List of Slides

Title

Java Just in Time

John Latham

February 11, 2019

Chapter 18

Files

Chapter aims

- Previously met **class** `Scanner` (Section **??** on page **??**)
  - used to read input **data**.
- Simple and convenient
  - at some point find out more about it
    - read **API** documentation.
- Here look at
  - reading **byte**s, **character**s, lines from **text file**s
  - writing **byte**s, **character**s, lines to **text file**s
  - reading/writing to/from **binary file**s.

Section 2

Example: Counting bytes from standard input

Aim

AIM: To introduce the principle of reading **byte**s from **standard input** using `InputStream`, meet the **try finally statement** and see that an **assignment statement** is actually an **expression** – and can be used as such *when appropriate*. We also meet `IOException` and briefly talk about initial values of **variable**s.

Counting bytes from standard input

- Program that reads **standard input**
  - reports how many **byte**s
  - how many of each value, for those that appeared at least once.
- Standard input could be redirected from a **file**
  - or from output of **run**ning program

  to see profile of bytes.

File IO API: `IOException`

- Processing **file**s – much potential for things to go wrong
  - e.g. attempt to read non-existing file
  - running out of file space while writing file
  - **operating system** experiencing disk / network problem
  - etc..
- Most operations on files capable of **throw**ing **exception**
  - `java.io.IOException`.
  - Many **subclass**es of `IOException`
    - ∗ e.g. `java.io.FileNotFoundException`.

File IO API: `IOException`

- `IOException` direct **subclass** of `java.lang.Exception`
  - not `java.lang.RuntimeException`
  - **instance**s are **checked exception**s
    - ∗ not generally avoidable by writing code
    - ∗ must write **catch clause**s
    - ∗ or **throws clause**s for them.

Counting bytes from standard input

- Read **data** from standard input
  - byte by byte.
- Use `InputStream`
  - typical use exploits fact **assignment statement** is **expression**.

Statement: assignment statement: is an expression

- Java **assignment statement** is actually **expression**
  - = is **operator**
    - ∗ takes **variable** as left **operand**
    - ∗ expression as right
    - ∗ evaluates expression
    - ∗ assigns value to variable
    - ∗ *and then* yields value as result.

Statement: assignment statement: is an expression

- So can write *horrible* code like:

```
int x = 10, y = 20, z;


int result = (z = x * y) + (y = z * 2);
```

- Example of more general idea: **side effect expressions**
  - expressions that change value of variables while **evaluate**d.
- Generally side effect expressions are dangerous
  - can lead to code difficult to understand
  - hence maintain
  - e.g. above!

Coffee time: What is the value of `result` in the example from the above concept?

Statement: assignment statement: is an expression

- However, few appropriate uses of treating assignments as expressions
  - e.g. assign same value to several variables.

```
x = y = z = 10;
```

- Unlike most operators, = has **right associativity**
  - so above is same as

```
x = (y = (z = 10));
```

- However such assignments not very common.

Statement: try statement: with finally

- The **try statement** may have **finally block**
  - piece of code **execute**d at end of statement
    - regardless of whether **try block** completes
    - or **catch clause** is executed
    - or control **throw**n out of try statement.
- General form of **try finally statement**. . .

Statement: try statement: with finally

```
try
{
  ... Code here that might cause an exception to happen.
} // try
catch (SomeException exception)
{
  ... Code here to deal with SomeException types of exception.
} // catch
catch (AnotherException exception)
{
  ... Code here to deal with AnotherException types of exception.
} // catch
... more catch clauses as required.
finally
{
  ... Code here that will be run, no matter what,
  ... as the last thing the statement does.
} // finally
```

File IO API: `InputStream`

- `java.io.InputStream`: basic building block for reading **data**
  - provides view of data as **byte stream**.
- Simplest way to access bytes one by one
  - via `read()` **instance method**.
  - Takes no **method argument**s
  - **return**s next byte from stream.
  - If/when no more bytes returns `-1`.
  - If something goes wrong **throw**s `IOException`.
- Value returned by `read()` must be able to distinguish -1 from byte value 255
  - so result is **`int`** not **`byte`**.

File IO API: `InputStream`

- Skeleton code to process all data from `InputStream`
  - another appropriate use of **assignment statement** as **expression**:
    - **loop** terminates when result of expression is certain value
    - also want to use result in loop body.
- Notice brackets around assignment statement
  - = has lower **operator precedence** than `!=`.
- . . .

File IO API: InputStream

```
InputStream inputData = null;
try
{
  inputData = ... Code to set up inputData.
  int currentByte;
  while ((currentByte = inputData.read()) != -1)
  {
    ... Code to do something with currentByte.
  } // while
} // try
catch (IOException exception)
{
  System.err.println("Ooops -- that didn’t work! " + exception.getMessage());
} // catch
finally
{
  try { if (inputData != null) inputData.close(); }
  catch (IOException exception)
    { System.err.println("Could not close input " + exception); }
} // finally
```

File IO API: `InputStream`

- Notice how used **try finally statement**
  to ensure attempt to **close** `InputStream`
  - even if something else goes wrong.
  - (Java 7.0 introduced **try with resources statement**.)
- Good idea to always close input / output streams when finished with
  - E.g. some **operating systems**
    do not separate notions of **file** name from file contents
    - file cannot be deleted / renamed
      if program has open for reading / writing.
- Also if not close output stream data might never get written to file!

Standard API: `System`: `in`: is an `InputStream`

- The **class variable** `in` inside `java.lang.System`
  - holds **reference** to **instance** of `java.io.InputStream`.
- Enables programs to access **byte**s of **standard input**.

Counting bytes from standard input

- `ByteCount` program has **array** of 256 **`int`** values
  - count occurrences of each possible byte
    - in **array element** at corresponding **array index**.
- Counts need start at zero
  - here rely on default initial values
  - rather than write **loop** to set them.

Variable: initial value

- When **class variable**s, **instance variable**s, and **array element**s created
  - given default initial value (unless also final variables).
- Whereas, **compiler** forces **local variable**s (**method variable**s) and **final variable**s to be initialized by our code.
- Dangerous to *quietly* rely on default values when happen to be desired initial values
  - anyone looking at code cannot tell difference between doing that and having forgotten to initialize!
  - Also you/they may misremember what initial value is for **variable** of particular **type**.
- So, rule of thumb: always perform own initialization.

Variable: initial value

- However where non-trivial
  - e.g. array elements
  - write clear **comment**
    - stating happy default value is desired
    - and what it is.

Counting bytes from standard input

```java
import java.io.IOException;

// Program to count the number of bytes on the standard input
// and report it on the standard output.
// Each byte that occurs at least once is listed with its own count.
public class ByteCount
{
  public static void main(String[] args)
  {
    // There are only 256 different byte values.
    // Default initial values will be zero, which is what we want.
    int[] byteCountSoFar = new int[256];

```

Counting bytes from standard input

```java
// The total number of bytes found so far.
int allBytesCountSoFar = 0;
try
{
  int currentByte;
  while ((currentByte = System.in.read()) != -1)
  {
    allBytesCountSoFar++;
    byteCountSoFar[currentByte]++;
  } // while
} // try
catch (IOException exception)
{
  System.err.println(exception);
} // catch
```

Counting bytes from standard input

```
    finally
    {
      try { System.in.close(); }
      catch (IOException exception)
        { System.err.println("Could not close input " + exception); }
    } // finally

    // Report results.
    System.out.println("The number of bytes read was " + allBytesCountSoFar);
    for (int byteValue = 0; byteValue <= 255; byteValue++)
      if (byteCountSoFar[byteValue] != 0)
        System.out.println("Byte value " + byteValue + " occurred "
                           + byteCountSoFar[byteValue] + " times");
  } // main

} // class ByteCount
```

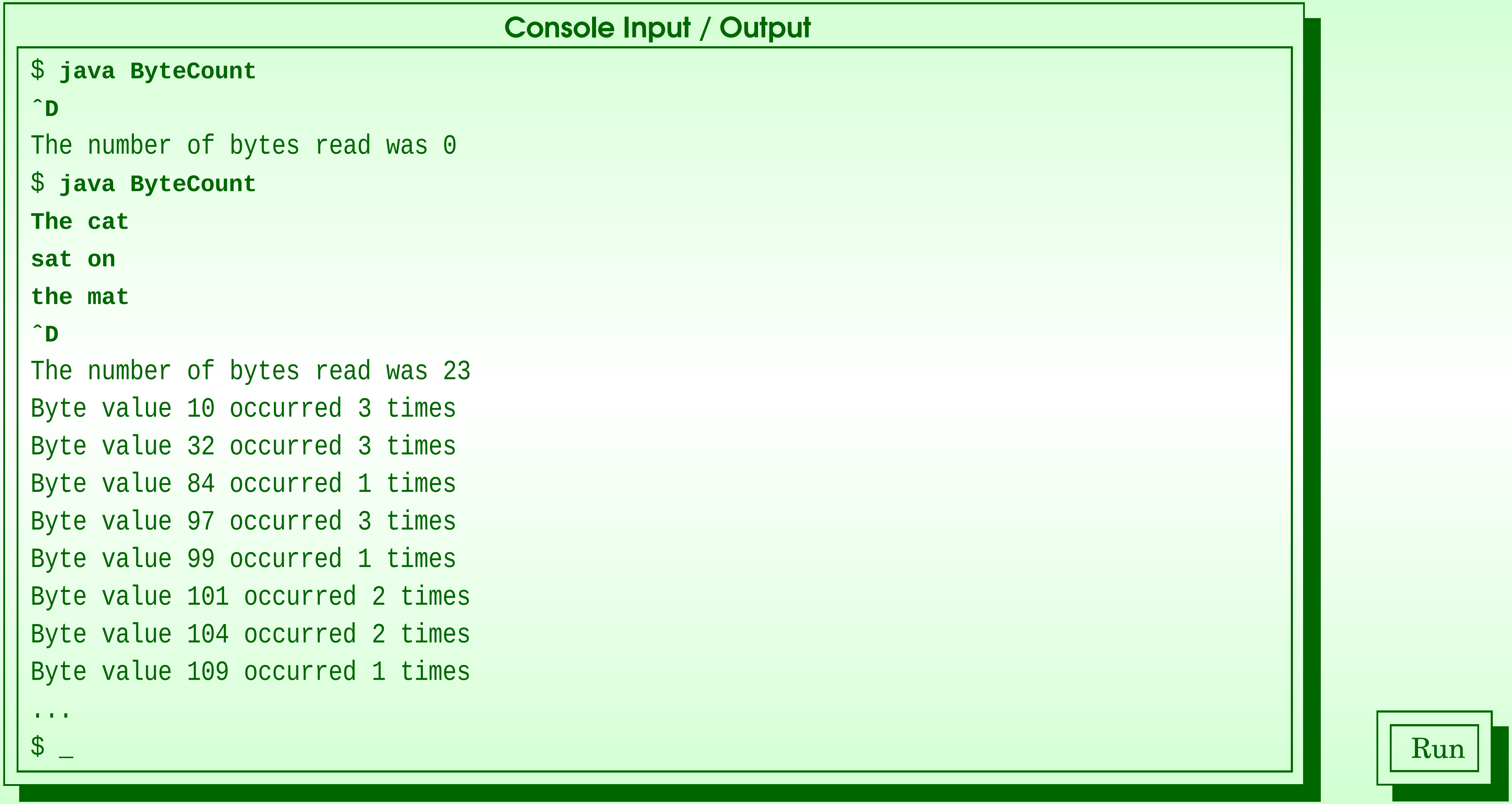
Counting bytes from standard input

- Seem odd to close `System.in`?
  - Program might be used with **standard input** redirected from **file**.
  - If not, no harm closing it,
  - If is, close means file released as soon as finished with.

Coffee time: Why did we not have to write an **import statement** for `java.io.InputStream`, even though we are using it?

Coffee time: Could we have used a **for-each loop** to print out the byte counts?

Trying it



Console Input / Output

```
$ java ByteCount
^D
The number of bytes read was 0
$ java ByteCount
The cat
sat on
the mat
^D
The number of bytes read was 23
Byte value 10 occurred 3 times
Byte value 32 occurred 3 times
Byte value 84 occurred 1 times
Byte value 97 occurred 3 times
Byte value 99 occurred 1 times
Byte value 101 occurred 2 times
Byte value 104 occurred 2 times
Byte value 109 occurred 1 times
...
$ _
```

Trying it

Coffee time: Is the above result correct? My example was being **run** under Linux. Would you expect to get the same result under Microsoft Windows? (Hint: do they have the same **line separator**?)

Coursework: A check sum program

(Summary only)

Write a program to produce a **check sum** of the **standard input**.

Section 3

Example: Counting characters from standard input

Aim

AIM: To introduce the principle of reading **characters**, instead of **bytes**, from **standard input**, using `InputStreamReader`.

Counting characters from standard input

- Good chance would want to profile **characters** of **standard input**
  - rather than **bytes**.
- Difference?
  - Depends on **locale**
    - collection of information about part of world
    - e.g. **file encoding** for characters, currency symbol, etc..
- Sometimes one character occupies one byte
  - sometimes some characters require more than one byte to represent them
  - e.g. China, Middle East,
  - possibly anywhere.

Counting characters from standard input

- For portability, treat **data** as characters
  - when we are concerned about characters

  as bytes
  - when we are concerned about bytes.
- Program here reads data from standard input,
  - character by character

File IO API: `InputStreamReader`

- An `InputStream` is sequence of **byte**s.
- When wish to treat as sequence of **character**s
  - wrap up in `java.io.InputStreamReader`.
- Provides **instance method**
  - `read`
  - **return**s next *character* from `InputStream`
    - or `-1` if no more to be read.
- Reads one or more bytes from underlying `InputStream` for each character.

File IO API: `InputStreamReader`

- Two **constructor methods**
  - one takes an `InputStream` which it wraps up
    - uses default **file encoding**.
- Other takes an `InputStream`
  - and character encoding to be used
    - permits reading character streams generated under different **locale**.

Counting characters from standard input

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

// Program to count the number of characters on the standard input
// and report it on the standard output.
// Each character that occurs at least once is listed with its own count.
public class CharacterCount
{
  public static void main(String[] args)
  {
    // There are 65536 different character values (two bytes).
    // Default initial values will be zero, which is what we want.
    int[] characterCountSoFar = new int[65536];

    // We will read the input as characters.
    InputStreamReader input = new InputStreamReader(System.in);
```

Counting characters from standard input

```java

    // The total number of characters found so far.
    int allCharactersCountSoFar = 0;
    try
    {
      int currentCharacter;
      while ((currentCharacter = input.read()) != -1)
      {
        allCharactersCountSoFar++;
        characterCountSoFar[currentCharacter]++;
      } // while
    } // try
    catch (IOException exception)
    {
      System.err.println(exception);
    } // catch
```

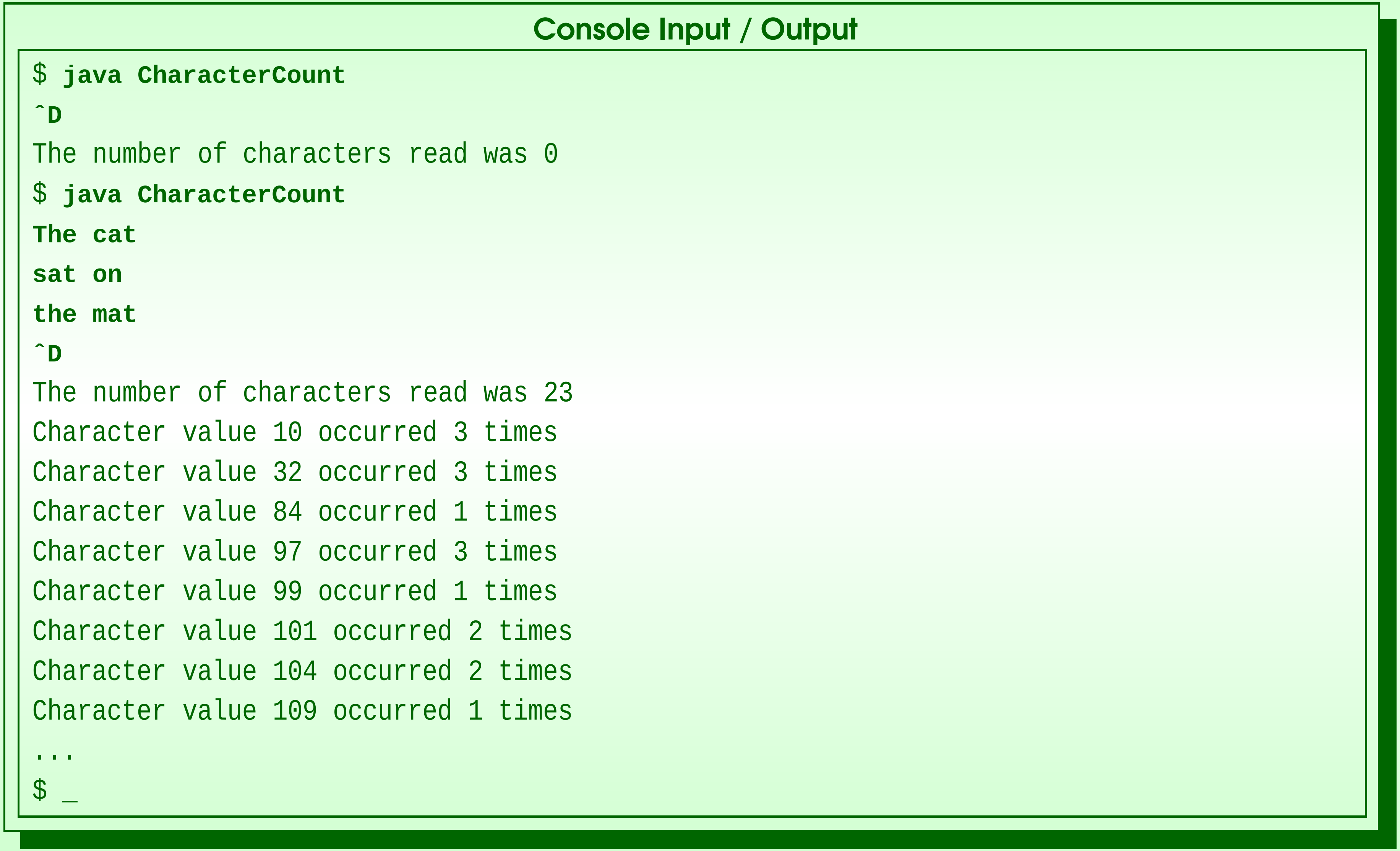
Counting characters from standard input

```
    finally
    {
      try { input.close(); }
      catch (IOException exception)
        { System.err.println("Could not close input " + exception); }
    } // finally

    // Report results.
    System.out.println("The number of characters read was "
                       + allCharactersCountSoFar);
    for (int characterValue = 0; characterValue <= 65535; characterValue++)
      if (characterCountSoFar[characterValue] != 0)
        System.out.println("Character value " + characterValue + " occurred "
                           + characterCountSoFar[characterValue] + " times");
  } // main

} // class CharacterCount
```

Trying it



```
$ java CharacterCount
^D
The number of characters read was 0
$ java CharacterCount
The cat
sat on
the mat
^D
The number of characters read was 23
Character value 10 occurred 3 times
Character value 32 occurred 3 times
Character value 84 occurred 1 times
Character value 97 occurred 3 times
Character value 99 occurred 1 times
Character value 101 occurred 2 times
Character value 104 occurred 2 times
Character value 109 occurred 1 times
...
$ _
```

Run

Trying it

Coffee time: Did the last two tests produce the same results as obtained from the `ByteCount` program in Section 17 on page 27?

- Try with popular Chinese New Year greeting
  - `HappyNewYear-GBK.txt` contains four Chinese **characters**
    - ∗ encoded in GBK encoding (still) commonly used in China

    plus **new line character**
  - total 9 **bytes**.

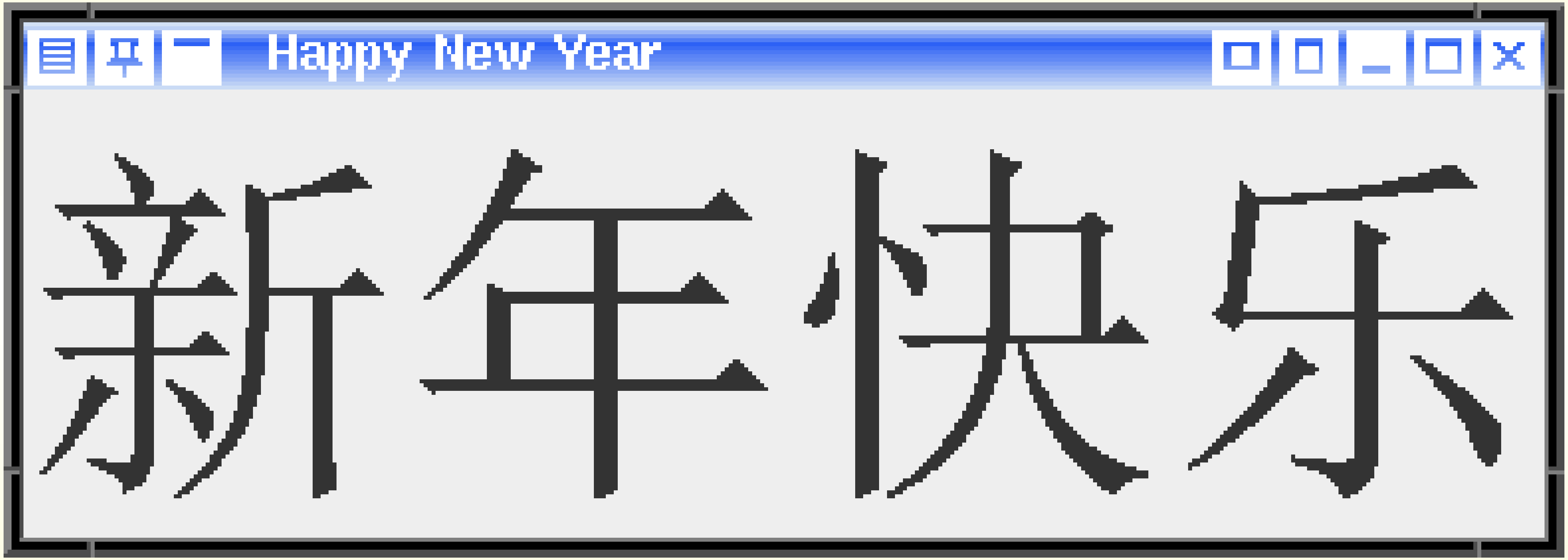
Console Input / Output

```
$ ls -l HappyNewYear-GBK.txt
-rw-------  1 jtl jtl 9 Jul 01 19:12 HappyNewYear-GBK.txt
$ _
```

Run

Trying it

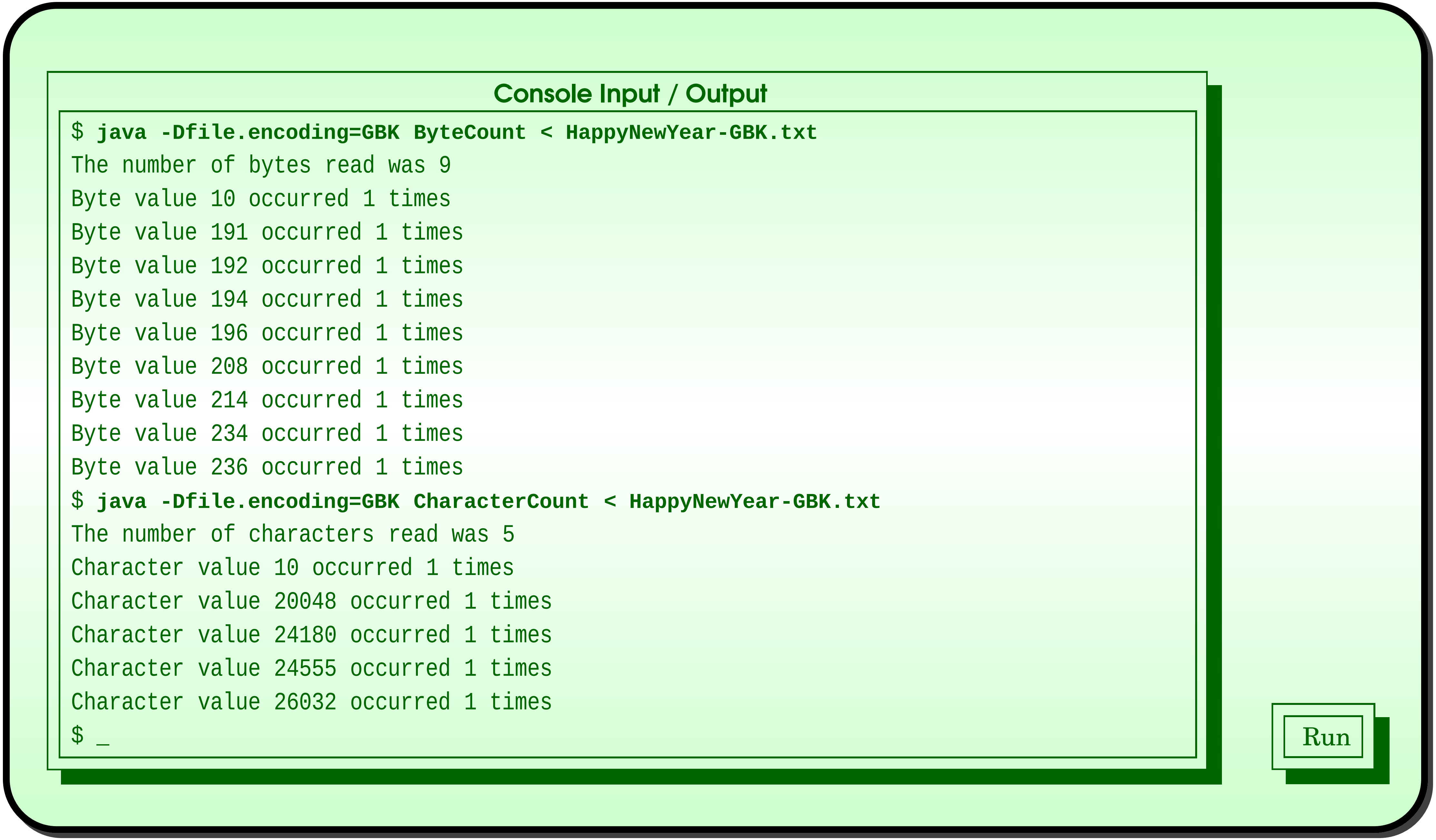
- Screen dump of GUI program displaying the four characters:



新年快乐

- Use `-Dfile.encoding=GBK` **command line argument** to set GBK encoding. . . .

Trying it



Console Input / Output

```
$ java -Dfile.encoding=GBK ByteCount < HappyNewYear-GBK.txt
The number of bytes read was 9
Byte value 10 occurred 1 times
Byte value 191 occurred 1 times
Byte value 192 occurred 1 times
Byte value 194 occurred 1 times
Byte value 196 occurred 1 times
Byte value 208 occurred 1 times
Byte value 214 occurred 1 times
Byte value 234 occurred 1 times
Byte value 236 occurred 1 times
$ java -Dfile.encoding=GBK CharacterCount < HappyNewYear-GBK.txt
The number of characters read was 5
Character value 10 occurred 1 times
Character value 20048 occurred 1 times
Character value 24180 occurred 1 times
Character value 24555 occurred 1 times
Character value 26032 occurred 1 times
$ _
```

Coursework: Counting words

(Summary only)

Write a program to count the number of words in its **standard input**.

Section 4

Example:
Numbering lines from standard input

Aim

AIM: To introduce the principle of reading lines from **standard input**, using `BufferedReader`.

File IO API: `BufferedReader`

- `java.io.BufferedReader`
  - wraps up an `InputStreamReader`

  provides **instance method** to read a whole line of characters.

- `readLine()`
  - takes no **method argument**s
  - **return**s `String`
    - ∗ next line of input from underlying `InputStreamReader`
    - ∗ or **null reference** if no more lines.

Numbering lines from standard input

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

// Program to add a line number to the lines from the standard input
// and show the result on the standard output.
public class LineNumber
{
  // The minimum number of digits in a line number.
  private static final int MINIMUM_LINE_NUMBER_DIGITS = 5;

  // The format to use with printf for the line number and line.
  private static final String LINE_FORMAT
    = "%0" + MINIMUM_LINE_NUMBER_DIGITS + "d %s%n";


```

Numbering lines from standard input

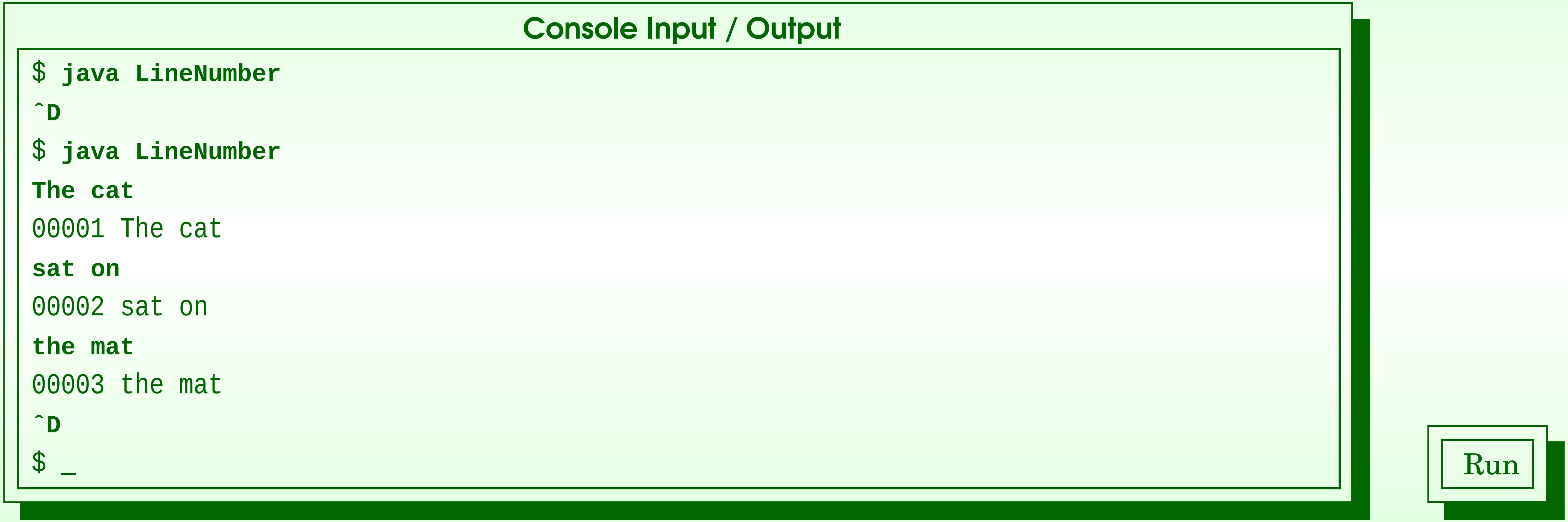
```
// Read each line from input, and copy to output with a count.
public static void main(String[] args)
{
  BufferedReader input
    = new BufferedReader(new InputStreamReader(System.in));
  try
  {
    // Now copy input to output, adding line numbers.
    int noOfLinesReadSoFar = 0;
    String currentLine;
    while ((currentLine = input.readLine()) != null)
    {
      noOfLinesReadSoFar++;
      System.out.printf(LINE_FORMAT, noOfLinesReadSoFar, currentLine);
    } // while
  } // try
```

Numbering lines from standard input

```
  catch (IOException exception)
  {
   System.err.println(exception);
  } // catch
  finally
  {
   try { input.close(); }
   catch (IOException exception)
    { System.err.println("Could not close input " + exception); }
  } // finally
 } // main

} // class LineNumber
```

Trying it



```
$ java LineNumber
^D
$ java LineNumber
The cat
00001 The cat
sat on
00002 sat on
the mat
00003 the mat
^D
$ _
```

Coursework: Deleting a field

(Summary only)

Write a program to delete a field in tab separated text from the **standard input**.

Section 5

Example: Numbering lines from text file to text file

Aim

AIM: To introduce the principle of reading from a **text file** and writing to another, using `BufferedReader` with `FileReader` and `PrintWriter` with `FileWriter`. We also meet `FileInputStream`, `OutputStream`, `FileOutputStream` and `OutputStreamWriter`.

Numbering lines from text file to text file

- `LineNumber` program
  - read data from **text file**
  - write result to another text file.
  - File names supplied as **command line argument**s.
- Also use `LineNumberException` **class**
  - not shown: similar to others.

Coffee time: Write the `LineNumberException` class.

File IO API: `FileInputStream`

- `java.io.FileInputStream`
  - **subclass** of `java.io.InputStream`
  - reads input bytes from file.
- E.g.

```
myDataAsBytes = new FileInputStream("my-binary-data");
```

File IO API: `FileReader`

- Wrap `FileInputStream` in `InputStreamReader`
  - can read **character**s from **file**
    - instead of **byte**s.
- Convenience: `java.io.FileReader`
  - creates required `FileInputStream`
  - and `InputStreamReader` internally.
- `FileReader` is **subclass** of `java.io.InputStreamReader`
  - has `read()` **instance method**
    - read **character**

  can be wrapped inside `BufferedReader`
  - to obtain `readLine()` instance method.
- One **constructor method** takes name of file to be accessed.

File IO API: `FileReader`

```java
FileReader fileReader = null;
try
{
  fileReader = new FileReader("my-data.txt");
  int currentCharacter;
  while ((currentCharacter = fileReader.read()) != -1)
  {
    ... do something with currentCharacter.
  } // while
} // try
catch (IOException exception)
{
  System.err.println(exception.getMessage());
} // catch
```

File IO API: FileReader

```
finally
{
  try { if (fileReader != null) fileReader.close(); }
  catch (IOException exception)
    { System.err.println("Could not close input file " + exception); }
} // finally
```

File IO API: `OutputStream`

- `java.io.OutputStream` allows writing of **byte**s
  - provides view of data as **byte stream**.
- Has **instance method** `write()`
  - write single **byte**.

File IO API: `OutputStreamWriter`

- `OutputStream` can be wrapped in `java.io.OutputStreamWriter`
  - provides view as sequence of **character**s
    - rather than **byte**s.
- Has **instance method** `write()`
  - write single character.

File IO API: `FileOutputStream`

- `java.io.FileOutputStream` is **subclass** of `java.io.OutputStream`
  - writes bytes to file.

File IO API: `FileWriter`

- Wrap `FileOutputStream` in `OutputStreamWriter`
  - can write **characters** to **file**
    - instead of **bytes**.
- Convenience: `java.io.FileWriter`
  - creates required `FileOutputStream`
  - and `OutputStreamWriter` internally.
- `FileWriter` is **subclass** of `java.io.OutputStreamWriter`
  - has `write()` **instance method**
    - write character.
- One **constructor method** takes name of file to be written to.

File IO API: FileWriter

```
FileWriter fileWriter = null;
try
{
  fileWriter = new FileWriter("my-results.txt");
  boolean iFeelLikeIt = ...
  while (iFeelLikeIt)
  {
    int currentCharacter = ...
    fileWriter.write(currentCharacter);
    ...
    iFeelLikeIt = ...
  } // while
} // try
```

File IO API: `FileWriter`

```
catch (IOException exception)
{
  System.err.println(exception.getMessage());
} // catch
finally
{
  try { if (fileWriter != null) fileWriter.close(); }
  catch (IOException exception)
    { System.err.println("Could not close output file " + exception); }
} // finally
```

- Notice call to `close()` instance method
  - if do not **close** output files
    - ∗ **data** written into `FileWriter` might still be in memory
    - ∗ never get written to physical file.

File IO API: `FileWriter`

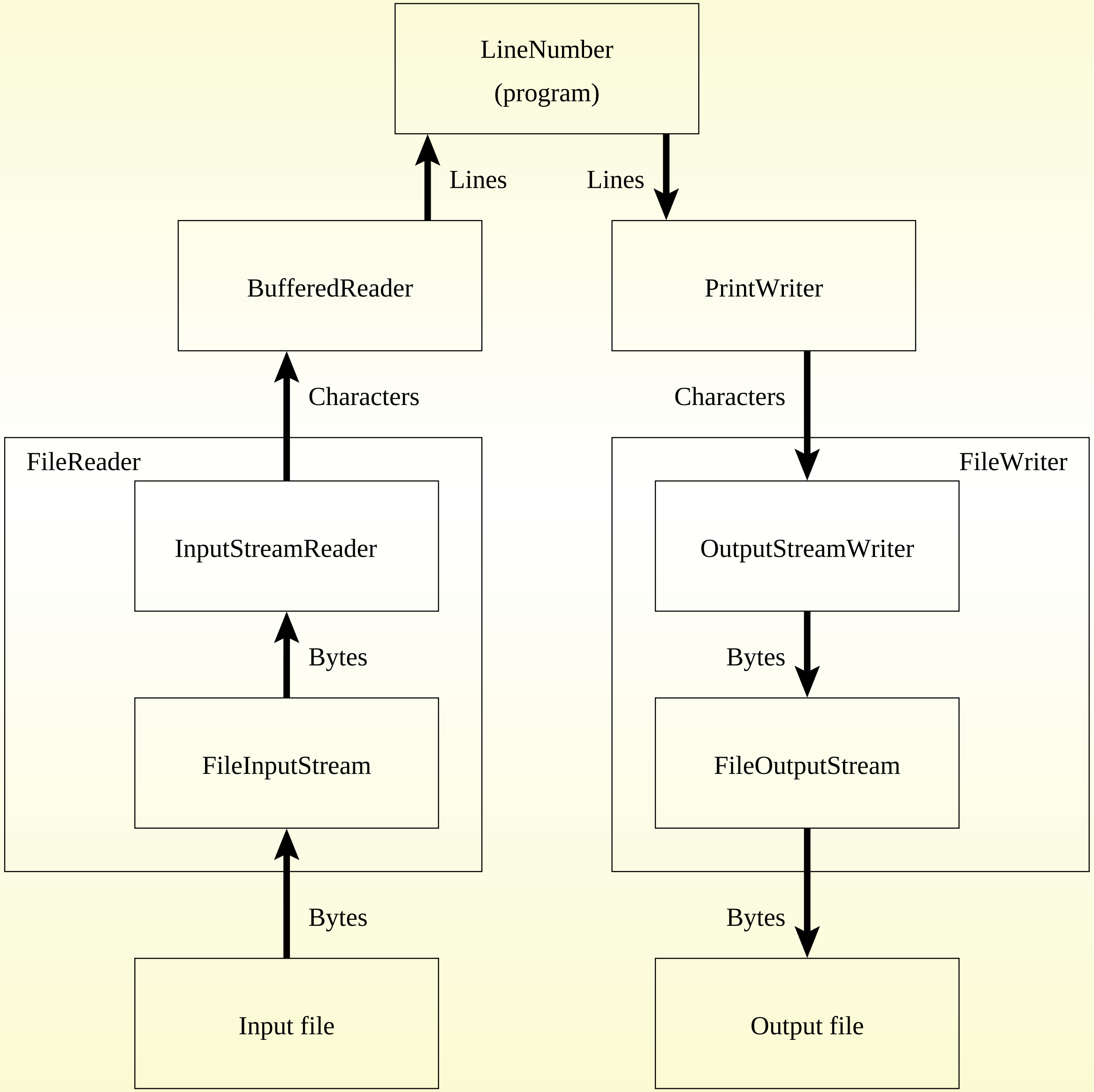
- Note: only lowest 16 **bits**
  - size of a **`char`**

  used by `write()`
  - avoids need to **cast** value to **`char`**
    - may have just obtained value from `read()` of `InputStream`.

File IO API: `PrintWriter`

- `java.io.PrintWriter` wraps up `OutputStreamWriter`
  - provides instance methods `println()` and `print()`
    - for range of possible **method arguments**.
  - Since Java 5.0 also has `printf()`.

Numbering lines from text file to text file



File IO API: `PrintWriter`: `checkError()`

- The **instance methods** of `java.io.PrintWriter` never **throw exceptions**!
  - Use `checkError()` to find out whether something has gone wrong.
    - **return** **`true`** iff there has been error.

File IO API: PrintWriter: checkError()

```
PrintWriter printWriter = null;
try
{
  printWriter = ...
  while (...)
  {
    ...
    printWriter.write(...);
    ...
  } // while
} // try
catch (IOException exception)
{
  System.err.println(exception.getMessage());
} // catch
```

File IO API: PrintWriter: checkError()

```
finally
{
  if (printWriter != null)
  {
    // printWriter.close() does not throw an exception.
    printWriter.close();
    if (printWriter.checkError())
      System.err.println("Something went wrong with the output");
  } // if
} // finally
```

Numbering lines from text file to text file

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

// Program to add a line number to the lines from an input file
// and produce the result in an output file.
// The two file names are given as command line arguments.
public class LineNumber
{
  // The minimum number of digits in a line number.
  private static final int MINIMUM_LINE_NUMBER_DIGITS = 5;

  // The format to use with printf for the line number and line.
  private static final String LINE_FORMAT
    = "%0" + MINIMUM_LINE_NUMBER_DIGITS + "d %s%n";
```

Numbering lines from text file to text file

Coffee time: Why need to set input and output to **null reference**?

```
// Read each line from input, and copy to output with a count.
public static void main(String[] args)
{
  BufferedReader input = null;
  PrintWriter output = null;
  try
  {
    if (args.length != 2)
      throw new LineNumberException
                 ("There must be exactly two arguments: infile outfile");

    input = new BufferedReader(new FileReader(args[0]));
    output = new PrintWriter(new FileWriter(args[1]));

```

Numbering lines from text file to text file

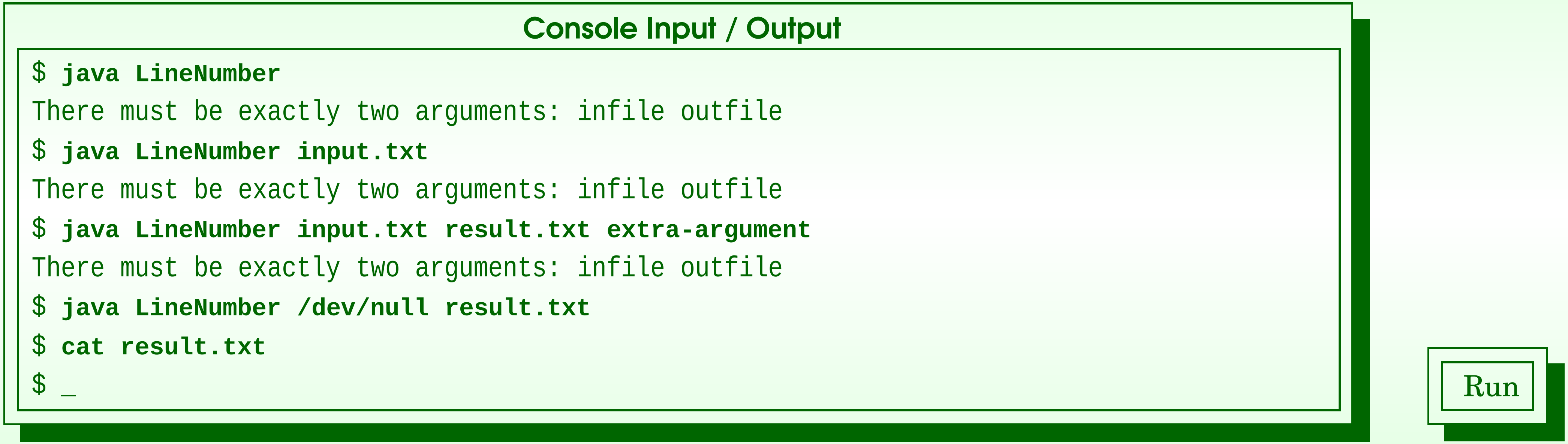
```
      // Now copy input to output, adding line numbers.
      int noOfLinesReadSoFar = 0;
      String currentLine;
      while ((currentLine = input.readLine()) != null)
      {
        noOfLinesReadSoFar++;
        output.printf(LINE_FORMAT, noOfLinesReadSoFar, currentLine);
      } // while
    } // try
    catch (LineNumberException exception)
    {
      // We report LineNumberExceptions to standard output.
      System.out.println(exception.getMessage());
    } // catch
    catch (IOException exception)
    {
      // Other exceptions go to standard error.
      System.err.println(exception);
    } // catch
```

Numbering lines from text file to text file

```
    finally
    {
      try { if (input != null) input.close(); }
      catch (IOException exception)
        { System.err.println("Could not close input " + exception); }
      if (output != null)
      {
        output.close();
        if (output.checkError())
          System.err.println("Something went wrong with the output");
      } // if
    } // finally
  } // main

} // class LineNumber
```

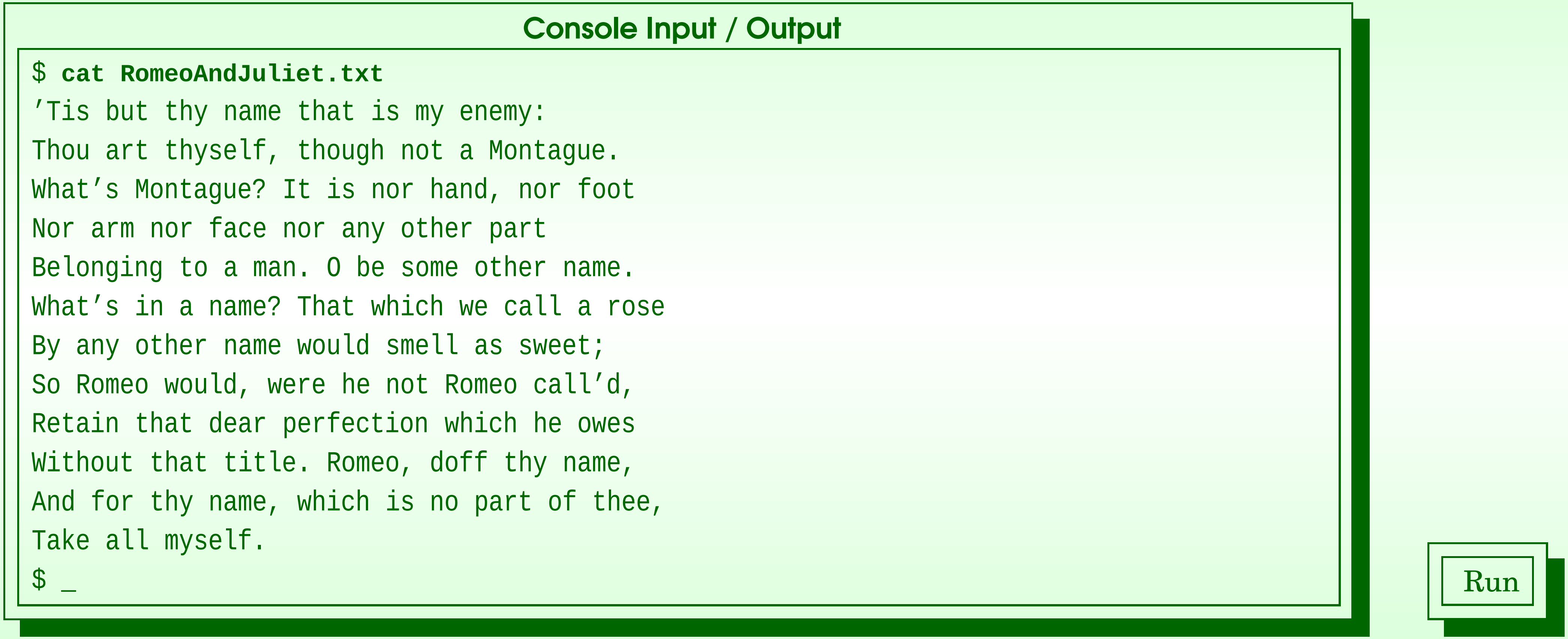
Trying it



Console Input / Output

```
$ java LineNumber
There must be exactly two arguments: infile outfile
$ java LineNumber input.txt
There must be exactly two arguments: infile outfile
$ java LineNumber input.txt result.txt extra-argument
There must be exactly two arguments: infile outfile
$ java LineNumber /dev/null result.txt
$ cat result.txt
$ _
```

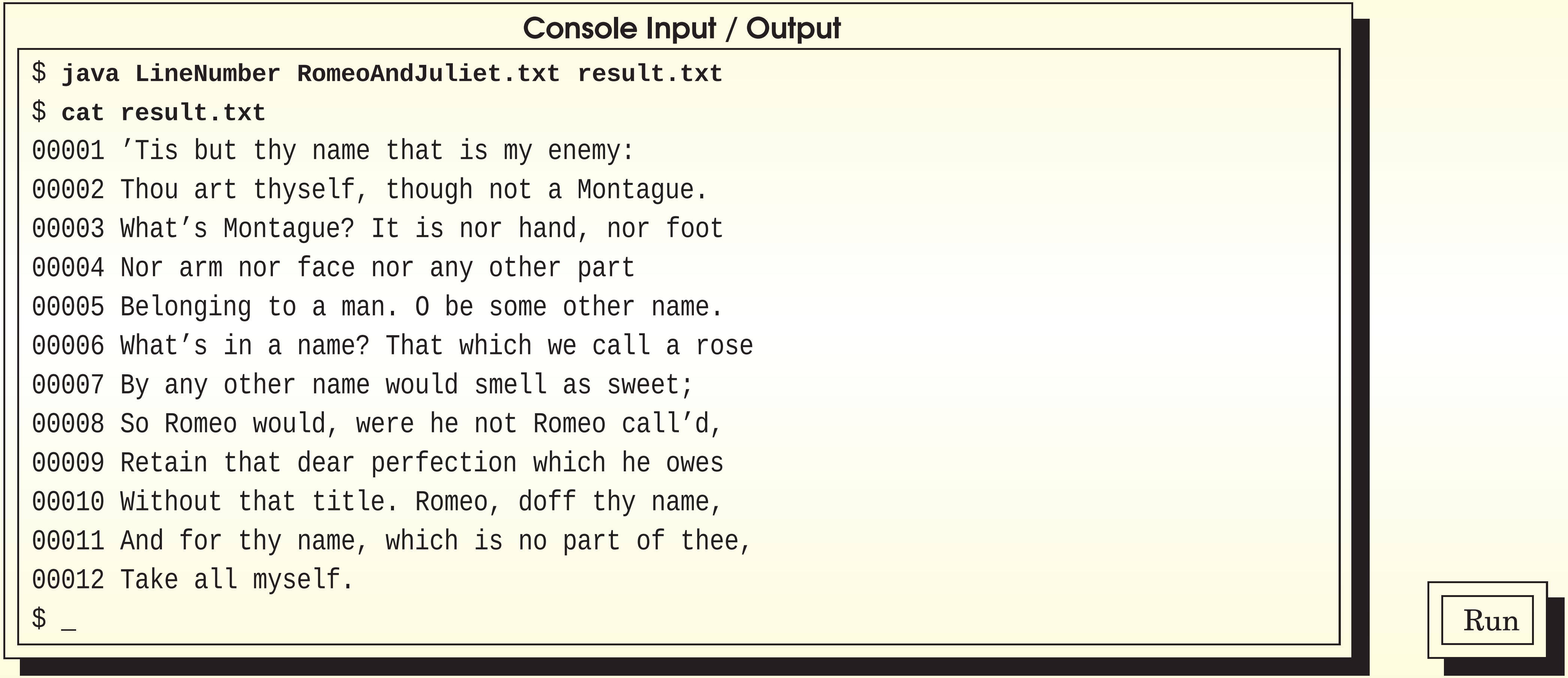
Trying it



Console Input / Output

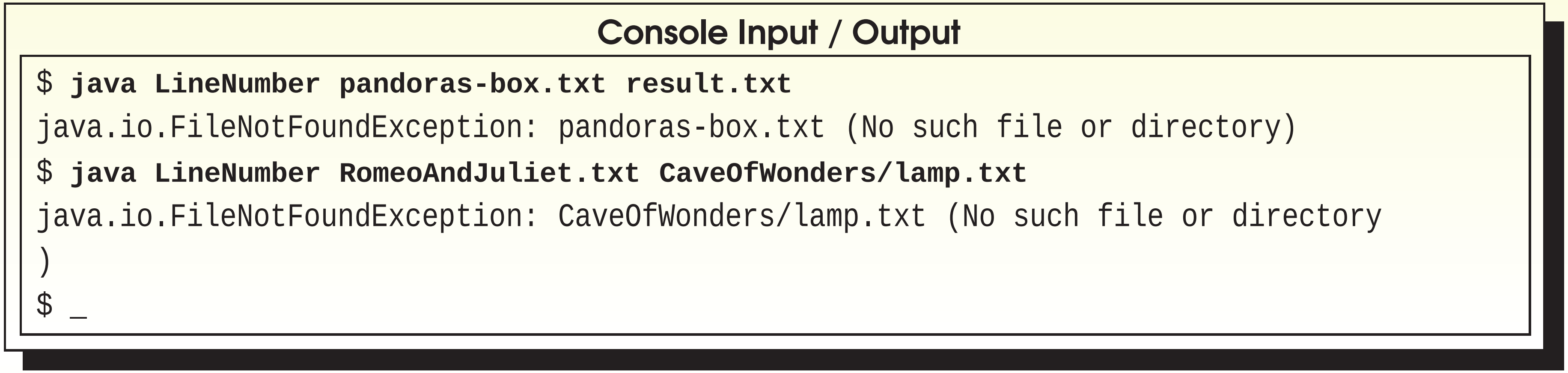
```
$ cat RomeoAndJuliet.txt
'Tis but thy name that is my enemy:
Thou art thyself, though not a Montague.
What's Montague? It is nor hand, nor foot
Nor arm nor face nor any other part
Belonging to a man. O be some other name.
What's in a name? That which we call a rose
By any other name would smell as sweet;
So Romeo would, were he not Romeo call'd,
Retain that dear perfection which he owes
Without that title. Romeo, doff thy name,
And for thy name, which is no part of thee,
Take all myself.
$ _
```

Trying it



```
$ java LineNumber RomeoAndJuliet.txt result.txt
$ cat result.txt
00001 'Tis but thy name that is my enemy:
00002 Thou art thyself, though not a Montague.
00003 What's Montague? It is nor hand, nor foot
00004 Nor arm nor face nor any other part
00005 Belonging to a man. O be some other name.
00006 What's in a name? That which we call a rose
00007 By any other name would smell as sweet;
00008 So Romeo would, were he not Romeo call'd,
00009 Retain that dear perfection which he owes
00010 Without that title. Romeo, doff thy name,
00011 And for thy name, which is no part of thee,
00012 Take all myself.
$ _
```

Trying it



Console Input / Output

```
$ java LineNumber pandoras-box.txt result.txt
java.io.FileNotFoundException: pandoras-box.txt (No such file or directory)
$ java LineNumber RomeoAndJuliet.txt CaveOfWonders/lamp.txt
java.io.FileNotFoundException: CaveOfWonders/lamp.txt (No such file or directory
)
$ _
```

Run

Coffee time: Observe the above **exception** – is it a surprise that an attempt to create a new file results in a complaint about it not being found?

Coursework: Deleting a field, from file to file

(Summary only)

Write a program to delete a field in tab separated text from a **file**, with the results in another file.

Section 6

Example:

Numbering lines from and to anywhere

Aim

AIM: To illustrate that reading from **text files** and from **standard input** is essentially the same thing, as is writing to **text files** and to **standard output**. We also look at testing for the existence of a **file** using the `File` **class**, and revisit `PrintWriter` and `PrintStream`.

Numbering lines from and to anywhere

- Wish to treat standard input in same way as file
  - get `BufferedReader` that gets input from either file
    - or standard input, as desired.
- Wish to treat standard output in same way as file
  - get `PrintWriter` that sends output to either file
    - or standard output, as desired.

Standard API: `System`: `out`: is an `OutputStream`

- `System.out` holds **reference** to **instance** of `java.io.OutputStream`
  - more precisely `java.io.PrintStream`
    - ∗ **subclass** of `OutputStream`.
- Unlike basic `OutputStream` objects
  `PrintStream` objects have extra **instance methods**
  - `print()`,
  - `println()`
  - and (since Java 5.0) `printf()`

  which write **character** representations as bytes.

Standard API: `System`: `err`: is an `OutputStream`

- `System.err` holds **reference** to **instance** of `java.io.PrintStream`
  - **subclass** of `java.io.OutputStream`.

File IO API: `PrintWriter`: versus `PrintStream`

- What is difference between `java.io.PrintStream` and `java.io.PrintWriter`?
- `PrintStream` is **subclass** of `OutputStream`
  - has `write()` **instance methods** for writing **bytes**
  - but *also* `print()`, `println()` and `printf()` for printing representations as **characters**
- `PrintWriter` is wrapper around **instance** of `java.io.OutputStreamWriter`
  - provides `print()`, `println()` and `printf()`
    - ∗ via that `OutputStreamWriter`.
  - has no way to write bytes.

File IO API: `PrintWriter`: versus `PrintStream`

- Desire to write *mixture* of bytes and characters to same stream highly unusual
  - nearly always want either all bytes
  - or all characters
    - sometimes with ability to print representations.
- `PrintStream` primarily exists for `System.out` and `System.err`
  - **standard output** / **standard error** available as byte stream
    - with convenient printing for error messages, debugging messages, or very simple programs.
- Programs that need representations as stream of characters
  - should use `PrintWriter`
    - *because* does not have instance methods to write bytes!

File IO API: PrintWriter: can also wrap an OutputStream

- System.out is an OutputStream
  - actually **subclass** PrintStream.
- If wish to treat as PrintWriter
  - wrap inside OutputStreamWriter
  - and then inside PrintWriter.

```
PrintWriter systemOut = new PrintWriter(new OutputStreamWriter(System.out));
```

- For convenience, one **constructor method** of PrintWriter takes OutputStream directly
  - **construct**s intermediate OutputStreamWriter internally.

```
PrintWriter systemOut = new PrintWriter(System.out);
```

File IO API: `PrintWriter`: can also wrap an `OutputStream`

- All **instance**s of output **class**es which wrap other output class **object**
  - may buffer output before sending to wrapped object
    - to speed up overall operation of programs.
- Buffers are **flush**ed by calling **instance method** `flush()`
  - or when output is **close**d via `close()`.
- For `PrintWriter` wrapping `System.out`
  - would want to enable **automatic flushing**
    - ensures **data** is sent all the way through whenever one of
    - `println()` or `printf()` has produced results.
- Automatic flushing enabled using constructor method with additional **`boolean`** **method argument**.

```
PrintWriter systemOut = new PrintWriter(System.out, true);
```

File IO API: `File`

- `java.io.File` allows examination of **file** properties
- Called `File`
  - but really about file *names* and properties.
- One **constructor method** takes path name of file **method argument**.
- Number of **instance methods**, e.g.
  - `exists()`
    - **returns** **`boolean`** indicating whether file actually exists.

Numbering lines from and to anywhere

Coffee time: Find out about the other features of the `File` **class** by looking at the **API** on-line documentation.

Numbering lines from and to anywhere

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintWriter;

// Program to add a line number to the lines from an input file
// and produce the result in an output file.
// The two file names are given as command line arguments.
// If a filename is missing, or is "-", then standard input/output is used.
public class LineNumber
{
  // The minimum number of digits in a line number.
  private static final int MINIMUM_LINE_NUMBER_DIGITS = 5;

  // The format to use with printf for the line number and line.
  private static final String LINE_FORMAT
    = "%0" + MINIMUM_LINE_NUMBER_DIGITS + "d %s%n";
```

Numbering lines from and to anywhere

```java
// Read each line from input, and copy to output with a count.
public static void main(String[] args)
{
  BufferedReader input = null;
  PrintWriter output = null;
  try
  {
    // Check for too many args before opening files, in case wrong names.
    if (args.length > 2)
      throw new LineNumberException("Too many arguments");

    if (args.length < 1 || args[0].equals("-"))
      input = new BufferedReader(new InputStreamReader(System.in));
    else
      input = new BufferedReader(new FileReader(args[0]));

```

Numbering lines from and to anywhere

```
if (args.length < 2 || args[1].equals("-"))
  output = new PrintWriter(System.out, true);
else
{
  if (new File(args[1]).exists())
    throw new LineNumberException("Output file "
                                  + args[1] + " already exists");

  output = new PrintWriter(new FileWriter(args[1]));
} // else

```

Numbering lines from and to anywhere

```
      // Now copy input to output, adding line numbers.
      int noOfLinesReadSoFar = 0;
      String currentLine;
      while ((currentLine = input.readLine()) != null)
      {
        noOfLinesReadSoFar++;
        output.printf(LINE_FORMAT, noOfLinesReadSoFar, currentLine);
      } // while
    } // try
    catch (LineNumberException exception)
    {
      // We report LineNumberExceptions to standard output.
      System.out.println(exception.getMessage());
    } // catch
    catch (IOException exception)
    {
      // Other exceptions go to standard error.
      System.err.println(exception);
    } // catch
```

Numbering lines from and to anywhere

```
    finally
    {
      try { if (input != null) input.close(); }
      catch (IOException exception)
        { System.err.println("Could not close input " + exception); }
      if (output != null)
      {
        output.close();
        if (output.checkError())
          System.err.println("Something went wrong with the output");
      } // if
    } // finally
  } // main

} // class LineNumber
```

Trying it

Console Input / Output

```
$ java LineNumber input.txt result.txt extra-argument
Too many arguments
$ _
```

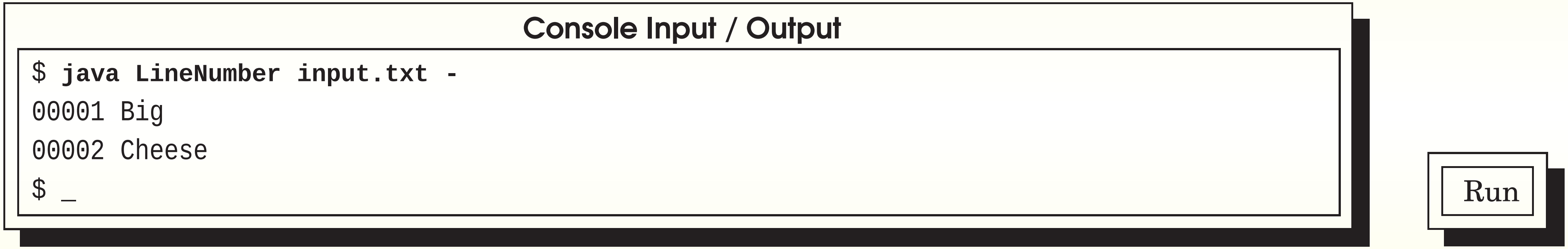
Run

Console Input / Output

```
$ cat input.txt
Big
Cheese
$ java LineNumber input.txt result.txt
$ cat result.txt
00001 Big
00002 Cheese
$ _
```

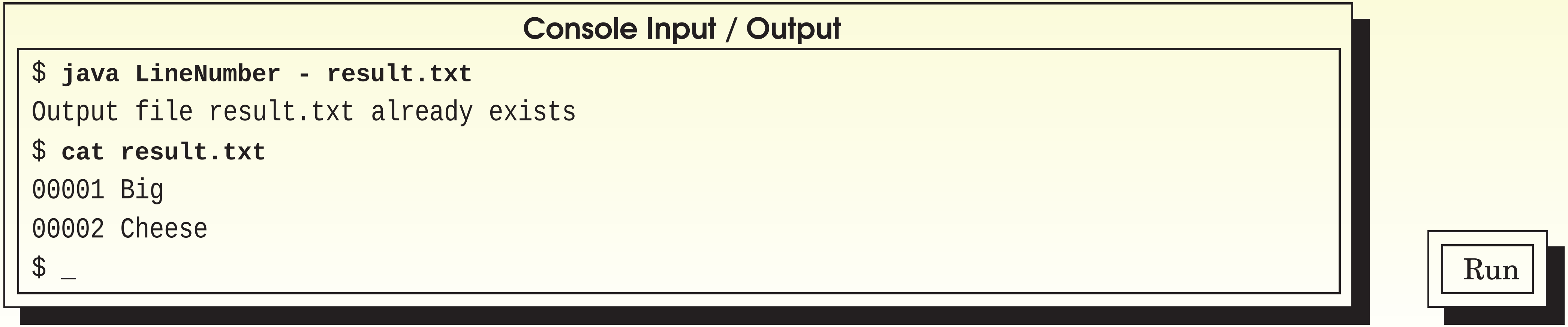
Run

Trying it

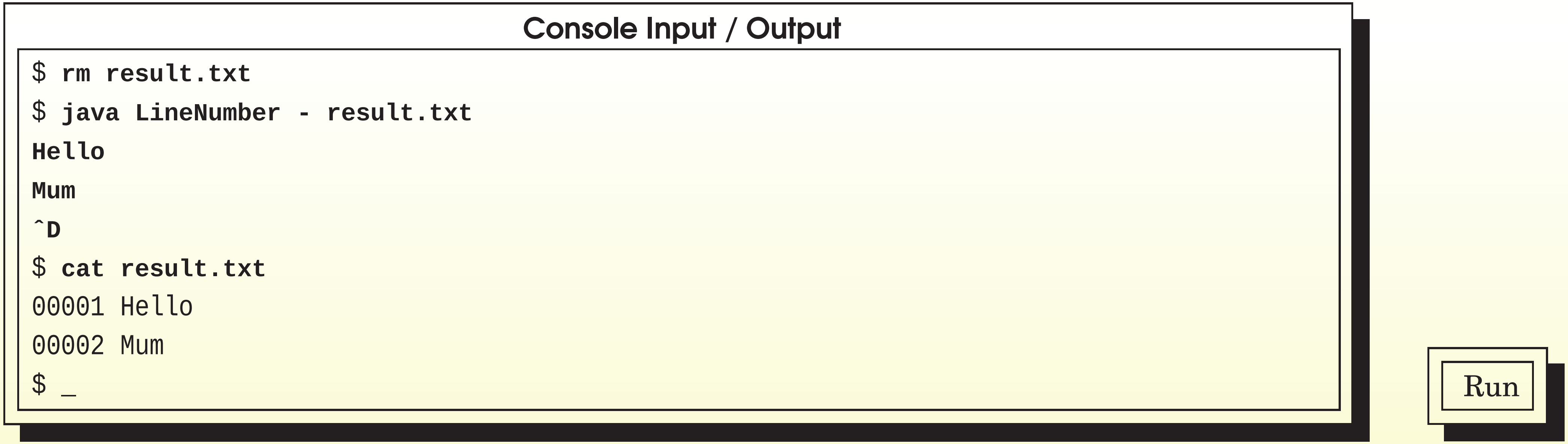


```
$ java LineNumber input.txt -
00001 Big
00002 Cheese
$ _
```

Trying it

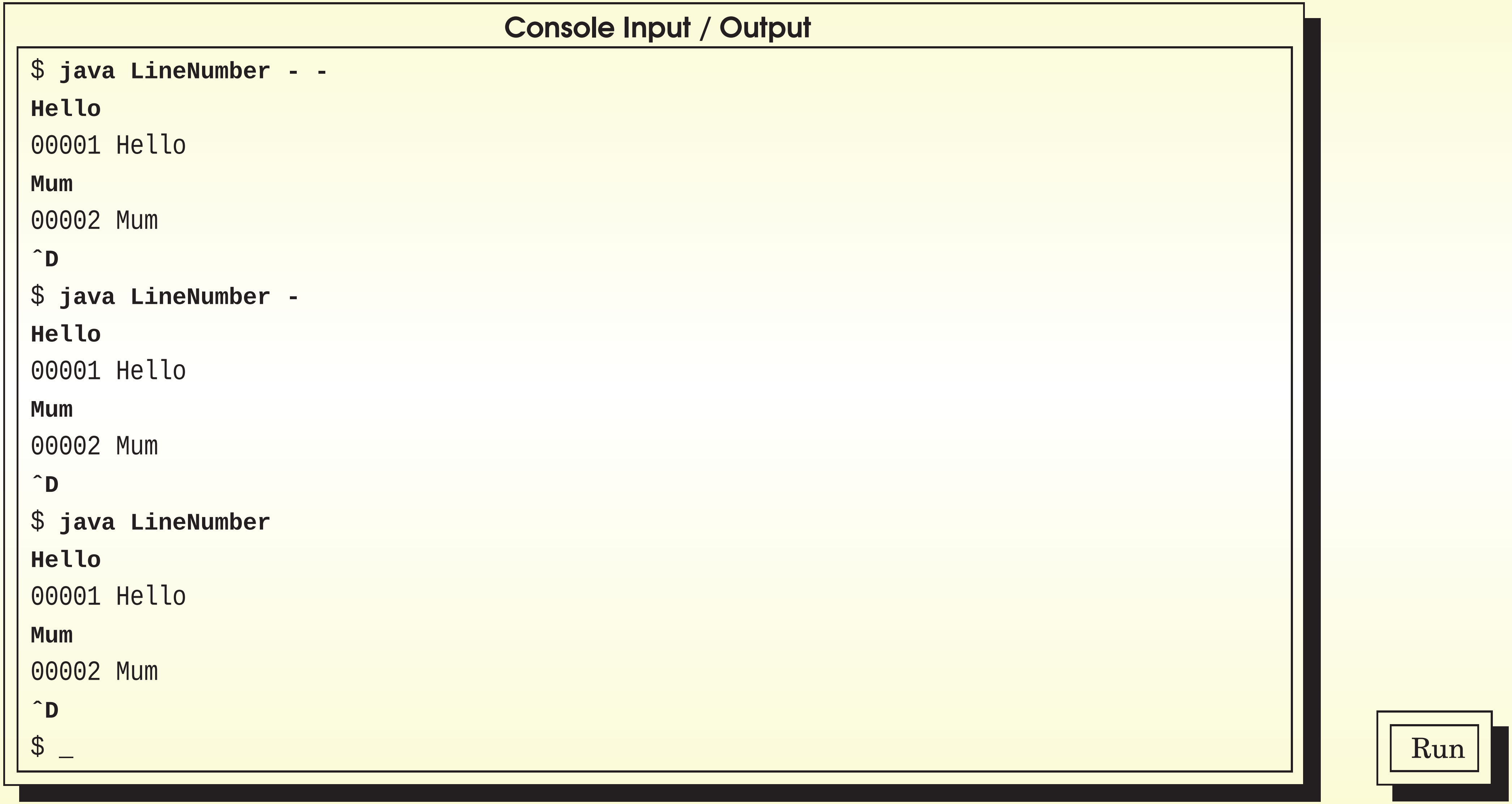


```
$ java LineNumber - result.txt
Output file result.txt already exists
$ cat result.txt
00001 Big
00002 Cheese
$ _
```



```
$ rm result.txt
$ java LineNumber - result.txt
Hello
Mum
^D
$ cat result.txt
00001 Hello
00002 Mum
$ _
```

Trying it



```
$ java LineNumber - -
Hello
00001 Hello
Mum
00002 Mum
^D
$ java LineNumber -
Hello
00001 Hello
Mum
00002 Mum
^D
$ java LineNumber
Hello
00001 Hello
Mum
00002 Mum
^D
$ _
```

Coursework: Deleting a field, from anywhere to anywhere

(Summary only)

Write a program to delete a field in tab separated text either from **standard input** or a **file**, with the results going to either **standard output** or another file.

Section 7

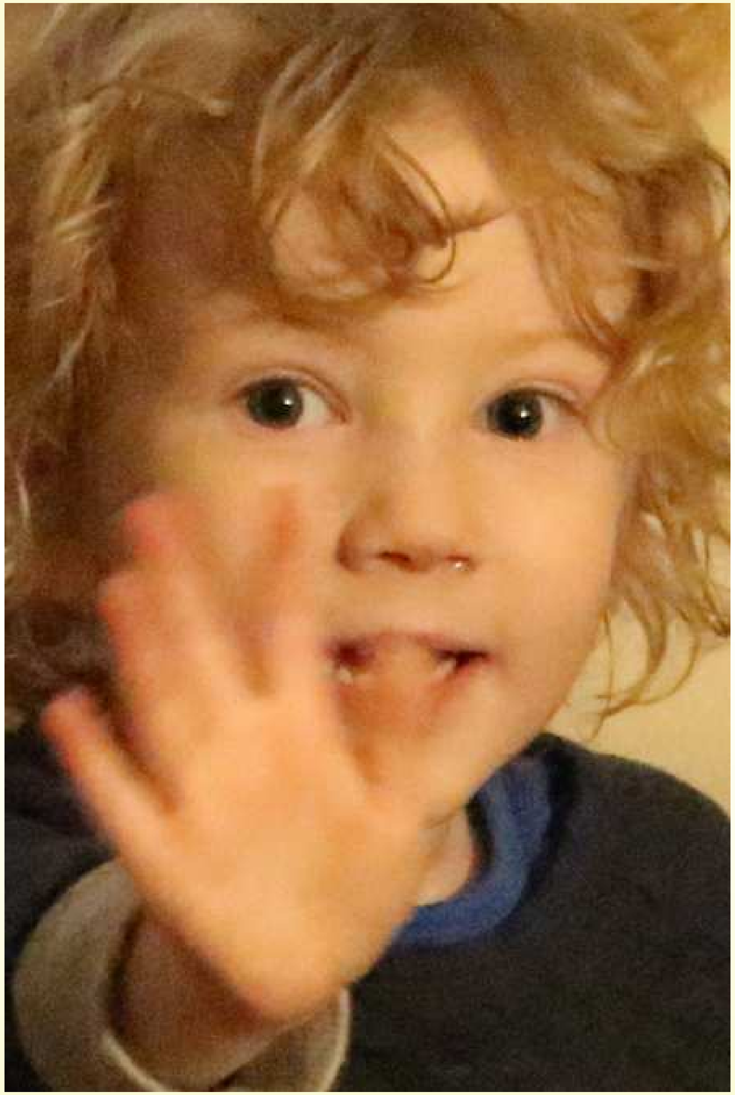
Example:
Text photographs

Aim

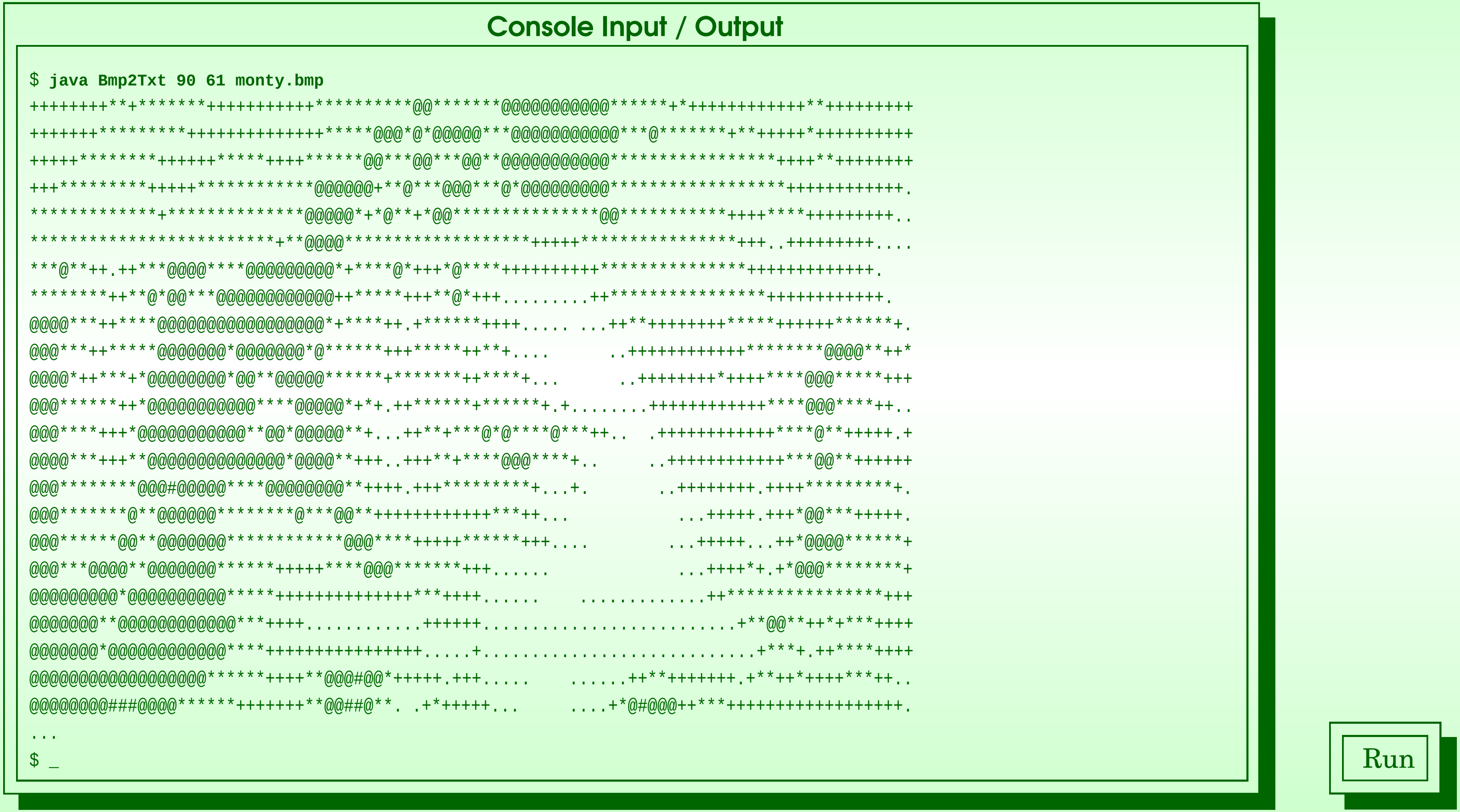
AIM: To see an example of reading **binary files**, where we did not choose the **file format**. This includes the process of turning **`byte`**s into **`int`**s, using a **shift operator** and an **integer bitwise operator**.

Text photographs

- ‘ASCII art’ impressionist version of given photograph
  - user chooses width and height of output text image.
  - Dark regions of image represented using dark characters, e.g. '#'
    - lighter as e.g. '*', '.', space.
- E.g. from this original . . .



Text photographs



```
$ java Bmp2Txt 90 61 monty.bmp
++++++++**+*******+++++++++++**********@@*******@@@@@@@@@@@******+*++++++++++++**+++++++++
+++++++*********++++++++++++++*****@@@*@*@@@@@***@@@@@@@@@@@***@*******+**+++++*++++++++++
+++++********++++++*****++++******@@***@@***@@**@@@@@@@@@@@*****************++++**++++++++
+++*********+++++************@@@@@@+**@***@@@***@*@@@@@@@@@******************++++++++++++.
*************+**************@@@@@*+*@**+*@@***************@@***********++++****+++++++++..
*************************+**@@@@*******************+++++****************+++..+++++++++....
***@**++.++***@@@@****@@@@@@@@@*+****@*+++*@****++++++++++***************+++++++++++++.
********++**@*@@***@@@@@@@@@@@@++*****+++**@*+++.........++****************++++++++++++.
@@@@***++****@@@@@@@@@@@@@@@@@*+****++.+******++++..... ...++**++++++++*****++++++******+.
@@@***++*****@@@@@@@*@@@@@@@*@******+++*****++**+....      ..++++++++++++********@@@@**++*
@@@@*++***+*@@@@@@@@*@@**@@@@@******+*******++****+...      ..++++++++*++++****@@@*****+++
@@@******++*@@@@@@@@@@@****@@@@@*+*+.++******+******+.+........++++++++++++****@@@****++..
@@@****+++*@@@@@@@@@@@**@@*@@@@@**+...++**+***@*@****@***++..  .++++++++++++****@**+++++.+
@@@@***+++**@@@@@@@@@@@@@@*@@@@**+++..+++**+****@@@****+..     ..++++++++++++***@@**++++++
@@@********@@@#@@@@@****@@@@@@@@**++++.+++*********+...+.       ..++++++++.++++*********+.
@@@*******@**@@@@@@********@***@@**++++++++++++***++...           ...+++++.+++*@@***+++++.
@@@******@@**@@@@@@@************@@@****+++++******+++....        ...+++++...++*@@@@******+
@@@***@@@@**@@@@@@@******+++++****@@@*******+++......             ...++++*+.+*@@@********+
@@@@@@@@@*@@@@@@@@@@*****++++++++++++++***++++......    .............++****************+++
@@@@@@@**@@@@@@@@@@@@***++++............++++++..........................+**@@**++*+***++++
@@@@@@@*@@@@@@@@@@@@****++++++++++++++++.....+............................+***+.++****++++
@@@@@@@@@@@@@@@@@@******++++**@@@#@@*+++++.+++.....    ......++**+++++++.+**++*++++***++..
@@@@@@@@###@@@@******+++++++**@@##@**. .+*+++++...     ....+*@#@@@++***++++++++++++++++++.
...
$ _
```

Text photographs

- Original image must be 24 **bit** per pixel `.bmp` **file**
  - easy to produce and not (usually) compressed
    - hence fairly easy to read as **data**.
- Program reads image data as **byte** stream
  - using `FileInputStream`.
- Much code is about making sense of bytes in file
  - we didn't choose file format.
  - e.g. width and height of image
    - stored at certain point
    - as sequence of bytes in particular order.

Text photographs

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

// Simple program to produce a text version of a 24 bit BMP format image file.
// The first argument is the desired text width, the second is the height.
// The third argument is the name of BMP file.
// The text image is produced on the standard output.
public class Bmp2Txt
{
```

Text photographs

```
// The characters used for the text image.
// The first is used for the darkest pixels,
// the second for the next lightest, and so on.
// A good choice will depend on the font in use on the output.
// (We should reverse the order when using white print on black.)
private static final String SHADES_STRING = "#@*+. ";

// The above is for convenient editing if we want to alter the
// characters used. This next array is actually used to
// map a scaled brightness on to a text character.
private static final char[] SHADE_CHARS = SHADES_STRING.toCharArray();

// The bytes from the input image.
private static FileInputStream inputImage;

```

Text photographs

```java
// The width and height of the input image.
private static int inputWidth, inputHeight;

// The width and height of the desired text image.
private static int outputWidth, outputHeight;

// Our output image will be stored in this 2D array.
// Position 0,0 is bottom left.
// Each pixel records the monochrome brightness level.
private static int[][] outputImage;
```

Text photographs

- Every time read byte, check have not reached end of file
  - have separate method.

```
// Read a single byte from the input image file
// and throw an exception if there is none left!
private static int readByte() throws IOException
{
  int result = inputImage.read();
  if (result == -1)
    throw new IOException("Unexpected end of file");
  return result;
} // readByte
```

Text photographs

- Need skip specific number of bytes from time to time
  - stuff not relevant to program.

```java
// Skip irrelevant bytes from the input image file.
private static void skipIrrelvantBytes(int skipCount) throws IOException
{
  for (int count = 1; count <= skipCount; count++)
    readByte();
} // skipIrrelvantBytes
```

Text photographs

- Heigth and width of image stored at certain point in file
  - using four consecutive bytes each.
- Need to read these four bytes
  - turn them into **integer** they represent.

Expression: arithmetic: shift operators

- More **arithmetic operator**s: **shift operator**s.
- The **left shift operator**, <<
  - yields number obtained by shifting first **operand** left by second operand number of **bit**s
  - placing zeroes on right.
- The **unsigned right shift** operator, >>>
  - shifts rightwards
  - placing zeroes on left.
- The **signed right shift** operator, >>
  - same except places ones on left if number negative.

Expression: arithmetic: shift operators

- E.g. `1000` is `0001111101000` in **binary**.

4096	**2048**	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
0	0	0	1	1	1	1	1	0	1	0	0	0
0+	0+	0+	512+	256+	128+	64+	32+	0+	8+	0+	0+	0 = 1000

- Shift left three places: `1000 << 3`
  - get `8000` which is `1111101000000` in binary.

4096	**2048**	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	1	1	1	1	0	1	0	0	0	0	0	0
4096+	2048+	1024+	512+	256+	0+	64+	0+	0+	0+	0+	0+	0 = 8000

Expression: arithmetic: shift operators

- `1000 >> 3` and `1000 >>> 3` both yield `0000001111101` (125.)

4096	**2048**	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
0	0	0	0	0	0	1	1	1	1	1	0	1
0+	0+	0+	0+	0+	0+	64+	32+	16+	8+	4+	0+	1 = 125

- Shift left n bits same effect as **multiplication** by 2^n
  - discarding overflow.
- Signed shift right by n bits same effect as **division** by 2^n
  - discarding remainder.

Expression: arithmetic: integer bitwise operators

- The **operator**s |, &, and ^
  - applied to numeric **operand**s
  - **integer bitwise or**, **integer bitwise and** and **integer bitwise exclusive or**

bit n **of** `op1`	**bit** n **of** `op2`	**bit** n **of** `op1 | op2`	**bit** n **of** `op1 & op2`	**bit** n **of** `op1 ^ op2`
`0`	`0`	`0`	`0`	`0`
`0`	`1`	`1`	`0`	`1`
`1`	`0`	`1`	`0`	`1`
`1`	`1`	`1`	`1`	`0`

Expression: arithmetic: integer bitwise operators

- E.g. `1000` is `1111101000` in **binary**
  - anded with `23` which is `0000010111` in binary
  - yields `0000000000`
    - have no corresponding bit values in common.
- When or-ed together
  - yields `1111111111` in binary, which is `1023`.
- `1023` = `1000 + 23`
  - **integer bitwise or** same as **addition**
    only when two numbers have no corresponding bits with same value.

Text photographs

- In BMP file, four bytes representing width or height
  - least significant byte comes first
  - so left shift second byte by 8, third by 16 and fourth by 24
  - then **integer bitwise or** all four.

```
// Read an int from the next four bytes in the input image file.
// Least significant byte is first.
private static int readInt() throws IOException
{
  return readByte() | readByte() << 8 | readByte() << 16 | readByte() << 24;
} // readInt
```

Coffee time: How could we have used **multiplication** and **addition** to achieve the same result as the **left shift** and **integer bitwise or** above?

Text photographs

- Each input pixel represented as 3 bytes: red, green and blue components.
- Convert to monochrome brightness:
  green perceived brighter than red and red brighter than blue.
  - Commonly used ratio: $299 : 587 : 114$

```
// Read a pixel value from the input file and return its brightness.
// The pixel is stored as 3 bytes for RGB.
// Compute the brightness as (R*299 + G*587 + B*114)/1000.
private static int readPixelBrightness() throws IOException
{
  int red = readByte();
  int green = readByte();
  int blue = readByte();
  return (red * 299 + green * 587 + blue * 114) / 1000;
} // readPixelBrightness
```

Text photographs

- At certain point in file, image stored as
  - height number of rows
    - each with width number of pixels.
- First row corresponds to image bottom
  - first pixel in row corresponds to image left.
- Read pixel values
  - store each one in corresponding scaled pixel of output image
  - output image typically many fewer pixels than input.

Text photographs

```java
// Read the image from the input file and scale into the output array.
private static void readImage() throws IOException
{
  // The first row of input pixels is the bottom of the image.
  // I.e., in a BMP file, position 0,0 is bottom left.
  for (int inputY = 0; inputY < inputHeight; inputY++)
  {
    for (int inputX = 0; inputX < inputWidth; inputX++)
    {
      int pixelValue = readPixelBrightness();
      // This pixel address needs to be scaled to fit output image.
      int outputX = inputX * outputWidth / inputWidth;
      int outputY = inputY * outputHeight / inputHeight;
      // Add the input pixel value to the output pixel,
      outputImage[outputX][outputY] += pixelValue;
    } // for
    // Each row of the input image is zero padded to a multiple of 4 bytes.
    skipIrrelvantBytes(inputWidth % 4);
  } // for
} // readImage
```

Text photographs

- Need find brightness of brightest pixel in output image
  - so can scale each value to range of output characters chosen.
- Zero brightness mapped onto darkest character
  - maximum brightness onto lightest character
  - others linearly between.

Text photographs

```
// Find the highest valued pixel in the output image.
private static int maxOutputBrightness()
{
  int maxBrightnessSoFar = 0;
  for (int y = 0; y < outputHeight; y++)
    for (int x = 0; x < outputWidth; x++)
      if (outputImage[x][y] > maxBrightnessSoFar)
        maxBrightnessSoFar = outputImage[x][y];
  return maxBrightnessSoFar;
} // maxOutputBrightness
```

Text photographs

```
// Write the text image to standard output.
private static void writeTextImage()
{
  int maxBrightness = maxOutputBrightness();
  // Scale each pixel brightness to one of the SHADE_CHARS.
  for (int y = outputHeight - 1; y >= 0; y--)
  {
    for (int x = 0; x < outputWidth; x++)
      System.out.print(SHADE_CHARS[outputImage[x][y] * SHADE_CHARS.length
                                                     / (maxBrightness + 1)]);
    System.out.println();
  } // for
} // writeTextImage
```

Text photographs

```java
// The main method gets arguments and parses the image file at the top level.
public static void main(String[] args)
{
  // The name of the input image file, which must be in 24 bit BMP format.
  String filename = null;
  try
  {
    // Check we have three arguments.
    if (args.length != 3)
      throw new IllegalArgumentException(); // Caught below.

    // The first two command line arguments
    // are the required width and height of the text image.
    outputWidth = Integer.parseInt(args[0]);
    outputHeight = Integer.parseInt(args[1]);
    outputImage = new int[outputWidth][outputHeight];

```

Text photographs

```
      // The third argument is the original BMP image file name.
      filename = args[2];
      inputImage = new FileInputStream(filename);

      skipIrrelvantBytes(18);
      inputWidth = readInt();
      inputHeight = readInt();
      skipIrrelvantBytes(28);
      readImage();

      // Check end of file.
      if (inputImage.read() != -1)
        throw new IOException("Data after end of image");

      writeTextImage();
    } // try
```

Text photographs

```java
    catch (NumberFormatException exception)
    {
      System.err.println("Supplied dimension is not a number: "
                         + exception.getMessage());
    } // catch
    catch (IllegalArgumentException exception)
    {
      System.err.println("Please (only) supply: width height filename");
    } // catch
    catch (FileNotFoundException exception)
    {
      System.err.println("Cannot open image file " + filename);
    } // catch
    catch (IOException exception)
    {
      System.err.println("Problem reading image file: "
                         + exception.getMessage());
    } // catch
```

Text photographs

```
  finally
  {
   try { if (inputImage != null) inputImage.close(); }
   catch (IOException exception)
    { System.err.println("Could not close image file " + exception); }
  } // finally
 } // main

} // class Bmp2Txt
```

Coursework: Encoding binary in text

(Summary only)

Write a program to encode a **binary file** as an **ASCII text file**, so that it can be sent in an email.

Section 8

Example:
Contour points

Aim

AIM: To show an example of writing and reading **binary files** where we choose the **data** format, using `DataOutputStream` and `DataInputStream` **classes**.

Contour points

- Wish to build application manipulating contour points in terrain surface model.
- Do not present whole program – nor its full requirements!
- Assume program will process and generate large amounts of **data**
  - wish to store in **binary file format**
    - more compact.
- Present early stage of development
  - for exploring writing to / reading from **binary file**s
  - where we chose data format.

File IO API: `DataOutputStream`

- `java.io.DataOutputStream` allows writing **primitive type** values to **binary file**.
  - Is **subclass** of `java.io.OutputStream`
    - ∗ **instance**s also wrap `OutputStream`
    - ∗ including subclasses, e.g. `java.io.FileOutputStream`.
- E.g. `DataOutputStream` **object** which writes to **file** `out.dat`:

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("out.dat"));
```

- Has **instance method**s to write all kinds of primitive type
  - e.g. `writeInt()`
    - ∗ write **`int`** value in four **byte**s

    `writeShort()`
    - ∗ write **`short`** value in two bytes.

File IO API: `DataOutputStream`

- *Most* significant byte of numbers written first
  - but no need to worry about byte order
    - if intend to read **data** back using corresponding `readXXX()` from `java.io.DataInputStream`.
- Instances of `java.lang.String` written
  - using `writeUTF()`
    - saves text in **8-bit Unicode Transformation Format file encoding** (ish)
    - all **Unicode characters** represented.

File IO API: `DataInputStream`

- `DataInputStream` used to read values from **binary file**
  - especially if written by `DataOutputStream`.
  - Is **subclass** of `java.io.InputStream`
    - **instance**s also wrap `InputStream`
    - including subclasses, e.g. `java.io.FileInputStream`.
- E.g. `DataInputStream` **object** which reads from **file** `in.dat`:

```
DataInputStream in = new DataInputStream(new FileInputStream("in.dat"));
```

- Has **instance method**s to read all kinds of **primitive type**
  - `readInt()`
    - read **`int`** value from four **byte**s

    `readShort()`
    - read **`short`** value from two bytes.

File IO API: `DataInputStream`

- *Most* significant byte of numbers read first
  - but no need to worry about byte order
    - if intend to read data written by `writeXXX()` from `DataOutputStream`.
- `Strings` written using `writeUTF()` read using `readUTF()`.

Contour points

Coffee time: Why could we *not* have used `DataInputStream` to read the four **byte** integer values for width and height, from the input image binary file in the last example?

- Early development stage `ContourPoint` class
  - assume points modelled on two-dimensional grid
    - with four-digit number for each X / Y

    and **integer** height above sea level
    - (negative heights for below sea).
- Use **`short`**s for X / Y
  - **`int`** for height.

Contour points

```java
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

// Representation of a contour point with X,Y grid reference
// and height above sea level.
public class ContourPoint
{
  // gridX and gridY are in the range 0-9999, so a short will do nicely.
  private final short gridX, gridY;

  // Height has a wider range, but int is plenty.
  private final int height;
```

Contour points

```java
// Construct a ContourPoint with the given dimensions.
public ContourPoint(int requiredGridX, int requiredGridY, int requiredHeight)
{
  gridX = (short) requiredGridX;
  gridY = (short) requiredGridY;
  height = requiredHeight;
} // ContourPoint
```

Contour points

- Second **constructor method** reads dimensions from `DataInputStream`
  - assumed written by `write()` (below)
    - ∗ or `IOException` **throw**n.

```
// Construct a ContourPoint, by reading the dimensions
// from the given DataInputStream.
public ContourPoint(DataInputStream in) throws IOException
{
  gridX = in.readShort();
  gridY = in.readShort();
  height = in.readInt();
} // ContourPoint
```

Contour points

- Write dimensions
  - in form expected by second constructor.

```
// Write the three dimensions to a given DataOutputStream
// so that it can be read back into the above constructor.
public void write(DataOutputStream out) throws IOException
{
  out.writeShort(gridX);
  out.writeShort(gridY);
  out.writeInt(height);
} // write
```

Contour points

```java
  // Accessor for gridX.
  public short getGridX()
  {
    return gridX;
  } // getGridX


  // Accessor for gridY.
  public short getGridY()
  {
    return gridY;
  } // getGridY


  // Accessor for height.
  public int getHeight()
  {
    return height;
  } // getHeight
```

Contour points

```java
// Linear interpolation between this and a given other point.
public ContourPoint[] interpolate(ContourPoint endPoint, int noOfSteps)
{
  ContourPoint[] result = new ContourPoint[noOfSteps];

  for (int stepCount = 1; stepCount <= noOfSteps; stepCount++)
  {
    short newGridX = (short) (gridX + stepCount * (endPoint.gridX - gridX)
                                      / (noOfSteps + 1));
    short newGridY = (short) (gridY + stepCount * (endPoint.gridY - gridY)
                                      /  (noOfSteps + 1));
    // Cast stepCount to long, to avoid int overflow.
    int newHeight = (int) (height + (long)stepCount
                                    * (endPoint.height - height)
                                    / (noOfSteps + 1));
    result[stepCount - 1] = new ContourPoint(newGridX, newGridY, newHeight);
  } // for
  return result;
} // interpolate
```

Contour points

```java
// Return a String representing the point.
@Override
public String toString()
{
  return "(" + gridX + "," + gridY + "," + height + ")";
} // toString
```

Contour points

```java
// Purely for testing during development, and so does not catch exceptions.
public static void main(String[] args) throws Exception
{
  ContourPoint point1 = new ContourPoint(0, 0, 0);
  ContourPoint point2 = new ContourPoint(9999, 9999, 100000000);

  DataOutputStream output
    = new DataOutputStream(new FileOutputStream("test.dat"));

  // Test the following interpolation steps.
  int[] trySteps = {0, 10, 100};

  // Write the number of lists.
  output.writeByte(trySteps.length);
```

Contour points

```java
for (int tryStep : trySteps)
{
  ContourPoint[] interpolation = point1.interpolate(point2, tryStep);
  // Write the length of this list,
  // plus 2 to include the original points.
  output.writeInt(interpolation.length + 2);
  // Now write the first point.
  point1.write(output);
  // Now write each interpolated point.
  for (ContourPoint aPoint : interpolation)
    aPoint.write(output);
  // Now write the last point.
  point2.write(output);
} // for

output.close();
```

Contour points

```
DataInputStream input
  = new DataInputStream(new FileInputStream("test.dat"));

// Read the number of lists.
int noOfLists = input.readByte();
for (int count = 1; count <= noOfLists; count++)
{
  // Read the length of this list.
  int length = input.readInt();
  ContourPoint[] pointArray = new ContourPoint[length];

  // Now read each point.
  for (int pointIndex = 0; pointIndex < length; pointIndex++)
    // Construct a point from the file.
    pointArray[pointIndex] = new ContourPoint(input);

```

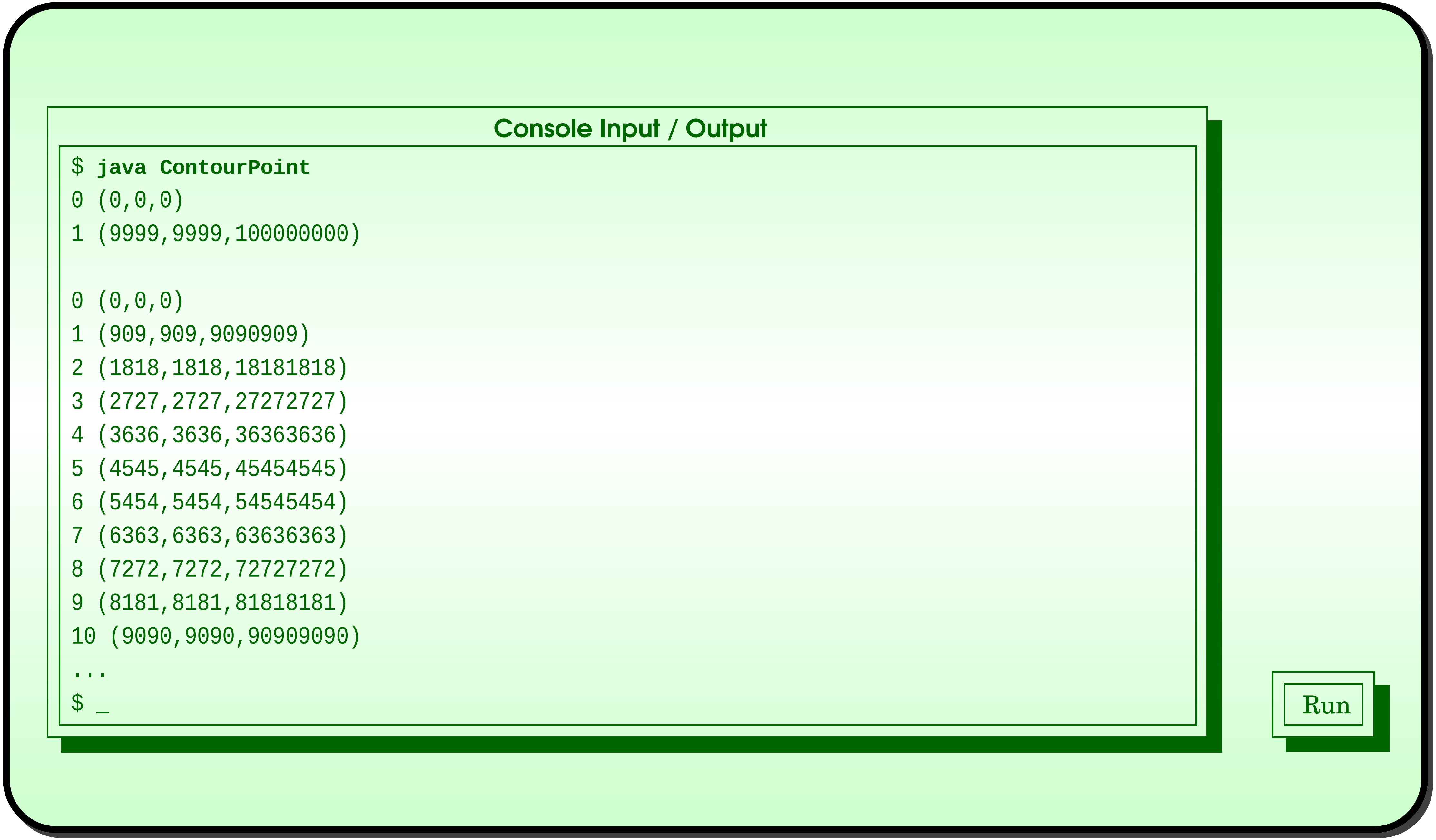
Contour points

```
      // Now print them out.
      for (int pointIndex = 0; pointIndex < length; pointIndex++)
        System.out.println(pointIndex + " " + pointArray[pointIndex]);
      System.out.println();
    } // for

    input.close();
  } // main

} // class ContourPoint
```

Trying it



```
$ java ContourPoint
0 (0,0,0)
1 (9999,9999,100000000)

0 (0,0,0)
1 (909,909,9090909)
2 (1818,1818,18181818)
3 (2727,2727,27272727)
4 (3636,3636,36363636)
5 (4545,4545,45454545)
6 (5454,5454,54545454)
7 (6363,6363,63636363)
8 (7272,7272,72727272)
9 (8181,8181,81818181)
10 (9090,9090,90909090)
...
$ _
```

Trying it

- Size matters?
  - **binary file** `test.dat` considerably smaller than if **data** stored as **text file**.
- Take **standard output**
  - strip off everything except text inside brackets
  - count **character**s
  - get approximation of minimum size needed to store as text.

Console Input / Output

```
$ java ContourPoint | cut -f2 -d"(" | cut -f1 -d ")" | wc -c
2135
$ ls -l test.dat
-rw-------  1 jtl jtl 941 Jul 01 19:12 test.dat
$ _
```

Run

Trying it

- Binary file less than half text file size
  - each **`short`** takes only two **byt**es
    - ∗ up to four as text
  - each **`int`** takes four bytes
    - ∗ instead of typical eight.
  - No separator byte between components of points nor between each point
    - ∗ because each component is fixed size.

Coursework: Saving greedy children

(Summary only)

Add features to some existing model **class**es so they can be written and read back from **binary file**s.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.