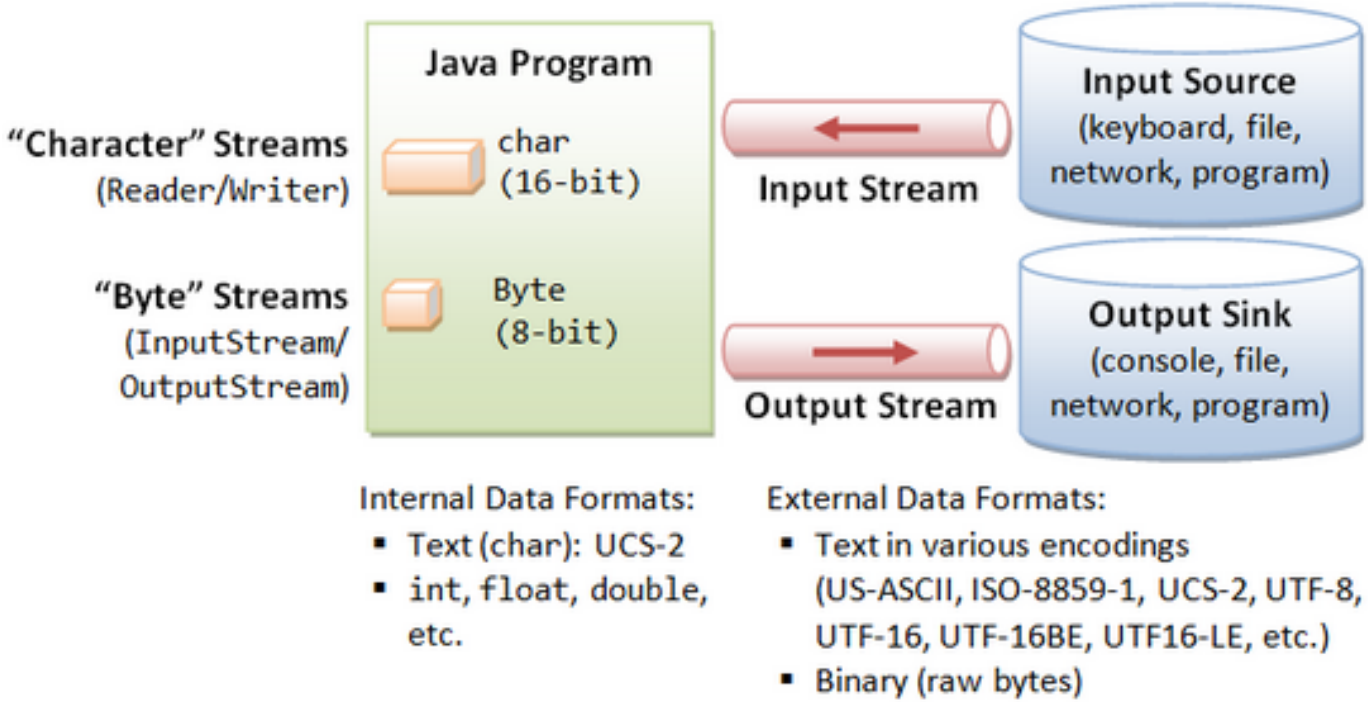


Streams and File I/O

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

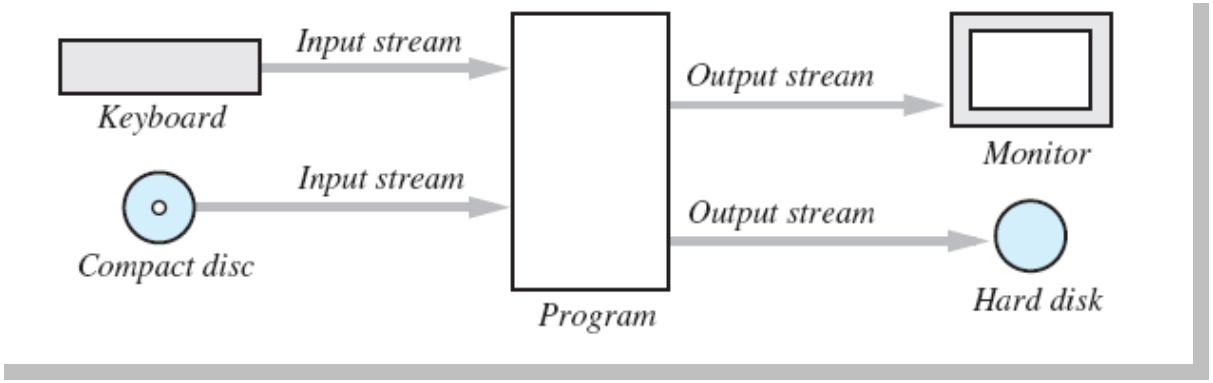
… The Concept of a Stream

- Streams are implemented as objects of special stream classes

  Class **`Scanner`**

  Object **`System.out`**

- I/O Streams



Why Use Files for I/O?

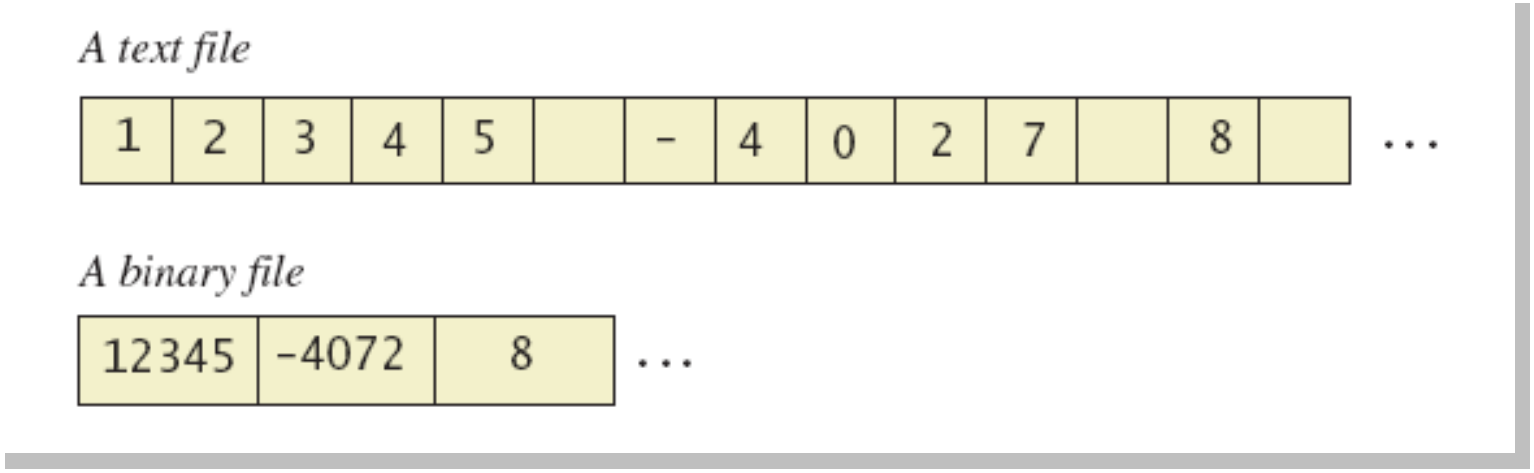
- Keyboard input, screen output deal with temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program

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 is one example
  - Can be viewed, edited with text editor
- All other files are called *binary files*
  - Movie files, music files, Java class files
  - Access requires specialized program

… Text Files and Binary Files

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



Text-File I/O: Outline

- Creating and Writing to 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 from package **`java.io`**
- To open the file
  - Declare a *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
  - Once created, it may be written to using method `println`
- Data goes initially to a “buffer” in memory
  - When the buffer is full, data goes to the file
- Closing a file empties buffer, disconnects from stream

… Creating a Text File …

- View sample program, listing **`class TextFileOutputDemo`**

Sample screen output

```
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
```

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 `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 …

```
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

… Reading from a Text File

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

Scannner_Object_Name`.hasNext()` Returns true if more input data is available to be read by the method `next`.
Scannner_Object_Name`.hasNextDouble()` Returns true if more input data is available to be read by the method `nextDouble`.
Scannner_Object_Name`.hasNextInt()` Returns true if more input data is available to be read by the method `nextInt`.
Scannner_Object_Name`.hasNextLine()` 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,
  `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

class TextFileInputDemo2

```java
//TextfileInputDemo2.java

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class TextFileInputDemo2
{
    public static void main(String[] args)
    {
        System.out.print("Enter file name: ");
        Scanner keyboard = new Scanner(System.in);
        String fileName = keyboard.next();
        Scanner inputStream = null;
        System.out.println("The file " + fileName +
            " contains the following lines:");
        try
        {
            inputStream = new Scanner(new File(fileName));
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Error opening the file " + fileName + ".");
            System.exit(0);
        }
        while (inputStream.hasNextLine())
        {
            String line = inputStream.nextLine();
            System.out.println(line);
        }
        inputStream.close();
    }
}
```

Class File

```
public abstract class Files {

    private static ArrayList<String> lines;

    /**
     * Removes a local file.
     *
     * @param fileName the file's name
     * @return true if the file has been removed, false otherwise
     */
    public static boolean delete(final String fileName) {
        final File file = new File(fileName);
        final boolean cond = file.delete();
        return cond;
    }

    /**
     * Renames a local file.
     *
     * @param fileName1 the name of the file to be renamed
     * @param fileName2 the new file name
     * @return true if the file has been renamed, false otherwise
     */
    public static boolean rename(final String fileName1, final String fileName2) {
        final File file1 = new File(fileName1);
        final File file2 = new File(fileName2);
        final boolean cond = file1.renameTo(file2);
        return cond;
    }
```

Class File

```java
/**
 * Counts how many lines a text file has.
 *
 * @param fileName the file's name
 * @return the number of lines of the file
 */
private static int countLines(final String fileName) {
    BufferedReader br = null;
    int nl = 0;
    try {
        final File fitxerText = new File(fileName);
        final FileReader fileReader = new FileReader(fitxerText);
        br = new BufferedReader(fileReader);
        while (br.readLine() != null) {
            nl++;
        }
    } catch (final FileNotFoundException ex) {
        System.err.println("S'ha produit una FileNotFoundException: " + ex.getMessage());
    } catch (final IOException ex) {
        System.err.println("S'ha produit una IOException: " + ex.getMessage());
    } finally {
        try {
            if (br != null) {
                br.close();
            }
        } catch (final IOException ex) {
            System.err.println("S'ha produit una IOException: " + ex.getMessage());
        }
    }
    return nl;
}
```

Class File

```java
/**
 * Reads all lines of a local CSV file and stores them in the field arrayLines.
 *
 * @param fileName the file's name
 */
private static void readFile(final String fileName) {
    BufferedReader br = null;
    String readLine = "";
    int nl = countLines(fileName);
    lines = new ArrayList<String>();
    try {
        final File textFile = new File(fileName);
        final FileReader fileReader = new FileReader(textFile);
        br = new BufferedReader(fileReader);
        for (int i = 0; i < nl; i++) {
            readLine = br.readLine();
            lines.add(readLine);
        }
    } catch (final FileNotFoundException ex) {
        System.err.println("S'ha produit una FileNotFoundException: " + ex.getMessage());
    } catch (final IOException ex) {
        System.err.println("S'ha produit una IOException: " + ex.getMessage());
    } finally {
        try {
            if (br != null) {
                br.close();
            }
        } catch (final IOException ex) {
            System.err.println("S'ha produit una IOException: " + ex.getMessage());
        }
    }
}
```

Class File

```java
/**
 * Sorts a CSV file by one of its fields.
 *
 * @param sourceFileName the source file's name
 * @param targetFileName the target file's name
 * @param fn the number of the field we want to order by
 * @param ft the type of the field (1 int, 2 double, 3 String)
 * @param fs the field separator
 * @return time wasted to order the file, in ms
 */
public static long createOrderedCsvFile(String sourceFileName, String targetFileName, int fn,
        int ft, String fs) {
    final long t1 = System.currentTimeMillis();
    PrintWriter w = null;
    readFile(sourceFileName);
    try {
        final File targetFile = new File(targetFileName);
        final FileWriter fileWriter = new FileWriter(targetFile);
        final BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
        w = new PrintWriter(bufferedWriter);
        if (ft == 1 || ft == 2) {
            // Build a number container
            ArrayList<Double> numbers = new ArrayList<Double>();
            // For each element of the lines, save in numbers the corresponding numeric value
            for (String lin : lines) {
                numbers.add(Double.parseDouble(lin.split(fs)[fn]));
            }
            // Sort numbers
            Collections.sort(numbers);
            // For each element in numbers find the corresponding value in lines
            for (int i = 0; i < numbers.size(); i++) {
                for (int j = 0; j < lines.size(); j++) {
                    if (Double.parseDouble(lines.get(j).split(fs)[fn]) == numbers.get(i)) {
                        w.print(lines.get(j) + System.getProperty("line.separator"));
```

Class File

```
            for (String lin : lines) {
                fields.add(lin.split(fs)[fn]);
            }
            // Sort fields
            Collections.sort(fields);
            // For each element of fields, find the correponding value in lines
            for (int i = 0; i < fields.size(); i++) {
                for (int j = 0; j < lines.size(); j++) {
                    if (lines.get(j).split(fs)[fn].equals(fields.get(i))) {
                        w.print(lines.get(j) + System.getProperty("line.separator"));
                        lines.remove(j);
                    }
                }
            }
        }
        w.close();
    } catch (final FileNotFoundException ex) {
        System.err.println("S'ha produit una FileNotFoundException: " + ex.getMessage());
    } catch (final IOException ex) {
        System.err.println("S'ha produit una IOException: " + ex.getMessage());
    } finally {
        try {
            if (w != null) {
                w.close();
            }
        } catch (final Exception ex) {
            System.err.println("S'ha produit una Exception: " + ex.getMessage());
        }
    }
    final long t2 = System.currentTimeMillis();
    return t2 - t1;
}
```

Class File

```java
/**
 * Finds out if two files are equals, character by character.
 *
 * @param fileName1 the file's name of one of the files
 * @param fileName2 the file's name of the other file
 * @return true if they are equals, false otherwise
 */
public static boolean areEquals(final String fileName1, final String fileName2) {
    final Reader r1 = new Reader(fileName1);
    final Reader r2 = new Reader(fileName2);
    int c1 = r1.read();
    int c2 = r2.read();
    boolean found = false;
    while ((c1 != -1 && c2 != -1) && !found) {
        found = c1 != c2 ? true : false;
        c1 = r1.read();
        c2 = r2.read();
    }
    if (!found) {
        found = (c1 == -1 ^ c2 == -1) ? true : false;
    }
    r1.close();
    r2.close();
    return !found;
}
}
```

Class Reader

```java
public class Reader {

    private BufferedReader br = null;

    /**
     * Constructor
     */
    public Reader() {
        br = new BufferedReader(new InputStreamReader(System.in));
    }

    /**
     * Constructor.
     *
     * @param fileName the file's name.
     */
    public Reader(final String fileName) {
        try {
            br = new BufferedReader(new FileReader(fileName));
        } catch (FileNotFoundException ex) {
            Logger.getLogger(Reader.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    /**
```

Class Reader

```java
/**
 * Reads a line from the keyboard or from the text file.
 *
 * @return a string with the read line without the ending \n
 */
public String readLine() {
    String line = null;
    try {
        line = br.readLine();
    } catch (IOException ex) {
        Logger.getLogger(Reader.class.getName()).log(Level.SEVERE, null, ex);
    }
    return line;
}

/**
 * Reads a character from the keyboard or from the text file.
 *
 * @return the int value of the read character or -1 if reading another
 *         character is not possible.
 */
public int read() {
    int c = -1;
    try {
        c = br.read();
    } catch (IOException ex) {
        Logger.getLogger(Reader.class.getName()).log(Level.SEVERE, null, ex);
    }
    return c;
}
```

Class Reader

```
    /**
     * Closes the buffer.
     */
    public void close() {
        try {
            br.close();
        } catch (IOException ex) {
            Logger.getLogger(Reader.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Class Writer

```java
public class Writer {

    private PrintWriter pw = null;

    /**
     * Constructor.
     */
    public Writer() {
        pw = new PrintWriter(System.out);
    }

    /**
     * Constructor.
     */
    public Writer(final String fileName) {
        try {
            pw = new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
        } catch (IOException ex) {
            Logger.getLogger(Writer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

Class Writer

```java
/**
 * Writes a char in the text file.
 *
 * @param c a character
 */
public void print(final char c) {
    pw.print(c);
}

/**
 * Writes a string in the text file.
 *
 * @param str a string
 */
public void print(final String str) {
    pw.print(str);
}

/**
 * Writes an integer number of type int in the text file.
 *
 * @param num an integer number
 */
public void print(final int num) {
    pw.print(num);
}
```

Class Writer

```java
/**
 * Writes an integer number of type int in the text file.
 *
 * @param num an integer number
 */
public void print(final int num) {
    pw.print(num);
}

/**
 * Writes a real number of type double in the text file.
 *
 * @param num a real number
 */
public void print(final double num) {
    pw.print(num);
}

/**
 * Writes a string plus a break line in the text file.
 *
 * @param str a string
 */
public void println(final String str) {
    pw.println(str);
}
```

Class Writer

```java
    /**
     * Writes an integer number of type int plus a break line in the text file.
     *
     * @param num an integer number
     */
    public void println(final int num) {
        pw.println(num);
    }

    /**
     * Writes a real number of type double plus a break line in the text file.
     *
     * @param num a real number
     */
    public void println(final double num) {
        pw.println(num);
    }

    /**
     * Closes the buffer.
     */
    public void close() {
        pw.close();
    }
}
```