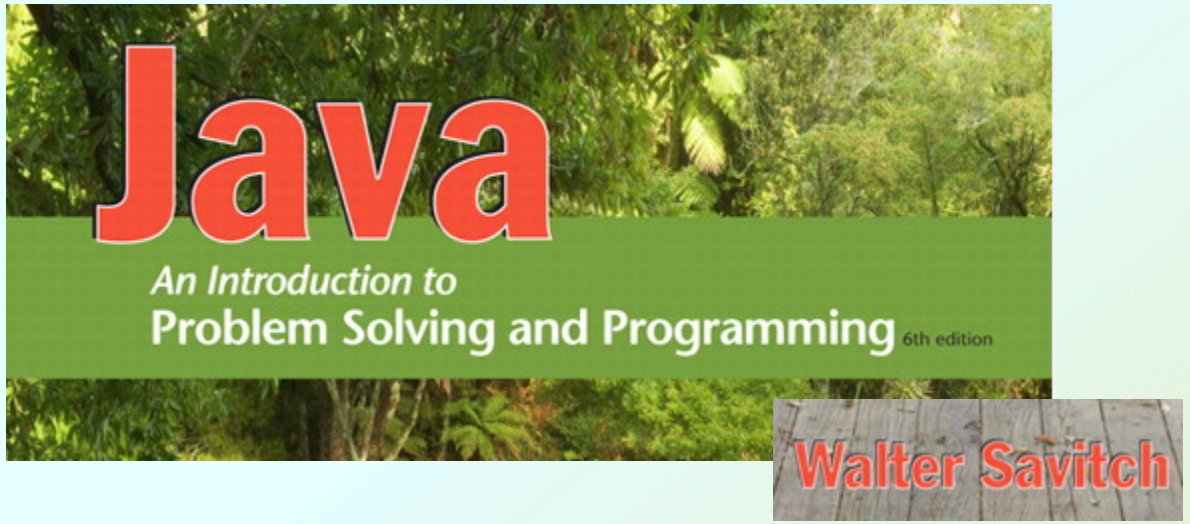



# Streams and File I/O

## Chapter 10

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709 

# Objectives

- Describe the concept of an I/O stream
- Explain the difference between text and binary files
- Save data, including objects, in a file
- Read data, including objects, in a file

# Overview: Outline

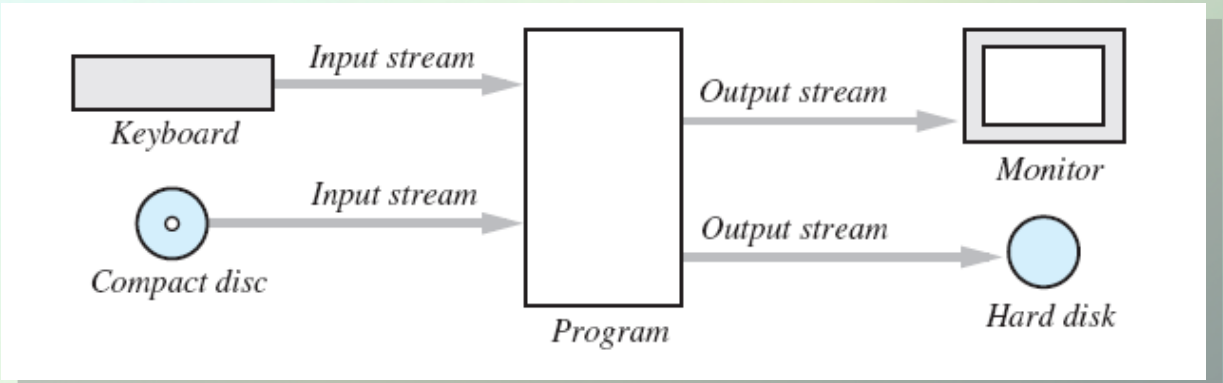
- The Concept of  a Stream
- Why Use Files for I/O?
- Text Files and Binary Files

# The Concept of a Stream

- Use of files
  - Store Java classes, programs
  - Store pictures, music, videos
  - Can also use files to store program I/O
- A *stream* is a flow of input or output data
  - Characters
  - Numbers
  - Bytes

# The Concept of a Stream

- Streams are implemented as objects of special stream classes
  - Class **`Scanner`**
  - Object **`System.out`**
- Figure 10.1 I/O Streams



# Why Use Files for I/O

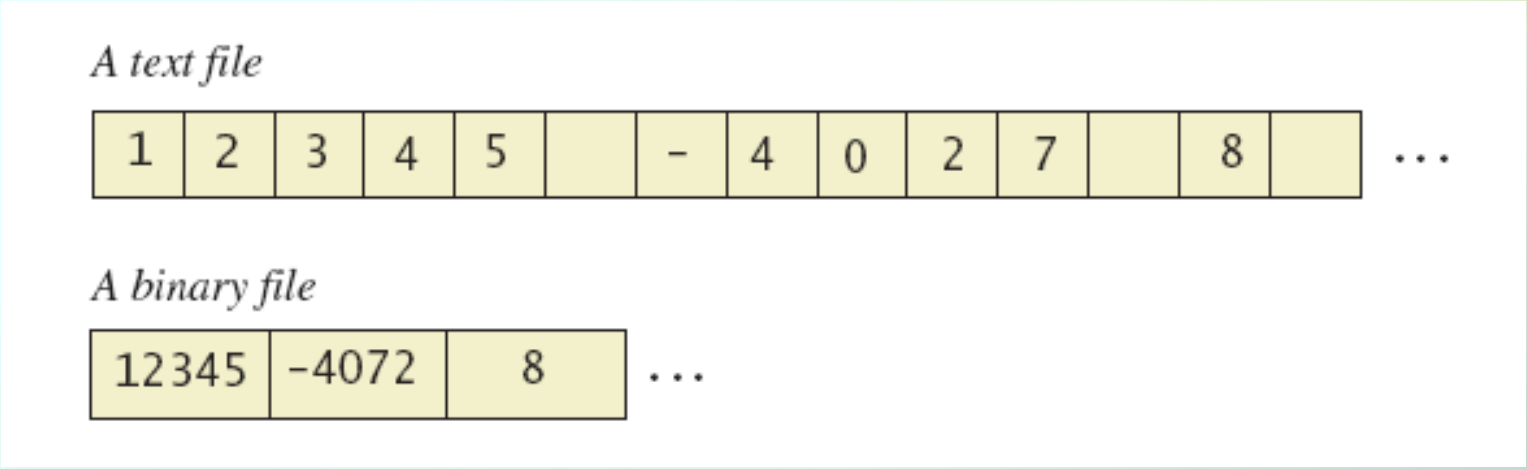
- Keyboard input, screen output deal with temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program

# Text Files and Binary Files

- All data in files stored as binary digits
  - Long series of zeros and ones
- Files treated as sequence of characters called *text files*
  - Java program source code
  - Can be viewed, edited with text editor
- All other files are called *binary files*
  - Movie, music files
  - Access requires specialized program

# Text Files and Binary Files

- Figure 10.2 A text file and a binary file containing the same values



# Text-File I/O: Outlline

- Creating a Text File
- Appending to a text File
- Reading from a Text File

# Creating a Text File

- Class **`PrintWriter`** defines methods needed to create and write to a text file
  - Must import package **`java.io`**
- To open the file
  - Declare *stream variable* for referencing the stream
  - Invoke **`PrintWriter`** constructor, pass file name as argument
  - Requires **`try`** and **`catch`** blocks

# Creating a Text File

- File is empty initially
  - May now be written to with method `println`
- Data goes initially to memory buffer
  - When buffer full, goes to file
- Closing file empties buffer, disconnects from stream

# Creating a Text File

- View sample program, listing 10.1

```
class TextFileOutput
```

```
Enter three lines of text:
A tall tree
in a short forest is like
a big fish in a small pond.
Those lines were written to out.txt
```

Sample screen output

**Resulting File**

```
1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

*You can use a text editor to read this file.*

# Creating a Text File

- When creating a file
  - Inform the user of ongoing I/O events, program should not be "silent"
- A file has two names in the program
  - File name used by the operating system
  - The stream name variable
- Opening, writing to file overwrites pre-existing file in directory

# Appending to a Text File

- Opening a file new begins with an empty file
  - If already exists, will be overwritten
- Some situations require appending data to existing file
- Command could be

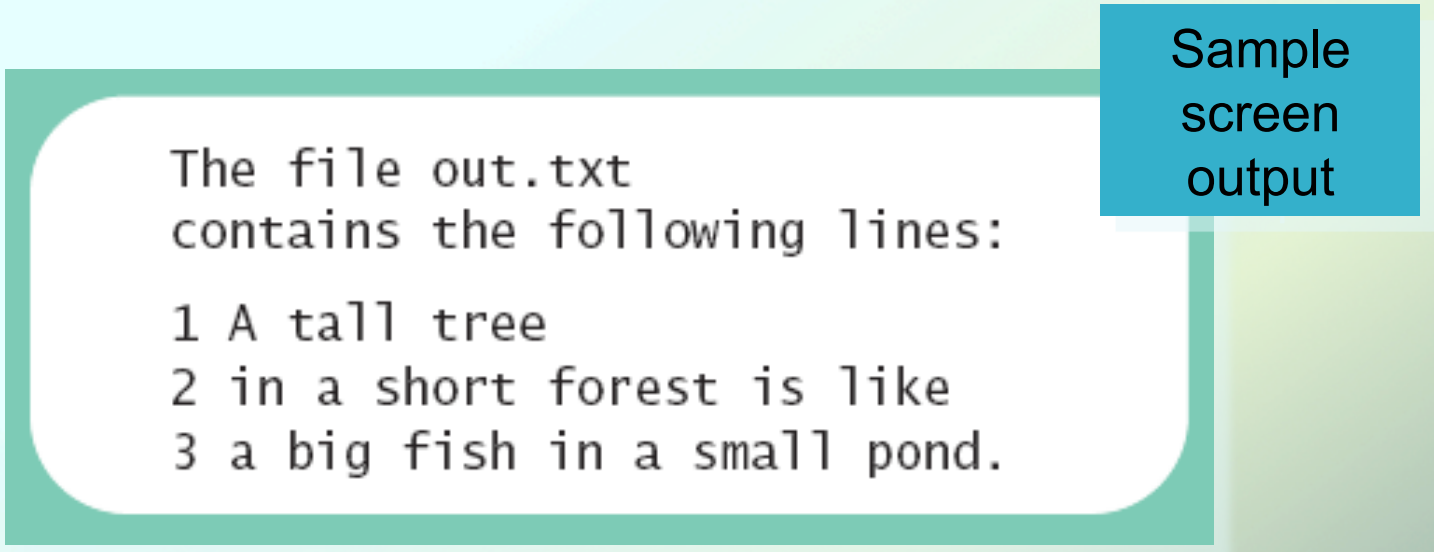
```
outputStream =
    new PrintWriter(
    new FileOutputstream(fileName, true));
```

- Method `println` would append data at end

# Reading from a Text File

- Note text file reading program, listing 10.2
  **`class TextFileInputDemo`**
- Reads text from file, displays on screen
- Note
  - Statement which opens the file
  - Use of **`Scanner`** object
  - Boolean statement which reads the file and terminates reading loop

# Reading from a Text File



# Reading from a Text File

- Figure 10.3 Additional methods in class **`Scanner`**

| |
|---|
| *Scannner_Object_Name*`.hasNext()`<br>Returns true if more input data is available to be read by the method `next`. |
| *Scannner_Object_Name*`.hasNextDouble()`<br>Returns true if more input data is available to be read by the method `nextDouble`. |
| *Scannner_Object_Name*`.hasNextInt()`<br>Returns true if more input data is available to be read by the method `nextInt`. |
| *Scannner_Object_Name*`.hasNextLine()`<br>Returns true if more input data is available to be read by the method `nextLine`. |

# Techniques for Any File

- The Class `File`
- Programming Example: Reading a File Name from the Keyboard
- Using Path Names
- Methods of the Class `File`
- Defining a Method to Open a Stream

# The Class `File`

- Class provides a way to represent file names in a general way
  - A `File` object represents the name of a file
- The object

  ```
  new File ("treasure.txt")
  ```

  is not simply a string
  - It is an object that *knows* it is supposed to name a file

# Programming Example

- Reading a file name from the keyboard
- View sample code, listing 10.3
  **`class TextFileInputDemo2`**

```
Enter file name: out.txt
The file out.txt
contains the following lines:

1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

Sample screen output

# Using Path Names

- Files opened in our examples assumed to be in same folder as where program run
- Possible to specify path names
  - Full path name
  - Relative path name
- Be aware of differences of pathname styles in different operating systems

# Methods of the Class File

- Recall that a **`File`** object is a system-independent abstraction of file's path name
- Class **`File`** has methods to access information about a path and the files in it
  - Whether the file exists
  - Whether it is specified as readable or not
  - Etc.

# Methods of the Class File

- Figure 10.4 Some methods in class **`File`**

| |
|---|
| `public boolean canRead()`<br>Tests whether the program can read from the file. |
| `public boolean canWrite()`<br>Tests whether the program can write to the file. |
| `public boolean delete()`<br>Tries to delete the file. Returns true if it was able to delete the file. |
| `public boolean exists()`<br>Tests whether an existing file has the name used as an argument to the constructor when the `File` object was created. |
| `public String getName()`<br>Returns the name of the file. (Note that this name is not a path name, just a simple file name.) |
| `public String getPath()`<br>Returns the path name of the file. |
| `public long length()`<br>Returns the length of the file, in bytes. |

# Defining a Method to Open a Stream

- Method will have a `String` parameter
  - The file name
- Method will return the stream object
- Will throw exceptions
  - If file not found
  - If some other I/O problem arises
- Should be invoked inside a `try` block and have appropriate `catch` block

# Defining a Method to Open a Stream

- Example code

```
public static PrintWriter openOutputTextFile(String fileName)
                    throws FileNotFoundException, IOException
{
    PrintWriter toFile = new PrintWriter(fileName);
    return toFile;
}
```

- Example call

```
PrintWriter outputStream = null;
try
{
    outputStream = openOutputTextFile("data.txt");
}
< appropriate catch block(s) >
```

# Case Study Processing a Comma-Separated Values File

- A comma-separated values or CSV file is a simple text format used to store a list of records
- Example from log of a cash register's transactions for the day:

```
SKU,Quantity,Price,Description
4039,50,0.99,SODA
    9100,5,9.50,T-SHIRT
1949,30,110.00,JAVA PROGRAMMING TEXTBOOK
5199,25,1.50,COOKIE
```

# Example Processing a CSV File

- View program that calculates total sales, listing 10.4 **`class TransactionReader`**
- Uses the split method which puts strings separated by a delimiter into an array

```
String line = "4039,50,0.99,SODA"
String[] ary = line.split(",");
System.out.println(ary[0]);          // Outputs 4039
System.out.println(ary[1]);          // Outputs 50
System.out.println(ary[2]);          // Outputs 0.99
System.out.println(ary[3]);          // Outputs SODA
```

# Basic Binary-File I/O

- Creating a Binary File
- Writing Primitive Values to a Binary File
- Writing Strings to a Binary File
- The Class **`EOFException`**
- Programming Example: Processing a File of Binary Data

# Creating a Binary File

- Stream class **`ObjectOutputStream`** allows files which can store
  - Values of primitive types
  - Strings
  - Other objects
- View program which writes integers, listing 10.5 **`class BinaryOutputDemo`**

# Creating a Binary File

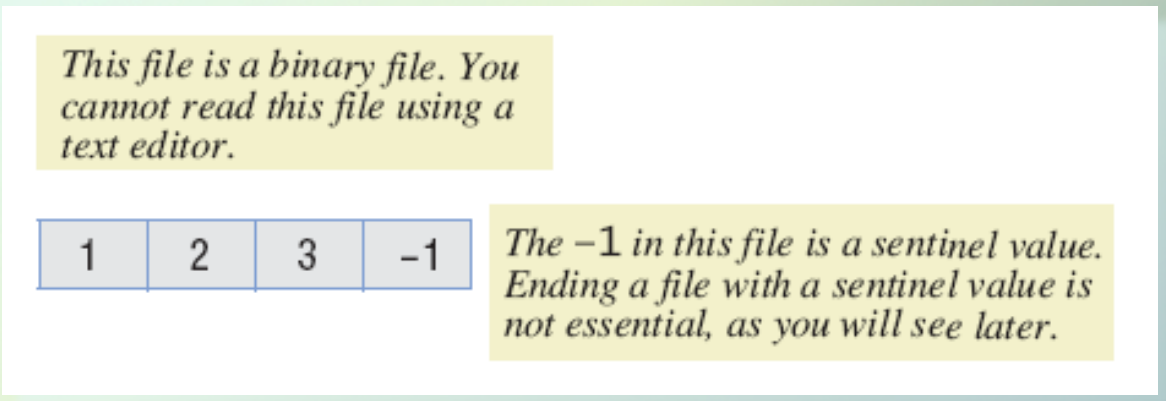
```
Enter nonnegative integers.
Place a negative number at the end.
1 2 3 -1
Numbers and sentinel value
written to the file numbers.dat.
```

Sample screen output

- Note the line to open the file
  - Constructor for **`ObjectOutputStream`** cannot take a **`String`** parameter
  - Constructor for **`FileOutputSream`** can

# Writing Primitive Values to a Binary File

- Method println not available
  - Instead use writeInt method
  - View in listing 10.5
- Binary file stores numbers in binary form
  - A sequence of bytes
  - One immediately after another



# Writing Primitive Values to a Binary File

- Figure 10.5a Some methods in class **`ObjectOutputStream`**

```
public ObjectOutputStream(OutputStream streamObject)
```

Creates an output stream that is connected to the specified binary file. There is no constructor that takes a file name as an argument. If you want to create a stream by using a file name, you write either

```
new ObjectOutputStream(new FileOutputStream(File_Name))
```

or, using an object of the class `File`,

```
new ObjectOutputStream(new FileOutputStream(
                       new File(File_Name)))
```

Either statement creates a blank file. If there already is a file named *File_Name*, the old contents of the file are lost.

The constructor for `FileOutputStream` can throw a `FileNotFoundException`. If it does not, the constructor for `ObjectOutputStream` can throw an `IOException`.

```
public void writeInt(int n) throws IOException
```

Writes the `int` value `n` to the output stream.

```
public void writeLong(long n) throws IOException
```

Writes the `long` value `n` to the output stream.

# Writing Primitive Values to a Binary File

- Figure 10.5b Some methods in class **`ObjectOutputStream`**

| |
|---|
| `public void writeDouble(double x) throws IOException`<br>Writes the `double` value `x` to the output stream. |
| `public void writeFloat(float x) throws IOException`<br>Writes the `float` value `x` to the output stream. |
| `public void writeChar(int c) throws IOException`<br>Writes a `char` value to the output stream. Note that the parameter type of `c` is `int`. However, Java will automatically convert a `char` value to an `int` value for you. So the following is an acceptable invocation of `writeChar`:<br>`outputStream.writeChar('A');` |
| `public void writeBoolean(boolean b) throws IOException`<br>Writes the `boolean` value `b` to the output stream. |
| `public void writeUTF(String aString) throws IOException`<br>Writes the string `aString` to the output stream. UTF refers to a particular method of encoding the string. To read the string back from the file, you should use the method `readUTF` of the class `ObjectInputStream`. These topics are discussed in the next section. |

# Writing Primitive Values to a Binary File

- Figure 10.5c Some methods in class **`ObjectOutputStream`**

```
public void writeObject(Object anObject) throws IOException,
                NotSerializableException, InvalidClassException
```

Writes `anObject` to the output stream. The argument should be an object of a serializable class, a concept discussed later in this chapter. Throws a `NotSerializableException` if the class of `anObject` is not serializable. Throws an `InvalidClassException` if there is something wrong with the serialization. The method `writeObject` is covered later in this chapter.

```
public void close() throws IOException
```

Closes the stream s connection to a file.

# Writing Strings to a Binary File

- Use method **`writeUTF`**
- Example

```
outputStream.writeUTF("Hi Mom");
```

- UTF stands for *Unicode Text Format*
- Uses a varying number of bytes to store different strings
  - Depends on length of string
  - Contrast to **`writeInt`** which uses same for each

# Reading from a Binary File

- File must be opened as an **`ObjectInputStream`**
- Read from binary file using methods which correspond to write methods
  - Integer written with **`writeInt`** will be read with **`readInt`**
- Be careful to read same type as was written

# Reading from a Binary File

- Figure 10.6a Some methods of class **`ObjectInputStream`**

`ObjectInputStream(InputStream streamObject)`

Creates an input stream that is connected to the specified binary file. There is no constructor that takes a file name as an argument. If you want to create a stream by using a file name, you use either

```
new ObjectInputStream(new FileInputStream(File_Name))
```

or, using an object of the class `File`,

```
new ObjectInputStream(new FileInputStream(
                      new File(File_Name)))
```

The constructor for `FileInputStream` can throw a `FileNotFoundException`. If it does not, the constructor for `ObjectInputStream` can throw an `IOException`.

`public int readInt() throws EOFException, IOException`

Reads an `int` value from the input stream and returns that `int` value. If `readInt` tries to read a value from the file that was not written by the method `writeInt` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

# Reading from a Binary File

- Figure 10.6b Some methods of class **`ObjectInputStream`**

| |
|---|
| `public long readLong() throws EOFException, IOException`<br>Reads a `long` value from the input stream and returns that `long` value. If `readLong` tries to read a value from the file that was not written by the method `writeLong` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.<br>Note that you cannot write an integer using `writeLong` and later read the same integer using `readInt`, or to write an integer using `writeInt` and later read it using `readLong`. Doing so will cause unpredictable results. |
| `public double readDouble() throws EOFException, IOException`<br>Reads a `double` value from the input stream and returns that `double` value. If `readDouble` tries to read a value from the file that was not written by the method `writeDouble` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown. |

# Reading from a Binary File

- Figure 10.6c Some methods of class **`ObjectInputStream`**

`public float readFloat() throws EOFException, IOException`

Reads a `float` value from the input stream and returns that `float` value. If `readFloat` tries to read a value from the file that was not written by the method `writeFloat` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

Note that you cannot write a floating-point number using `writeDouble` and later read the same number using `readFloat`, or write a floating-point number using `writeFloat` and later read it using `readDouble`. Doing so will cause unpredictable results, as will other type mismatches, such as writing with `writeInt` and then reading with `readFloat` or `readDouble`.

# Reading from a Binary File

- Figure 10.6d Some methods of class **`ObjectInputStream`**

| |
|---|
| `public char readChar() throws EOFException, IOException`<br>Reads a `char` value from the input stream and returns that `char` value. If `readChar` tries to read a value from the file that was not written by the method `writeChar` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown. |
| `public boolean readBoolean() throws EOFException, IOException`<br>Reads a `boolean` value from the input stream and returns that `boolean` value. If `readBoolean` tries to read a value from the file that was not written by the method `writeBoolean` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown. |

# Reading from a Binary File

- Figure 10.6e Some methods of class **`ObjectInputStream`**

```
public String readUTF() throws IOException,
                               UTFDataFormatException
```
Reads a `String` value from the input stream and returns that `String` value. If `readUTF` tries to read a value from the file that was not written by the method `writeUTF` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. One of the exceptions `UTFDataFormatException` or `IOException` can be thrown.

```
Object readObject() throws ClassNotFoundException,
    InvalidClassException, OptionalDataException, IOException
```
Reads an object from the input stream. Throws a `ClassNotFoundException` if the class of a serialized object cannot be found. Throws an `InvalidClassException` if something is wrong with the serializable class. Throws an `OptionalDataException` if a primitive data item, instead of an object, was found in the stream. Throws an `IOException` if there is some other I/O problem. The method `readObject` is covered in Section 10.5.

```
public void close() throws IOException
```
Closes the stream's connection to a file.

# Reading from a Binary File

- View program to read, listing 10.6
  **`class BinaryInputDemo`**

```
Reading the nonnegative integers
in the file numbers.dat.
1
2
3
End of reading from file.
```

Sample screen output

# The Class `EOFException`

- Many methods that read from a binary file will throw an `EOFException`
  - Can be used to test for end of file
  - Thus can end a reading loop
- View example program, listing 10.7 class `EOFExceptionDemo`

# The Class `EOFException`

- Note the -1 formerly needed as a sentinel value is now also read

```
Reading ALL the integers
in the file numbers.dat.
1
2
3
-1
End of reading from file.
```

Sample screen output

- Always a good idea to check for end of file even if you have a sentinel value

# Programming Example

- Processing a file of binary data
  - Asks user for 2 file names
  - Reads numbers in input file
  - Doubles them
  - Writes them to output file
- View processing program, listing 10.8 class **`Doubler`**

# Binary-File I/O, Objects & Arrays

- Binary-File I/O with Objects of a Class
- Some Details of Serialization
- Array Objects in Binary Files

# Binary-File I/O with Class Objects

- Consider the need to write/read objects other than **`Strings`**
  - Possible to write the individual instance variable values
  - Then reconstruct the object when file is read
- A better way is provided by Java
  - *Object serialization* – represent an object as a sequence of bytes to be written/read
  - Possible for any class implementing **`Serializable`**

# Binary-File I/O with Class Objects

- Interface **`Serializable`** is an empty interface
  - No need to implement additional methods
  - Tells Java to make the class serializable (class objects convertible to sequence of bytes)
- View sample class , listing 10.9

**`class Species`**

# Binary-File I/O with Class Objects

- Once we have a class that is specified as **`Serializable`** we can write objects to a binary file
  - Use method **`writeObject`**
- Read objects with method **`readObject();`**
  - Also required to use typecast of the object
- View sample program, listing 10.10 **`class ObjectIODemo`**

# Binary-File I/O with Class Objects

```
Records sent to file species.record.
Now let's reopen the file and echo the records.
The following were read
from the file species.record:
Name = Calif. Condor
Population = 27
Growth rate = 0.02%

Name = Black Rhino
Population = 100
Growth rate = 1.0%
End of program.
```

Sample screen output

# Some Details of Serialization

- Requirements for a class to be serializable
  - Implments interface **`Serializable`**
  - Any instance variables of a class type are also objects of a serializable class
  - Class's direct superclass (if any) is either serializable or defines a default constructor

# Some Details of Serialization

- Effects of making a class serializable
  - Affects how Java performs I/O with class objects
  - Java assigns a *serial number* to each object of the class it writes to the `ObjectOutputStream`
  - If same object written to stream multiple times, only the serial number written after first time

# Array Objects in Binary Files

- Since an array is an object, possible to use **`writeObject`** with entire array
  - Similarly use **`readObject`** to read entire array
- View array I/O program, listing 10.11 class ArrayIODemo

# Array Objects in Binary Files

```
Array written to file array.dat and file is closed.
Open the file for input and echo the array.
The following were read from the file array.dat:
Name = Calif. Condor
Population = 27
Growth rate = 0.02%

Name = Black Rhino
Population = 100
Growth rate = 1.0%

End of program.
```

Sample screen output

# Graphics Supplement

- Programming Example
- A `JFrame` GUI for Manipulating Files

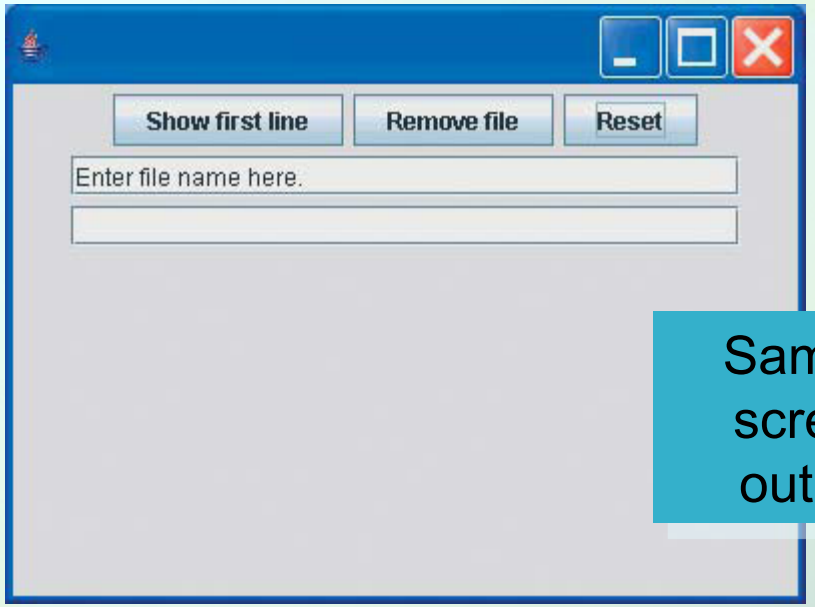
# Programming Example

- A **`JFrame`** GUI for manipulating files
- Note buttons
  - Show first line
  - Remove file
  - Reset
- Note also the text fields
  - Type in a file name
  - Display first line of file

# Programming Example

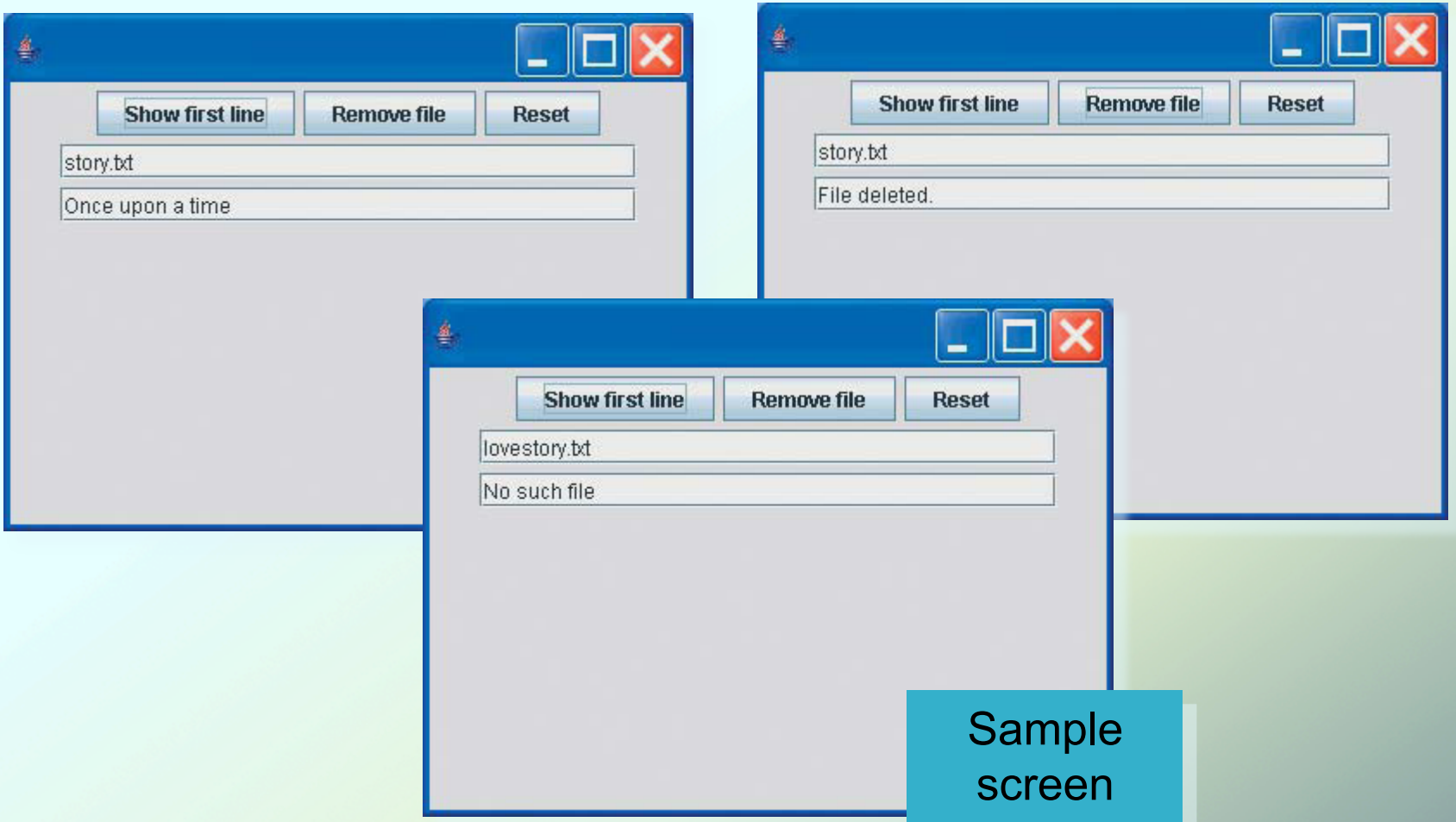
- View **`JFrame`** program, listing 10.12

  **`class FileOrganizer`**




Sample screen output

# Programming Example




Sample screen output

# Programming Example

- Note we did this with a **`JFrame`** GUI program
  - Not an applet
- For security reasons applets are limited in what they can do
  - Designed to be embedded in a Web page, run from another computer
  - Thus applets cannot manipulate files on a remote computer

# Summary

- Files with characters are text files
  - Other files are binary files
- Programs can use **`PrintWriter`** and **`Scanner`** for I/O
- Always check for end of file
- File name can be literal string or variable of type **`String`**
- Class **`File`** gives additional capabilities to deal with file names

# Summary

- Use **`ObjectOutputStream`** and **`ObjectInputStream`** classes enable writing to, reading from binary files
- Use **`writeObject`** to write class objects to binary file
- Use **`readObject`** with type cast to read objects from binary file
- Classes for binary I/O must be serializable