specific character encoding such as ASCII or Unicode.
- Text files are easily readable and editable by humans using text editors or word processors
- Binary files store data in a non-text format, using a sequence of binary digits (0s and 1s) to represent various types of information.
- They can store other data such as numbers, images, audio, video, formatted texts, executable code, and more.



Text Files vs. Binary Files

- Number: 127 (decimal)
  - Text file
    - Three bytes: “1”, “2”, “7”
    - ASCII (decimal): 49, 50, 55
    - ASCII (octal): 61, 62, 67
    - ASCII (binary): 00110001, 00110010, 00110111
  - Binary file:
    - One byte (`byte`): 01111111
    - Two bytes (`short`): 00000000 01111111
    - Four bytes (`int`): 00000000 00000000 00000000 01111111

The File Class

- An object created from the File class (from java.io). can be used to obtain info about file
- A File object can represent a file or a directory

```
File inFile = new File("sample.dat");
```

Creates File object for the file sample.dat in the current directory.

```
File inFile = new File
           ("C:/SamplePrograms/test.dat");
```

Creates File object for the file test.dat in the directory C:\SamplePrograms using the generic file separator / and providing the absolute pathname.

An **absolute pathname** contains all the directories, starting with the root directory, that lead to a specific file or directory.
A relative **pathname** is relative to the current directory.

Some File Methods

```
if ( inFile.exists( ) ) {
```

To check if inFile is associated to a file that exist.

```
if ( inFile.isFile() ) {
```

```
if ( inFile.isDirectory() ) {
```

To see if inFile is associated to a file. If false, it is a directory. Also, can test directly if it is a directory.

```
File directory = new

         File("C:/JavaPrograms/Ch12");

String filename[] = directory.list();

for (int i = 0; i < filename.length; i++) {
         System.out.println(filename[i]);
}
```

List the name of all files in the directory C:\JavaProjects\Ch12

Some File Methods

```
if ( inFile.length( ) ) {
```

To see the size of the file in bytes represented by inFile

```
if ( inFile.getName() ) {
```

To get the name of the file represented by inFile

```
if ( inFile.canRead() ) {
```

To see if inFile is associated to a file that exist & can be read

```
if ( inFile.canWrite() ) {
```

To see if inFile is associated to a file that exist & can be written

The JFileChooser Class

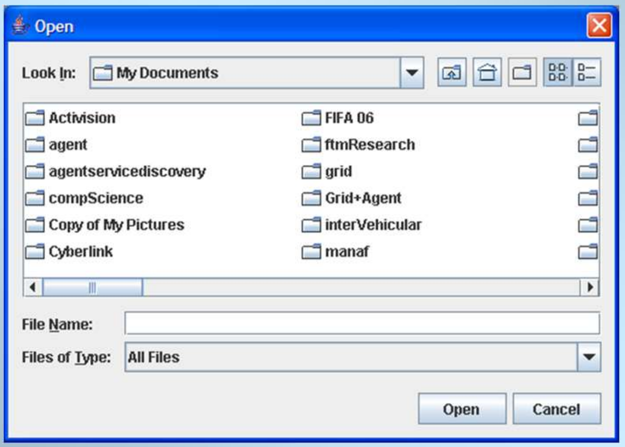
- **A javax.swing.JFileChooser object allows the user to select a file.**

```
JFileChooser chooser = new JFileChooser( );

chooser.showOpenDialog(null);
```

To start the listing from a specific directory:

```
JFileChooser chooser = new JFileChooser("D:/JavaPrograms/Ch12");

chooser.showOpenDialog(null);
```

JFileChooser `showOpenDialog()`



Getting Info from JFileChooser

```
int status = chooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION) {
        JOptionPane.showMessageDialog(null, "Open is clicked");

} else { //== JFileChooser.CANCEL_OPTION
        JOptionPane.showMessageDialog(null, "Cancel is
clicked");
}
```

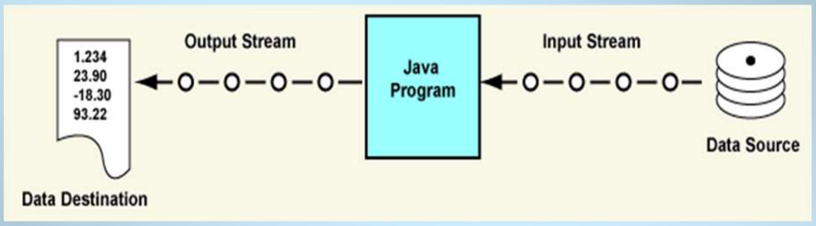
```
File selectedFile  = chooser.getSelectedFile();
```

```
File currentDirectory = chooser.getCurrentDirectory();
```

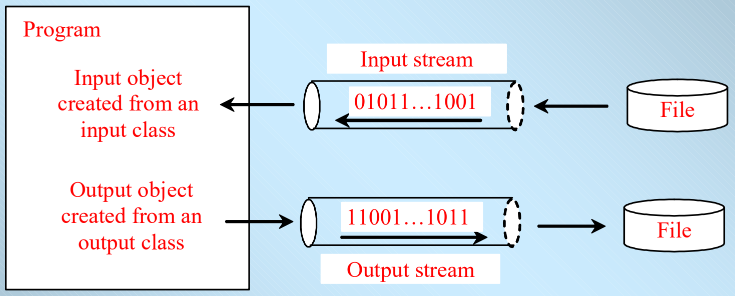
I/O Streams

- To read data from or write data to a file, we must create one of the Java stream objects and attach it to the file.
- A ***stream*** is a sequence of data items, usually 8-bit bytes.
- Java has two types of streams: an ***input stream*** and an ***output stream***.
- An ***input stream*** has a source form which the data items come, and an ***output stream*** has a destination to which the data items are going.

I/O Streams

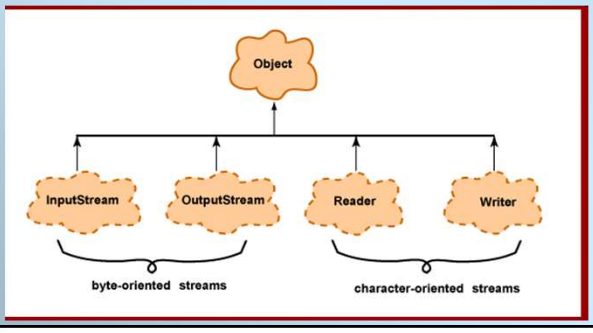


I/O Streams


Program
Input object created from an input class
Output object created from an output class
Input stream
01011…1001
File
11001…1011
File
Output stream

Java I/O Streams

- IO streams are either character-oriented or byte-oriented.
- Character-oriented IO has special features for handling character data (text files).
- Byte-oriented IO is for all types of data (binary files)



IO class Hierarchy in java.io package

Streams for Byte-level Binary File I/O

- FileOutputStream and FileInputStream are two stream objects that facilitate file access.
- FileOutputStream allows us to output a sequence of bytes; values of data type byte.
- FileInputStream allows us to read in an array of bytes.

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.

FileOutputStream

The FileOutputStream provides several methods that are commonly used for writing data to a file:

- write(byte[] b) : Writes an array of bytes to the file.
- write(byte[] b, int off, int len) : Writes a portion of an array of bytes ‘b’ to the file, starting from the specified index ‘off’ and writing the specified length ‘len’.
- write(int b) : Writes a single byte to the file.
- close() : Closes the output stream, flushing any buffered content and releasing system resources.

19

Sample: Byte-level Binary File Output

```
//set up file and stream
File  outFile   = new File("sample1.data");

FileOutputStream
      outStream = new FileOutputStream( outFile );

//data to save
byte[] byteArray = {10, 20, 30, 40,
                    50, 60, 70, 80};

//write data to the stream
outStream.write( byteArray );

//output done, so close the stream
outStream.close();
```

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.

21

FileInputStream

The FileInputStream provides several methods that are commonly used for reading data from a file:

- read():int : Read the next byte of data from a file. The value is returned as int value in the range 0 to 255.
- read(byte[] b, int off, int len):int : Read 'len' bytes and stores into array b starting at index 'off'.
- read(byte[] b):int : Reads up to b.length bytes into array b.
- close() : Closes the input stream and releasing any system resources associated with the stream

In all read() methods, the value -1 is returned if no byte is available because the end of the file has been reached.

22

Sample: Byte-level Binary File Input

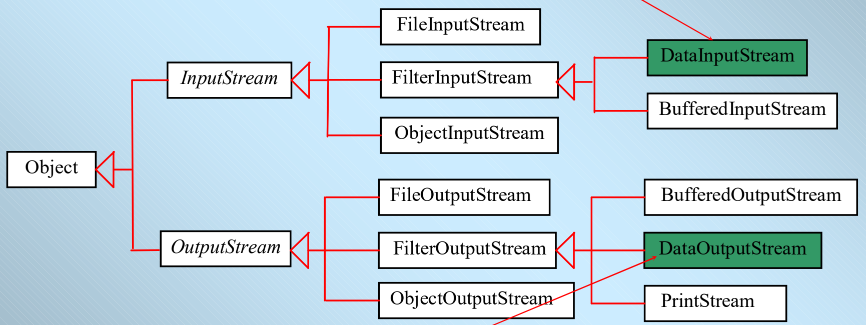
```
//set up file and stream
File            inFile   = new File("sample1.data");
FileInputStream inStream = new FileInputStream(inFile);

//set up an array to read data in
int    fileSize  = (int)inFile.length();
byte[] byteArray = new byte[fileSize];

//read data in and display them
inStream.read(byteArray);
for (int i = 0; i < fileSize; i++) {
       System.out.println(byteArray[i]);
}

//input done, so close the stream
inStream.close();
```

Streams for **Data-Level** Binary File I/O

- DataOutputStream are used to output primitive data values
- DataInputStream are used to input primitive data values
- To read the data correctly, we must know the order of the data stored and their data types

DataInputStream/DataOutputStream

DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.

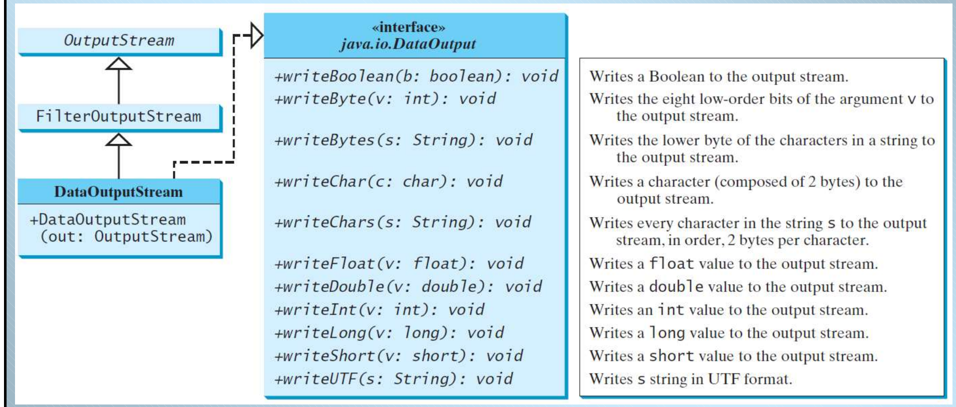


DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

25

DataOutputStream

DataOutputStream extends FilterOutputStream and implements the DataOutput interface.

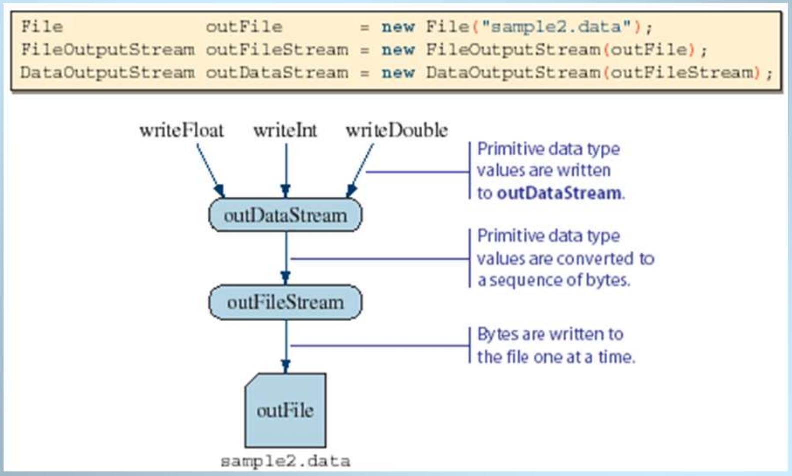


+writeBoolean(b: boolean): void	Writes a Boolean to the output stream.
+writeByte(v: int): void	Writes the eight low-order bits of the argument v to the output stream.
+writeBytes(s: String): void	Writes the lower byte of the characters in a string to the output stream.
+writeChar(c: char): void	Writes 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.

26

Setting up DataOutputStream

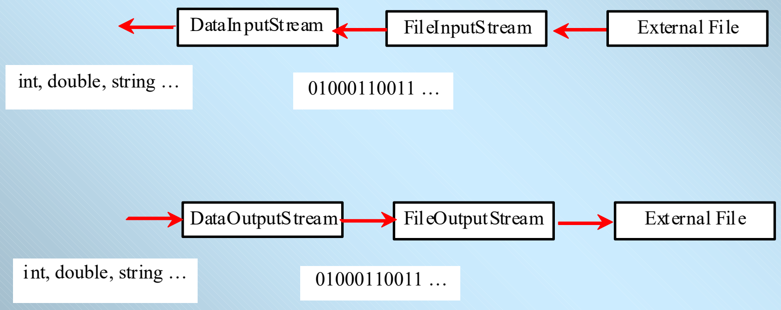
- **A standard sequence to set up a DataOutputStream object:**



```
File             outFile       = new File("sample2.data");
FileOutputStream outFileStream = new FileOutputStream(outFile);
DataOutputStream outDataStream = new DataOutputStream(outFileStream);
```

writeFloat
writeInt
writeDouble
Primitive data type values are written to **outDataStream**.
outDataStream
Primitive data type values are converted to a sequence of bytes.
outFileStream
Bytes are written to the file one at a time.
outFile
sample2.data

Concept of pipe line



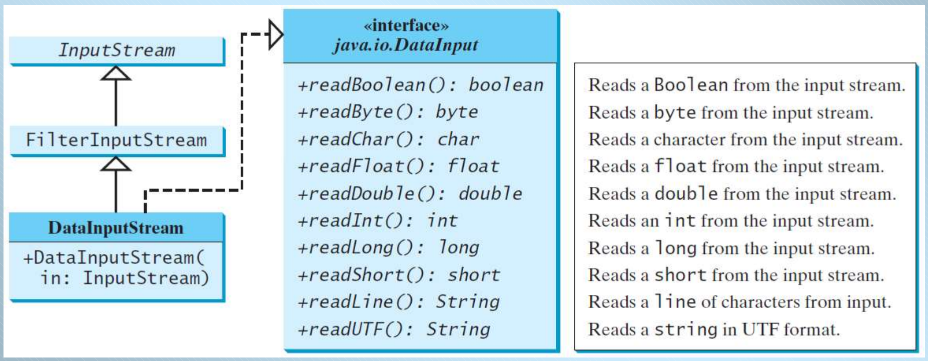
28

Sample: Data-level Binary File Output

```
import java.io.*;
class Ch12TestDataOutputStream {
    public static void main (String[] args) throws IOException {

        . . . //set up outDataStream

        //write values of primitive data types to the stream
        outDataStream.writeInt(987654321);
        outDataStream.writeLong(11111111L);
        outDataStream.writeFloat(22222222F);
        outDataStream.writeDouble(3333333.0);
        outDataStream.writeChar('A');
        outDataStream.writeBoolean(true);
        outDataStream.writeUTF("Hello");

        //output done, so close the stream
        outDataStream.close();
    }
}
```

DataInputStream

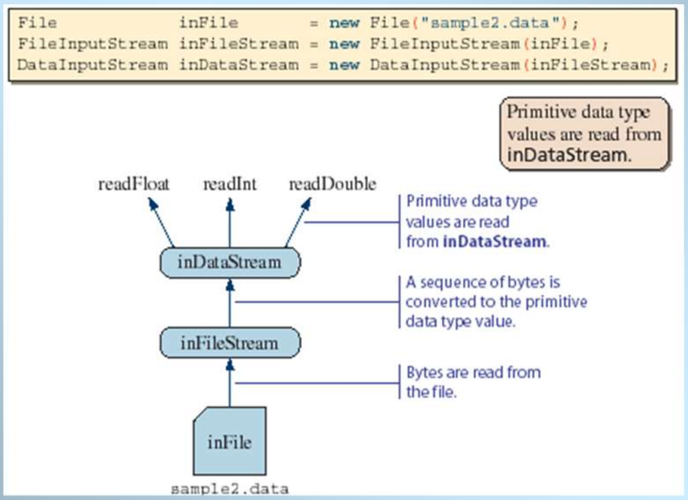
DataInputStream extends FilterInputStream and implements the DataInput interface.


InputStream
FilterInputStream
DataInputStream
+DataInputStream(in: InputStream)
«interface» java.io.DataInput
+readBoolean(): boolean
+readByte(): byte
+readChar(): char
+readFloat(): float
+readDouble(): double
+readInt(): int
+readLong(): long
+readShort(): short
+readLine(): String
+readUTF(): String
Reads a Boolean from the input stream.
Reads a byte from the input stream.
Reads a character from the input stream.
Reads a float from the input stream.
Reads a double from the input stream.
Reads an int from the input stream.
Reads a long from the input stream.
Reads a short from the input stream.
Reads a line of characters from input.
Reads a string in UTF format.

30

Setting up DataInputStream

- **A standard sequence to set up a DataInputStream object:**



```
File            inFile       = new File("sample2.data");
FileInputStream inFileStream = new FileInputStream(inFile);
DataInputStream inDataStream = new DataInputStream(inFileStream);
```

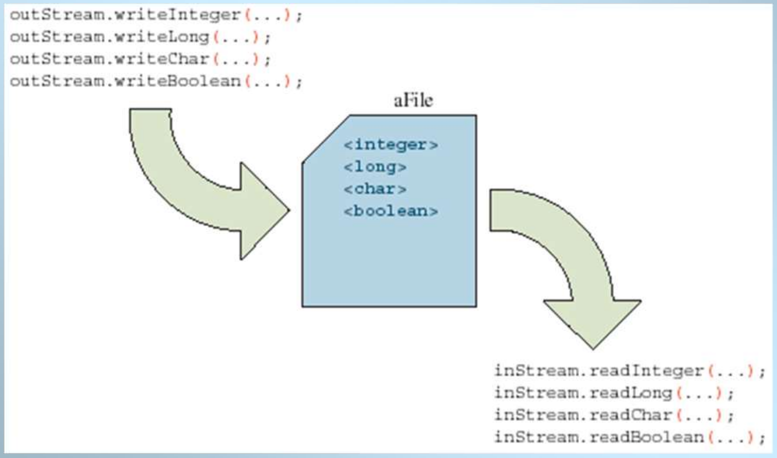
Primitive data type values are read from inDataStream.
readFloat
readInt
readDouble
inDataStream
Primitive data type values are read from inDataStream.
A sequence of bytes is converted to the primitive data type value.
inFileStream
Bytes are read from the file.
inFile
sample2.data

Sample: Data-level Binary File Input

```
import java.io.*;
class Ch12TestDataInputStream {
    public static void main (String[] args) throws IOException {

        . . . //set up inDataStream

        //read values back from the stream and display them
        System.out.println(inDataStream.readInt());
        System.out.println(inDataStream.readLong());
        System.out.println(inDataStream.readFloat());
        System.out.println(inDataStream.readDouble());
        System.out.println(inDataStream.readChar());
        System.out.println(inDataStream.readBoolean());
        System.out.println(inDataStream.readUTF());

        //input done, so close the stream
        inDataStream.close();
    }
}
```

Reading Data in Correct Order

- **The order of write and read operations must match in order to read the stored primitive data back correctly.**



```
outStream.writeInteger(...);
outStream.writeLong(...);
outStream.writeChar(...);
outStream.writeBoolean(...);
```

```
inStream.readInteger(...);
inStream.readLong(...);
inStream.readChar(...);
inStream.readBoolean(...);
```

Reading & Writing Text File

- Instead of storing primitive data values as binary data in a file, we can store them as a string data.
  - This allows us to view the file content using any text editor
- To write data as a string to text file, use FileWriter and PrintWriter object
- To read data from textfile, use FileReader and BufferedReader classes
  - From Java 5.0 (SDK 1.5), we can also use the Scanner class for reading textfiles

Writing to Text File

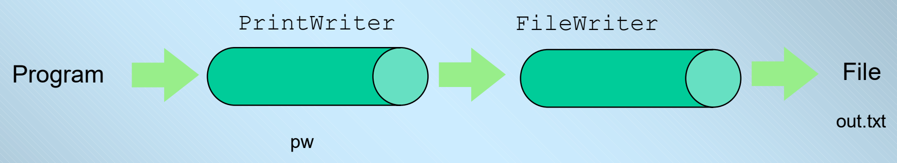
- To open a text file for output: create a `PrintWriter` object which uses `FileWriter` to open a text file
- `FileWriter` connects `PrintWriter` to a text file:

```
FileWriter fw = new FileWriter("out.txt");
PrintWriter pw = new PrintWriter(fw);
```

- Similar shorter way:

```
PrintWriter pw =
   new PrintWriter(new FileWriter("out.txt"));
```

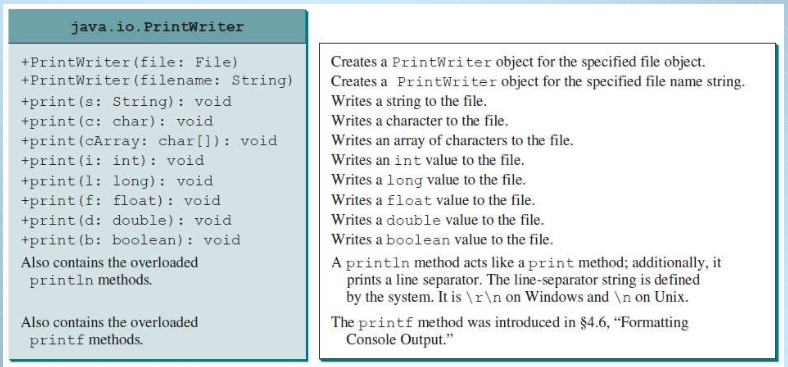
Output File Streams



PrintWriter pw = new PrintWriter(new FileOutputStream("out.txt"));

PrintWriter

The print, println, and printf methods on the PrintWriter object can be called to write text data to a file.



java.io.PrintWriter	
+PrintWriter(file: File)	Creates a PrintWriter object for the specified file object.
+PrintWriter(filename: String)	Creates a PrintWriter object for the specified file name string.
+print(s: String): void	Writes a string to the file.
+print(c: char): void	Writes a character to the file.
+print(cArray: char[]): void	Writes an array of characters to the file.
+print(i: int): void	Writes an int value to the file.
+print(l: long): void	Writes a long value to the file.
+print(f: float): void	Writes a float value to the file.
+print(d: double): void	Writes a double value to the file.
+print(b: boolean): void	Writes a boolean value to the file.
Also contains the overloaded println methods.	A println method acts like a print method; additionally, it prints a line separator. The line-separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §4.6, "Formatting Console Output."

The PrintWriter class contains the methods for writing data to a text file.

37

Sample Writing to Textfile

```
import java.io.*;
class TestPrintWriter {
    public static void main (String[] args) throws IOException {

        //set up file and stream
        File outFile = new File("myfile.txt");
        FileWriter outFileStream
                       = new FileWriter(outFile);
        PrintWriter outStream = new PrintWriter(outFileStream);

        //write values of primitive data types to the stream
        outStream.println(987654321);
        outStream.println("Hello!");
        outStream.println(true);

        //output done, so close the stream
        outStream.close();
    }
}
```

Appending to a Text File

- To add/append to a file instead of replacing it, use a different constructor for **`FileWriter`**:

```
FileWriter outfile = new FileWriter("out.txt", true);
PrintWriter outStream = new PrintWriter(outfile);
```

- Second parameter of **`FileWriter`** constructor is set to true - append to the end of the file

Reading Text File

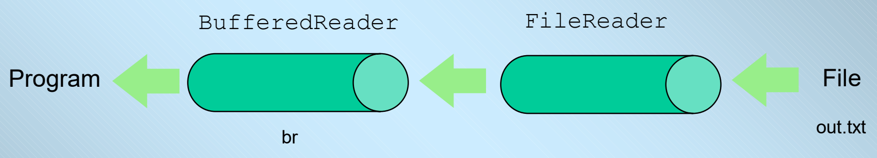
- To open a text file for input: create a `BufferedReader` object which uses `FileReader` to open a text file
- `FileReader` connects `BufferedReader` to a text file:

```
FileReader fr = new FileReader(“out.txt");
BufferedReader br = new BufferedReader(fr);
```

- Similar shorter way:

```
BufferedReader br =
      new BufferedReader(new FileReader(“out.txt”));
```

Input File Streams



BufferedReader br = new BufferedReader(new FileReader(“out.txt”));

Methods for `BufferedReader`

- `readLine()`: read a line as a `String`
  - `Returns null if at the end of file`
- no methods to read numbers directly, so read numbers as `String` and then convert them
- `read`: read a `char` at a time
- `close`: close `BufferedReader` stream

Sample: Reading from Textfile

```
import java.io.*;
class TestBufferedReader {

    public static void main (String[] args) throws IOException {
        //set up file and stream
        File inFile = new File(“myfile.txt");
        FileReader fileReader = new FileReader(inFile);
        BufferedReader bufReader = new BufferedReader(fileReader);
        String str;

        str = bufReader.readLine(); //read the first line
        int i = Integer.parseInt(str);
         str = bufReader.readLine(); //read the 2nd line
        str = bufReader.readLine(); //read the 3rd line
         boolean b = Boolean.parseBoolean();

        bufReader.close();
    }
}
```

Sample: Reading all lines from Text file

```
BufferedReader inFile = …
name = inFile.readline();
while (name != null)
{
    id = inFile.readline();
    balance = inFile.readLine();
    // …  new Account(name, id, balance);

    name = inFile.readline();

}
```

Alternative with Scanner

- Instead of `BufferedReader` with `FileReader`, use `Scanner` with `File`:

```
Scanner inFile =
  new Scanner(new File("in.txt"));
```

- Similar to `Scanner` with `System.in`:

```
Scanner keyboard =
  new Scanner(System.in);
```

Sample Reading Text file using Scanner

```
import java.io.*;

class TestScanner {

    public static void main (String[] args) throws IOException {

        //open the Scanner
         File inFile = new File("myfile.txt");
         Scanner scanner = new Scanner(inFile);

        //Read all data
        int i = scanner.nextInt();
        String msg = scanner.next();
        boolean b = scanner.nextBoolean();

        scanner.close();
    }
}
```

Sample Reading Text file using Scanner

Code fragments shows how to read the whole contents of a file

Use `hasNextLine() & nextLine()` methods

```
try{
      // read line one by one till all line is read.
     Scanner scanner = new Scanner(inFile);
     while (scanner.hasNextLine()) {  //check if there are more line
           String line = scanner.nextLine();
           System.out.println(line);
     }
} catch (Exception e) {
            e.printStackTrace();
}
```

Multiple types on one line

```
// Name, id, balance
Scanner inFile = new Scanner(new File("in.txt"));
while (inFile.hasNext())
  {
    name = inFile.next();
    id = inFile.nextInt();
    balance = inFile.nextFloat();
    // …  new Account(name, id, balance);
  }
------------------------------------------------------------
String line;
while (inFile.hasNextLine())
  {
    line = inFile.nextLine();
    Scanner parseLine = new Scanner(line) // Scanner again!
    name = parseLine.next();
    id = parseLine.nextInt();
    balance = parseLine.nextFloat();
    // …  new Account(name, id, balance);
  }
```