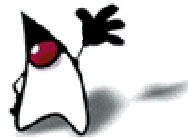
Part I

General Principles



Streams

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data (sometimes both)
- An input stream may be associated with the keyboard
- An input stream or an output stream may be associated with a file
- Different streams have different characteristics:
  - A file has a definite length, and therefore an end
  - Keyboard input has no specific end

How to do I/O

```
import java.io.*;
```

- *Open* the stream
- *Use* the stream (read, write, or both)
- *Close* the stream

Why Java I/O is hard

open
use
close

- Java I/O is very powerful, with an overwhelming number of options
- Any given kind of I/O is not particularly difficult
- The trick is to find your way through the maze of possibilities

Opening a stream

open
use
close

- There is data external to your program that you want to get, or you want to put data somewhere outside your program
- When you open a stream, you are making a connection to that external place
- Once the connection is made, you forget about the external place and just use the stream

Example of opening a stream

open
use
close

- A FileReader is a used to connect to a file that will be used for input:

```
FileReader fileReader =
    new FileReader(fileName);
```

- The fileName specifies where the (external) file is to be found
- You never use fileName again; instead, you use fileReader

Using a stream

open
use
close

- Some streams can be used only for input, others only for output, still others for both
- *Using* a stream means doing input from it or output to it
- But it's not usually that simple--you need to manipulate the data in some way as it comes in or goes out

Example of using a stream

open
use
close

```
int charAsInt;
charAsInt = fileReader.read( );
```

- The fileReader.read() method reads one character and returns it as an integer, or -1 if there are no more characters to read
- The meaning of the integer depends on the file encoding (ASCII, Unicode, other)
- You can *cast* from int to char:

```
char ch = (char)fileReader.read( );
```

Manipulating the input data

- Reading characters as integers isn't usually what you want to do
- A BufferedReader will convert integers to characters; it can also read whole lines
- The constructor for BufferedReader takes a FileReader parameter:

```
BufferedReader bufferedReader =
    new BufferedReader(fileReader);
```

Reading lines

open
use
close

```
String s;
s = bufferedReader.readLine( );
```

- A BufferedReader will return null if there is nothing more to read

Closing

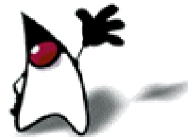
open
use
close

- A stream is an expensive resource
- There is a limit on the number of streams that you can have open at one time
- You should not have more than one stream open on the same file
- You must close a stream before you can open it again
- *Always close your streams!*

- Java will normally close your streams for you when your program ends, but it isn't good style to depend on this

Simple Java I/O

Part II
LineReader and LineWriter



Text files

- Text (`.txt`) files are the simplest kind of files
  - Text files can be used by many different programs
- Formatted text files (such as `.doc` files) also contain binary formatting information
- Only programs that "know the secret code" can make sense of formatted text files
- Compilers, in general, work only with text

My LineReader class

```
class LineReader {
  BufferedReader bufferedReader;

  LineReader(String fileName) {...}

  String readLine( ) {...}

  void close( ) {...}
}
```

Basics of the LineReader constructor

- Create a FileReader for the named file:

```
FileReader fileReader =
    new FileReader(fileName);
```

- Use it as input to a BufferedReader:

```
BufferedReader bufferedReader =
    new BufferedReader(fileReader);
```

- Use the BufferedReader; but first, we need to catch possible Exceptions

The full LineReader constructor

```
LineReader(String fileName) {
   FileReader fileReader = null;
   try { fileReader = new FileReader(fileName); }
   catch (FileNotFoundException e) {
     System.err.println
        ("LineReader can’t find input file: " + fileName);
     e.printStackTrace( );
   }
   bufferedReader = new BufferedReader(fileReader);
 }
```

readLine

```
String readLine( ) {
    try {
       return bufferedReader.readLine( );
    }
    catch(IOException e) {
        e.printStackTrace( );
    }
    return null;
}
```

close

```
void close() {
    try {
        bufferedReader.close( );
    }
    catch(IOException e) { }
}
```

How did I figure that out?

- I wanted to read lines from a file
- I thought there might be a suitable read*Something* method, so I went to the API Index
  - Note: Capital letters are all alphabetized *before* lowercase in the Index
- I found a readLine method in several classes; the most promising was the BufferedReader class
- The constructor for BufferedReader takes a Reader as an argument
- Reader is an abstract class, but it has several implementations, including InputStreamReader
- FileReader is a subclass of InputStreamReader
- There is a constructor for FileReader that takes as its argument a (String) file name

The LineWriter class

```
class LineWriter {
    PrintWriter printWriter;

    LineWriter(String fileName) {...}

   void writeLine(String line) {...}

   void close( ) {...}
}
```

The constructor for LineWriter

```
LineWriter(String fileName) {
    try {
        printWriter =
            new PrintWriter(
                new FileOutputStream(fileName), true);
    }
    catch(Exception e) {
        System.err.println("LineWriter can’t " +
            "use output file: " + fileName);
    }
}
```

Flushing the buffer

- When you put information into a buffered output stream, it goes into a buffer
- The buffer may *or may not* be written out right away
- If your program crashes, you may not know how far it got before it crashed
- Flushing the buffer forces the information to be written out

PrintWriter

- Buffers are automatically flushed when the program ends normally
- Usually it is your responsibility to flush buffers if the program does not end normally
- PrintWriter can do the flushing for you

```
public PrintWriter(OutputStream out,
                   boolean autoFlush)
```

writeLine

```
void writeLine(String line) {
    printWriter.println(line);
}
```

close

```
void close( ) {
    printWriter.flush( );
    try {
        printWriter.close( );
    }
    catch(Exception e) { }
}
```

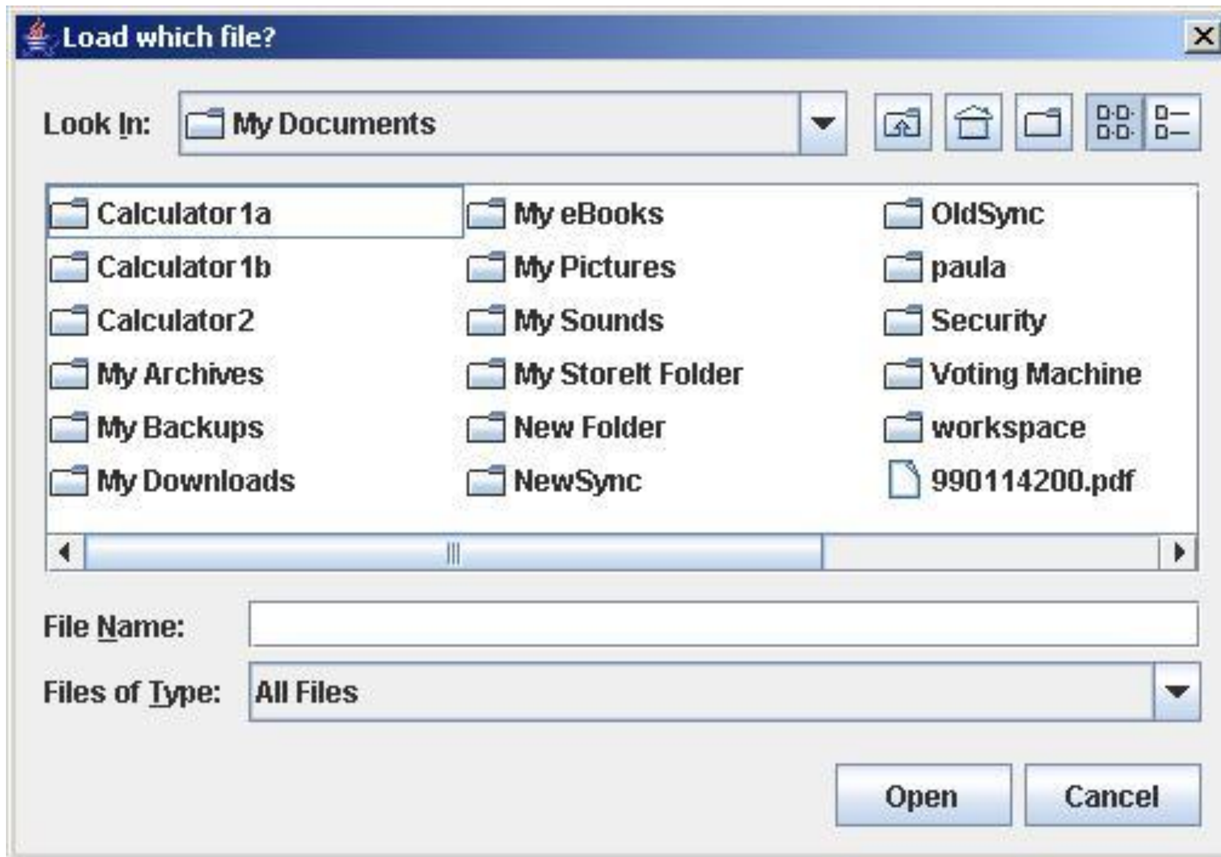
Part III

JFileChoosers

About JFileChoosers

- The JFileChooser class displays a window from which the user can select a file
- The dialog window is modal--the application cannot continue until it is closed
- Applets cannot use a JFileChooser, because applets cannot access files

Typical JFileChooser window



JFileChooser constructors

- JFileChooser()
  - Creates a JFileChooser starting from the user's directory
- JFileChooser(File *currentDirectory*)
  - Constructs a JFileChooser using the given File as the path
- JFileChooser(String *currentDirectoryPath*)
  - Constructs a JFileChooser using the given path

Useful JFileChooser methods I

- int showOpenDialog(Component ***enclosingJFrame***);
  - Asks for a file to read; returns a flag (see below)
- int showSaveDialog(Component ***enclosingJFrame***);
  - Asks where to save a file; returns a flag (see below)
- Returned flag value may be:
  - JFileChooser.APPROVE_OPTION
  - JFileChooser.CANCEL_OPTION
  - JFileChooser.ERROR_OPTION

Useful JFileChooser methods II

- File getSelectedFile()
  - showOpenDialog and showSaveDialog return a flag telling what happened, but don't return the selected file
  - After we return from one of these methods, we have to ask the JFileChooser what file was selected
  - If we are saving a file, the File may not actually exist yet—that's OK, we still have a File *object* we can use

Using a File

- Assuming that we have successfully selected a File:

  - ```
    File file = chooser.getSelectedFile();
    if (file != null) {
        String fileName = file.getCanonicalPath();
        FileReader fileReader = new FileReader(fileName);
        BufferedReader reader = new BufferedReader(fileReader);
     }
    ```

  - ```
    File file = chooser.getSelectedFile();
    if (file != null) {
        String fileName = file.getCanonicalPath();
        FileOutputStream stream = new FileOutputStream(fileName);
        writer = new PrintWriter(stream, true);
    }
    ```

The End

"There is no reason anyone would want a computer in their home."

--Ken Olson,
President/founder of Digital Equipment Corp.,
1977

"I think there is a world market for maybe five computers."

--Thomas Watson
Chairman of IBM,|
1943