CS1092: Object-Oriented Programming with Java
Text Input and Output with Files

Howard Barringer

Room KB2.20/22: email: howard.barringer@manchester.ac.uk

February 2005

Supporting and Background Material

- Copies of key slides (already handed out)
- Chapter 18 of the JTL book (follows this presentation) — lecture notes
- Recommended reading books for course
  - Chapter 9 (9.2, 9.2) of Savitch "JAVA: An introduction to Computer Science and Programming", 3rd Edition
  - Chapter 17 (17.2, 17.3, 17.4, 17.7) of Liang "Introduction to JAVA Programming", International Edition

Outline

Reading/Writing from/to Standard Input/Output

Basic input and output occurs via streams

The `System` class provides three instance variables

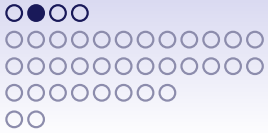
- `in`: a byte stream representing *standard input*
- `out`: a byte stream (actually, a `PrintStream` object) representing *standard output*
- `err`: a byte stream (also a `PrintStream` ojbect) representing *standard error*

Textual input and output is character-based.

Conversion between `bytes` and `characters` is required.

For output, `System.out.println` and `System.out.print` does this for you.

For input, the user must do this.

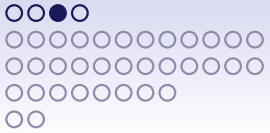


InputStreamReader

The Reader classes provides character based reading.
The InputStreamReader class provides a bridge from byte streams to character streams.

The read method will read a character from the input and return its integer value, and -1 if the end of file is reached.

```
import java.io.InputStreamReader;
import java.io.IOException;

public class Copy {
  public static void main(String args[]) throws IOException {
    InputStreamReader input = new InputStreamReader(System.in);
    int ch;
    while ((ch = input.read()) != -1)
      System.out.print((char)ch);
  } }
```



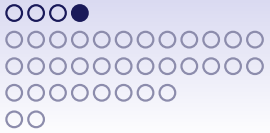
The `IOException` class

The `IOException` class is the root class for exceptions related to input and output.

There are 22 direct subclasses.

The subclass `FileNotFoundException` is one that may be more familiar.

`IOExceptions` must either be caught or declared (unlike `RuntimeExceptions`.



BufferedReader

The `InputStreamReader` class doesn't provide a method for reading a whole line of text.

The `BufferedReader` class provides such a `readLine` method.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class BufferedCopy {
  public static void main(String args[]) throws IOException {
    BufferedReader bufferedInput =
        new BufferedReader(new InputStreamReader(System.in));
    String line;
    while ((line = bufferedInput.readLine()) != null)
      System.out.println(line);
  } }
```

The Line Numbering Task

Write a program `LineNumber` that add line numbers to the text of an input file, given as the first command line argument, and write the resulting numbered lines to the file given as second argument.

If the 1st argument is `"-"`, then the program should read from standard input.

Similarly, if the 2nd argument is `"-"`, the program should write to standard output.

The filenames given as arguments should be assumed to be distinct.

Reading from a Text File

We use `Reader` subclasses for textual input from files
The class `FileReader` is used to set up a reader on a file.

We wrap a buffered reader around in order to provide a `readLine` method.

What else does the buffering do?

Writing to a Text File

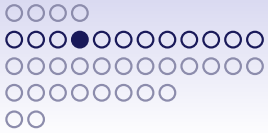
We use `Writer` subclasses for textual output to files.

To create a writer on a file

1. create a `FileWriter` object

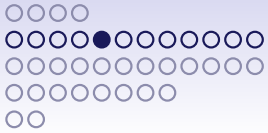
then, in order to use `print` and `println`

2. create a `PrintWriter` object from the `FileWriter` object



Almost the `main` program method

```
public static void main(String [] args) {
  try {
    BufferedReader inReader =
        new BufferedReader( new FileReader( args[0] ));
    PrintWriter outWriter =
        new PrintWriter( new FileWriter( args[1] ));

    int lineNumber = 0;
    String inputString;
    while ( (inputString = inReader.readLine()) != null ) {
      lineNumber++;
      outWriter.println(lineNumberPrefix(lineNumber, 4) + ": "
                        + inputString); } // while
    inReader.close(); outWriter.close();} //try
  catch (Exception e) {System.err.println(e);}
}// main
```

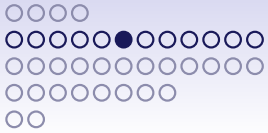


Checking the arguments - I

We must modify the above to take care of erroneous arguments.

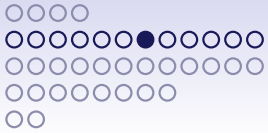
Replace the `inReader` and `outWriter` declarations by:

```
if (args.length < 1)
  throw new LineNumberFilesException("No input file argument");
BufferedReader inReader =
        new BufferedReader( new FileReader( args[0] ));
if (args.length < 2)
  throw new LineNumberFilesException("No output file argument");
PrintWriter outWriter =
        new PrintWriter( new FileWriter( args[1] ));
if (args.length > 2)
  throw new LineNumberFilesException("Too many arguments supplied");
```



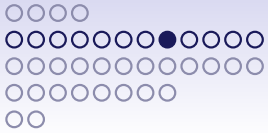
The LineNumberFilesException class

```
class LineNumberFilesException extends Exception
{
  LineNumberFilesException(){
    super();
  }

  LineNumberFilesException(String message){
    super(message);
  }

}
```



And just for completion ...

```
public static String lineNumberPrefix(int lineNumber, int width)
{
  String leadingZeroes = "";

  for (int i = 0; i < width; i++) leadingZeroes += "0";

  leadingZeroes += lineNumber;

  return leadingZeroes.substring(leadingZeroes.length() - width);
}
```



Checking Properties of Files

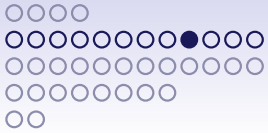
Suppose files `fred.txt` and `joe.txt` both exist. Then:

```
$ java LineNumberFiles fred.txt joe.txt

$
```

will overwrite the contents on `joe.txt` with a line numbered copy of `fred.txt`, thus destroying the original file `joe.txt`!!

This may be rather UNDESIRABLE

How do we check to see whether the file exists?



The `File` class

We create a `File` object associated with the given name.

The `exists` method of the `File` class returns true if the file actually exists.

```
if (args.length < 2)
  throw new LineNumberFilesException("No output file argument");
File outFile = new File( args[1] );
if (outFile.exists())
  throw new LineNumberFilesException("Output file " + args[1]
                                     + " already exists");
PrintWriter outWriter =
    new PrintWriter( new FileWriter( args[1] ));
```

Many other file object properties ...

Some other useful methods of the `File` class are:-

method	**effect**
`canRead()`	true if `File` object exists and is readable
`canWrite()`	true if `File` object exists and is writable
`isFile()`	true if `File` object represents a file
`isDirectory()`	true if `File` object represents a directory
`length()`	returns the size of the file in bytes
⋮	⋮

Admitting standard input

```
BufferedReader inReader;
if (args[0].equals("-"))
  inReader = new BufferedReader
                    ( new InputStreamReader( System.in ));
else
  inReader = new BufferedReader( new FileReader( args[0] ));
```

Admitting standard output

```
PrintWriter outWriter;
if (args[1].equals("-"))
  outWriter = new PrintWriter( System.out );
else {
  File outFile = new File( args[1] );
  if (outFile.exists())
    throw new LineNumberFilesException
              ("Output file " + args[1] + " already exists");
  outWriter = new PrintWriter( new FileWriter( args[1] ));
}
```

A Text Translator

Line numbering is a specialised form of general text translation.

Think of text translation as string translation applied on a line by line basis.

For line numbering, the string translation prefixes a line count.

Think of some other string translations ...

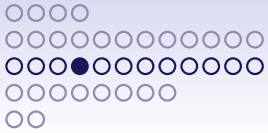
Create a generic text translator that can be instantiated with various underlying string translation "engines".

The Task ...

1. define an abstract class `StringTranslator` with one method `translate`,
2. a class `TextTranslator` with a method `translate` (for repeated application of a StringTranslator's `translate`).
3. a class `TextTranslatorProgram` that supplies the appropriate reader and writer to the `translate` method of a `TextTranslator`.
4. a non-abstract subclass of `StringTranslator` for achieving line numbering.

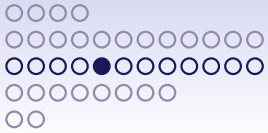
The abstract StringTranslator class

```
public abstract class StringTranslator
{

  public abstract String translate(String string);

}
```



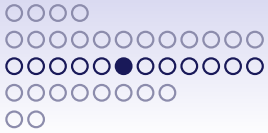
The TextTranslator class

```
public class TextTranslator
{
  private final StringTranslator stringTranslator;

  public TextTranslator(StringTranslator requiredStringTranslator)
  {
    stringTranslator = requiredStringTranslator;
  } // TextTranslator


  public void translate(BufferedReader input, PrintWriter output)
              throws IOException, TextTranslatorException
  {
    ...
  }
}
```



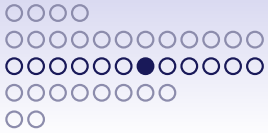
TextTranslatorException

`TextTranslatorException` is a user-defined extension of the `Exception` class.



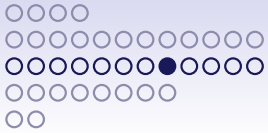
The TextTranslator translate method

```
public void translate(BufferedReader input, PrintWriter output)
            throws IOException, TextTranslatorException
{
  if (input == null)
    throw new TextTranslatorException("Input is null");
  else if (output == null)
    throw new TextTranslatorException("Output is null");

  String line;
  while ((line = input.readLine()) != null)
  {
    String translatedLine = stringTranslator.translate(line);
    if (translatedLine != null)
      output.println(translatedLine);
  }
}
```



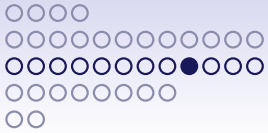
The TextTranslatorProgram class

```
public class TextTranslatorProgram {

  private final TextTranslator textTranslator;

  public TextTranslatorProgram
           (StringTranslator requiredStringTranslator)
  {
    textTranslator = new TextTranslator(requiredStringTranslator);
  }

  public void translate(String [] args)
              throws IOException, ArgumentException,
                     TextTranslatorException
  {
    ...
  }
}
```



ArgumentException

`ArgumentException` is a user-defined extension of the `Exception` class.



The TextTranslatorProgram's translate method

```
public void translate(String [] args)
    throws IOException, ArgumentException, TextTranslatorException
{
  BufferedReader input;  PrintWriter output;

  if (args.length < 1 || args[0].equals("-"))
    input = new BufferedReader(new InputStreamReader(System.in));
  else input = new BufferedReader(new FileReader(args[0]));
  if (args.length < 2 || args[1].equals("-"))
    output = new PrintWriter(System.out, true);
  else output = new PrintWriter(new FileWriter(args[1]));
  if (args.length > 2)
    throw new ArgumentException("Too many arguments");

  textTranslator.translate(input, output);

  input.close(); output.close();
}
```

The CountingStringTranslator class - I

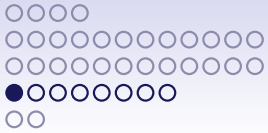
```
public class CountingStringTranslator extends StringTranslator
{
  private int lineCount = 0;
  private int lineNumberDigits;
  private String leadingZeroes;

  public CountingStringTranslator(int requiredLineNumberDigits)
  {
    lineNumberDigits = requiredLineNumberDigits;
    leadingZeroes = "";
    for (int count =1; count <= lineNumberDigits; count++)
      leadingZeroes += "0";
  }
```

The CountingStringTranslator class - II

```
  public String translate(String line)
  {
    lineCount++;
    String lineNumber = leadingZeroes + lineCount;
    lineNumber
      = lineNumber.substring(lineNumber.length() - lineNumberDigits);
    return lineNumber + " " + line;
  }
}
```

Putting it to the test ... AddLineNumber class

```
public class AddLineNumber
{
  public static void main(String [] args)
  {
    try
    {
      new TextTranslatorProgram(new CountingStringTranslator(3))
            .translate(args);
    }
    catch (Exception exception)
    {
      System.err.println(exception);
    }
  }
}
```

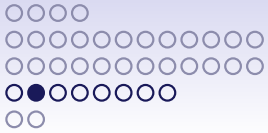


Another string translator

Construct a `StringTranslator` subclass, `StringMatcher` that

1. passes all strings that contain a given string
2. yields `null` for strings that don't match the given string

The `StringMatcher` constructor is passed the string to be matched.

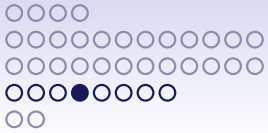


The StringMatcher class

```
public class StringMatcher extends StringTranslator
{
  private String matchString;

  public StringMatcher( String reqdString )
  {
    matchString = reqdString;
  }

  public String translate(String string)
  {
    return (string.indexOf(matchString)==-1?null:string);
  }

}
```

And now a LineFilter Program

```
public class LineFilter {
  public static void main(String [] args){
    try {
      if ( args.length < 1)
        throw new ArgumentException
                    ("Insufficient arguments for LineFilter");
      String [] newArgs = new String [args.length-1];
      for (int i = 1; i < args.length; i++) newArgs[i-1] = args[i];

      new TextTranslatorProgram( new StringMatcher(args[0]) )
            .translate(newArgs); } // try
    catch (Exception exception)
    { System.err.println(exception); }
  }
}
```



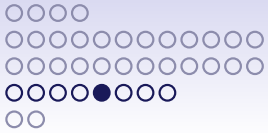
Let's try it out

Display all lines of `LineFilter.java` containing `int` ...

```
$ java LineFilter int LineFilter.java
      for (int i = 1; i < args.length; i++) newArgs[i-1] = args[i];
      System.err.println(exception);

$
```

And now for `String` ...

```
$ java LineFilter String LineFilter.java -
  public static void main(String [] args)
      String [] newArgs = new String [args.length-1];
      new TextTranslatorProgram( new StringMatcher(args[0]) )

$
```



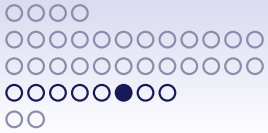
Combining String Translators

It would be useful to have line numbers (from the original file) attached to the output of `LineFilter`.

We have a `CountingStringTranslator` that attaches line numbers.

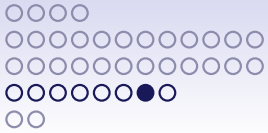
How can we combine the two, in a generic fashion?

Define a subclass of StringTranslator that composes two string translators.



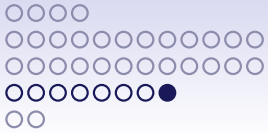
CompositeStringTranslator

```
public class CompositeStringTranslator extends StringTranslator
{
  StringTranslator first, second;

  public CompositeStringTranslator(StringTranslator reqdFirst,
                                   StringTranslator reqdSecond)
  { first = reqdFirst;
    second = reqdSecond; }

  public String translate (String inputString)
  {
    if (inputString != null) {
      String temp = first.translate(inputString);
      return (temp!=null?second.translate(temp):null); }
    else
      return null;
  }
}
```



The LineCountFilter program

```
public class LineCountFilter {
  public static void main(String [] args) {
    try {
      if ( args.length < 1)
        throw new ArgumentException
                    ("Insufficient arguments for LineCountFilter");
      String [] newArgs = new String [args.length-1];
      for (int i = 1; i < args.length; i++)
        newArgs[i-1] = args[i];
      new TextTranslatorProgram
          ( new CompositeStringTranslator(
                new CountingStringTranslator(3),
                new StringMatcher(args[0]))
          ).translate(newArgs); } // try
    catch (Exception exception)
    { System.err.println(exception);}
} }
```



Let's try this version out

Display all lines of `LineFilter.java` containing `int` ...

```
$ java LineCountFilter int LineFilter.java
010         for (int i = 1; i < args.length; i++) newArgs[i-1] = args[i];
017         System.err.println(exception);

$
```

And now for `String` ...

```
$ java LineCountFilter String LineFilter.java -
003     public static void main(String [] args)
009         String [] newArgs = new String [args.length-1];
012         new TextTranslatorProgram( new StringMatcher(args[0]) )

$
```

Summary - I

- Use the `Reader` and `Writer` classes and their subclasses
- `FileReader` and `FileWriter` to create a reader and a writer on files
- `BufferedReader` class as wrapper on a Reader to provide a `readLine` method
- `PrintWriter` class as wrapper on a Writer to provide `print` and `println`
- `PrintWriter` can also wrap a print stream, e.g. `System.out`

Summary - II

- `File` class can be used to query the properties of files
- `IOException` class is root for exceptions arising with input & output
- thrown `IOException`s must be caught or declared
- demonstrated Generalisation and the power of Re-usable classes, e.g. `TextTranslator`