Input/Output (I/O)

Fri. Apr. 12, 2013

CS111 Computer Programming

Department of Computer Science
Wellesley College

What is File I/O?

A file is an abstraction for storing information (text, images, music, etc.) on a computer.

Today we will explore how to read information from (Input) files and write information to (Output) files in Java. Input/Output is often abbreviated I/O.

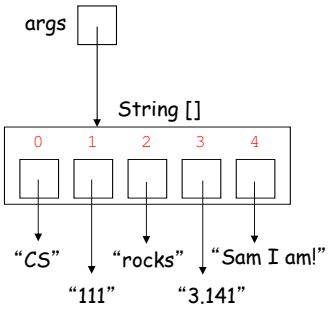
We will focus on files with textual information, but the techniques we'll learn generalize to any kind of information.

Writing to Standard Output

System.out.println writes a string to the Java console (standard output file).

```
public class WriteToConsole
{
   public static void main (String [ ] args) {
      for (int i = 0;  i < args.length; i++) {
          System.out.println(args[i]);
      }
   }
}
```

```
> java WriteToConsole CS 111 rocks 3.141 "Sam I am!"
CS
111
rocks
3.141
Sam I am!
```



Writing to a File

PrintWriter, File, and FileNoutFoundException live in the java.io package

File operations can generate FileNotFoundException

```
import java.io.*;

public class WriteToFile
{
   public static void main (String [ ] args) throws FileNotFoundException
   {
      PrintWriter writer = new PrintWriter(new File("commandArgs.txt"));
      for (int i = 0;  i < args.length; i++) {
         writer.println(args[i]);
      }
      writer.close();
   }
}
```

Append string to end of file.

Idiom for creating a file writer. If file doesn't exist, creates it. If file exists, overwrites it.

Finish any output to file. File is not guaranteed to contain all output until closed.

```
> java WriteToFile CS 111 rocks 3.141 "Sam I am!"
```

commandArgs.txt

```
CS
111
rocks
3.141
Sam I am!
```

What's An Exception?

- A way to say, "Something bad happened. I failed."
- When an unusual condition arises, a method stops its normal execution and **throws** an exception describing the problem.
- Allows a program to recover from a problem (robustness).

Declaring Exceptions in Method Headers

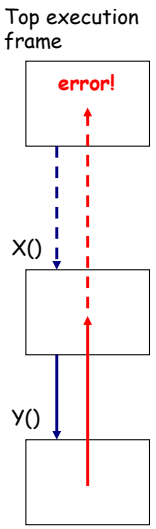
```
// Invoking a method that throws an exception
public void method_X () throws FileNotFoundException
{
    method_Y();  // method_Y() throws a FileNotFoundException
}
```


Exception propagation and handling

- If method X calls method Y and Y throws an exception, then X can **propagate** the exception.
- An exception that propagates all the way up to the top of the program results in an error message. E.g.

```
Exception in thread "AWT-EventQueue-0"
 BuggleException:  FORWARD: Can't move through wall!
at Buggle.forwardStep(BuggleWorld.java:2040)
at Buggle.forward(BuggleWorld.java:2034)
      ⋮
```

- If a method X explicitly throws an exception or even just propagates one, its header must include a **throws** clause.
- There are ways for methods to "catch" and handle exceptions (rather than propagating them), but we won't discuss those. (You'll learn these in CS230).



Reading Lines From a File

```
import java.io.*;   // Needed to use File and FileNotFoundException classes
import java.util.*; // Needed to use Scanner class

public class FileOps
{
   public static void displayFile (String name) throws FileNotFoundException
   { // Display the contents of a file, line by line
      Scanner reader = new Scanner(new File(name));
      while (reader.hasNextLine()){ // continue until we reach end of file
         System.out.println(reader.nextLine());
      }
      reader.close(); // Tidy up
   }
  // Other methods go here …
}
```

Idiom for creating a file reader. Throws a FileNotFoundException if file doesn't exist.

Check if there are more lines in file

Reads next line of input.

```
> FileOps.displayFile("list-elements.txt" )
CS
111
rocks
3.141
Sam I am!
```

list-elements.txt

```
CS
111
rocks
3.141
Sam I am!
```

Reading Integers From a File

```
import java.io.*;    // Needed to use File and FileNotFoundException classes
import java.util.*;  // Needed to use Scanner class

public class FileOps
{
   public static void sumIntsFromFile (String name) throws FileNotFoundException
   {  // Sum all the integers in a file
      Scanner reader = new Scanner(new File(name));
      int result = 0;
      while (reader.hasNextInt()) {  // continue until we reach end of file
         result = result +  reader.nextInt();
      }
      reader.close(); // Tidy up
      return result;
   }
   // Other methods go here …
}
```

Check if there are more integers in file

Reads next integer from file

```
> FileOps.sumIntsFromFile("ints.txt" )
71
```

ints.txt

```
17 3 5 8
10

9 12 7
```


Transforming a File

Let's write a Java class method that uppercases each line of a file (using the String toUpperCase() method).

```
> FileOps.upperCaseFile("cat4.txt", "cat4Upper.txt" )
```

input filename

output filename

cat4.txt

```
The sun did not shine.
It was too wet to play.
So we sat in the house
All that cold, cold, wet day.
```

cat4Upper.txt

```
THE SUN DID NOT SHINE.
IT WAS TOO WET TO PLAY.
SO WE SAT IN THE HOUSE
ALL THAT COLD, COLD, WET DAY.
```


upperCaseFile()

```
// Write to an output file the uppercase version of each line of an input file
public static void upperCaseFile(String inputName, String outputName)
        throws FileNotFoundException
{

   Scanner reader = new Scanner(new File(inputName));
   PrintWriter writer = new PrintWriter(new File(outputName));
   while (reader.hasNextLine()) {
      String line = reader.nextLine();
      writer.println(line.toUpperCase());
   }
   reader.close();  // Tidy up
   writer.close();  // Tidy up

}
```


Reading Input from the Java Console

The Java console (e.g., the Dr. Java Interaction Pane) can be viewed as a special kind of file (known as the **standard input/output file**).

We already know how to write to standard output using System.out.println().

We can read from it as well by creating a reader for it using this idiom:

```
Scanner reader = new Scanner(System.in);
```

Here's an example of interacting with the upperCaseInteractive method defined on the next slide:

```
> FileOps.upperCaseInteractive()
Type a line to convert to upper case (or quit to exit):
This is a test; it is only a test.
THIS IS A TEST; IT IS ONLY A TEST.
Type a line to convert to upper case (or quit to exit):
another example
ANOTHER EXAMPLE
Type a line to convert to upper case (or quit to exit):
quit
>
```

InteractiveUpperCase Application

```
// Interactively prompt the user for a string,
// which is then displayed in upper case form.
public static void upperCaseInteractive () throws FileNotFoundException
{
   Scanner reader = new Scanner(System.in);
   System.out.println("Type a line to upper case (or quit to exit):");
   String line = reader.nextLine();
   while (! line.equals("quit")) {
      System.out.println(line.toUpperCase());
      System.out.println("Type a line to upper case (or quit to exit):");
      line = reader.nextLine();
   }
   reader.close();  // Tidy up
}
```


Terminating interactive reads with Ctrl-D

```
// Version of upperCaseInteractive terminated by Ctrl-D,
// which acts as "end of file" for interactive input.
public static void upperCaseInteractive2 () throws FileNotFoundException
{
   Scanner reader = new Scanner(System.in);
   System.out.println( "Type a line to upper case (or Ctrl-D to exit): " );
   while (reader.hasNextLine()) { // Reads from standard input until Ctrl-D typed
      String line = reader.nextLine();
      System.out.println(line.toUpperCase());
      System.out.println( "Type a line to upper case (or Ctrl-D to exit): " );
   }
   reader.close();  // Tidy up
}
```

```
> FileOps.upperCaseInteractive2()
Type a line to convert to upper case (or Ctrl-D to exit):
Yet another test.
YET ANOTHER TEST.
Type a line to convert to upper case (or Ctrl-D to exit):

>
```

User types Ctrl-D here

Reading file lines into a list

```
> FileOps.linesFromFile( "list-elements.txt " )
[CS,111,rocks,3.141,Sam I am]
```

list-elements.txt

```
CS
111
rocks
3.141
Sam I am!
```


linesFromFile(): postpend version

```
// Return a String list where each element is a line from a given file.
// This version uses postpend to put the list elements in correct order.
public static StringList linesFromFile (String inputName)
        throws FileNotFoundException
{
   Scanner reader = new Scanner(new File(inputName));
   StringList ans = SL.empty();
   while (reader.hasNext()) {
      String line = reader.nextLine();
      ans = SLO.postpend(ans, line);
       // Note: postpend is very inefficient for long lists.
   }
   return ans;
}
```

linesFromFile(): prepend/reverse version

```
// Return a String list where each element is a line from a given file.
// This version uses postpend to put the list elements in correct order.
public static StringList linesFromFile (String inputName)
        throws FileNotFoundException
{
   Scanner reader = new Scanner(new File(inputName));
   StringList ans = SL.empty();
   while (reader.hasNext()) {
      String line = reader.nextLine();
      ans = SL.prepend(line, ans);
   }
   return SLO.reverse(ans); // Reverse needed to correctly order elts.
   // Note: prepending and (efficiently) reversing is way more
   // efficient than postpending.
}
```

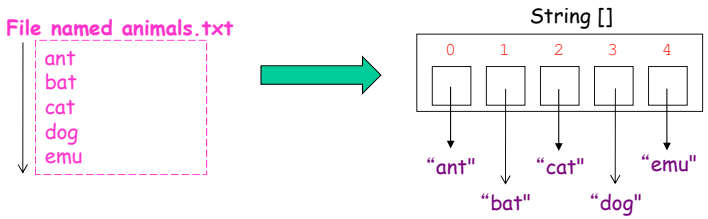

linesFromFile(): recursive version

```
// Return a String list where each element is a line from a given file.
// This version uses recursion to put the list elements in correct order
public static StringList linesFromFile (String inputName)
        throws FileNotFoundException {
    return linesFromScannerRec(new Scanner(new File(inputName)));
}

// Return a String list where each element is a line from a given Scanner.
public static StringList linesFromScannerRec (Scanner reader)
        throws FileNotFoundException {
  if (!reader.hasNext()) {
     reader.close(); // Tidy up
     return SL.empty();
  } else {
     String line = reader.nextLine();
     return SL.prepend(line, linesFromScannerRec(reader));
     // This version does the minimal number of prepends
  }
}
```


Problem: reading an array from a file

Given a text file, often want to read the lines into an array of strings:



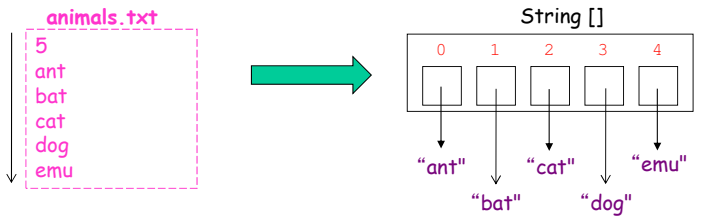
The fundamental problem is that we don't know in advance how many lines are in the file, so we don't know how big to make the array.

There are **lots** of ways to do this, but many are expensive, fragile, impractical, and/or inelegant.

We'll examine and compare a few strategies. This is a preview of the kind of issues considered in CS230.

Approach 1: include length header in file

It's easy to solve the problem if each file begins with a length header:



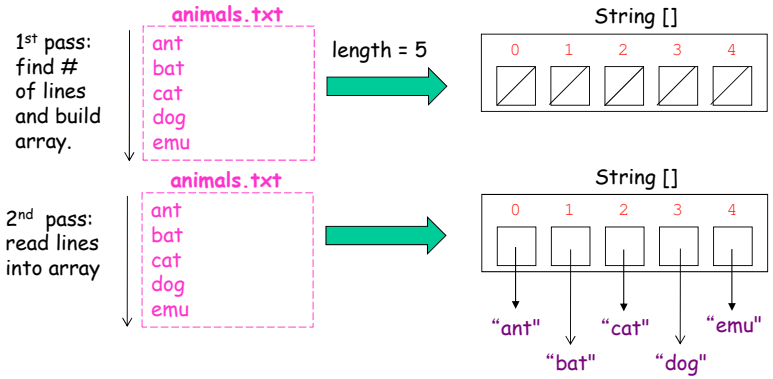
You saw this in Lab 11 and will see it again in PS9.

But this solution is impractical in general, because most existing files do not begin with length headers.

Furthermore, in many file-generating processes don't know the number of lines that will be in file when they begin.

Approach 2: read file twice

We can read the file twice: 1st time for length, 2nd time for lines:

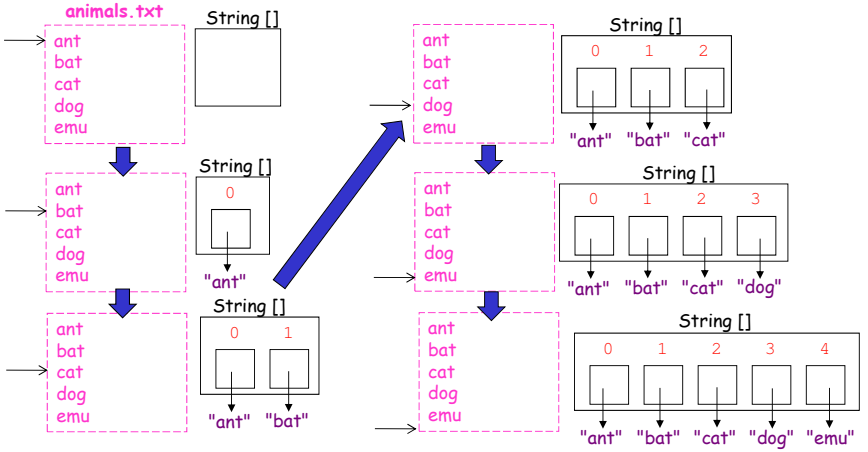


It's inelegant to read through file twice.

This approach doesn't work when file input is interactive (System.in) . Can't ask user to enter same inputs twice!

Approach 3: incrementally increase array size

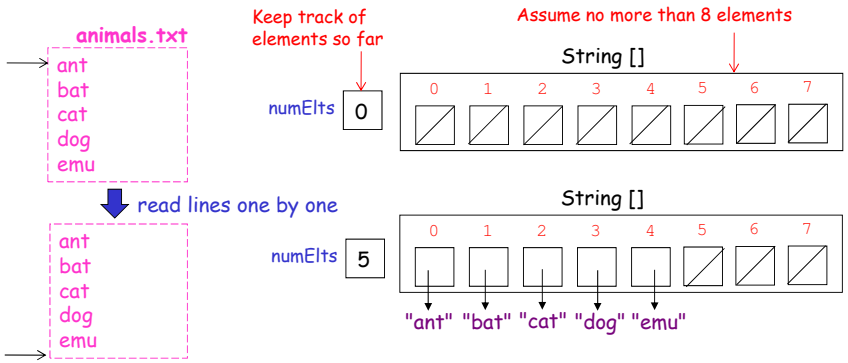
Start with an empty array. To add a new element, create a new array that's one element bigger, copy old elements to it, and insert new element.



Impractical! Copying costs are proportional to array size and become unacceptably high for large arrays.

Approach 4: assume maximum size

A common "quick and dirty" solution is to assume a maximum # of elements:

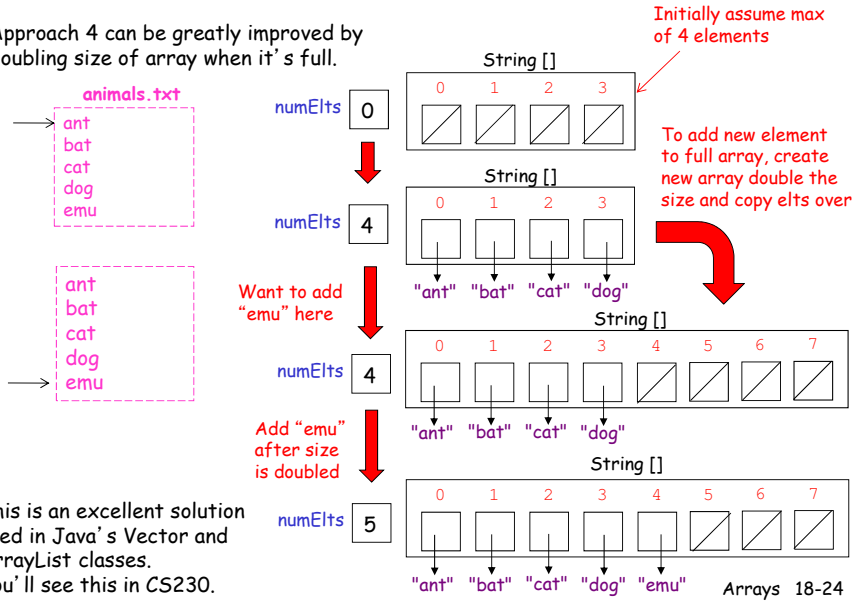


This "solution" is very fragile because it limits the number of lines that can be read. It's a terrible idea to hardwire arbitrary limits into a program.

It also wastes resources if maximum size is large. So it's really **unacceptable**!

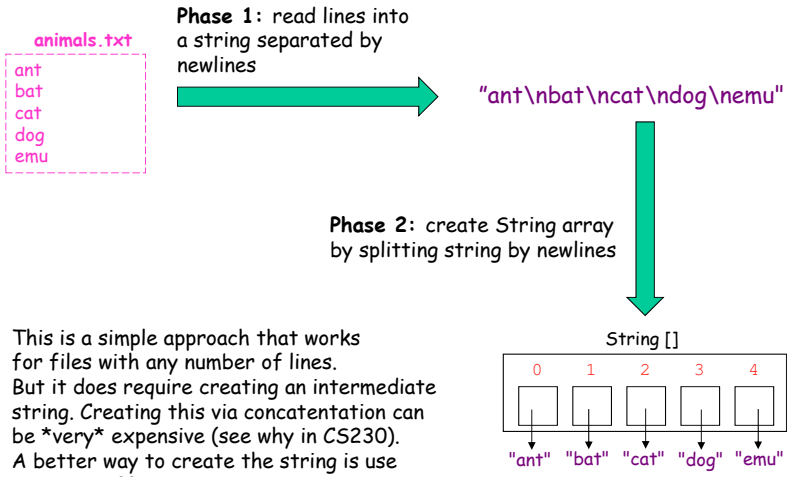
Approach 5: successive doubling

Approach 4 can be greatly improved by doubling size of array when it's full.



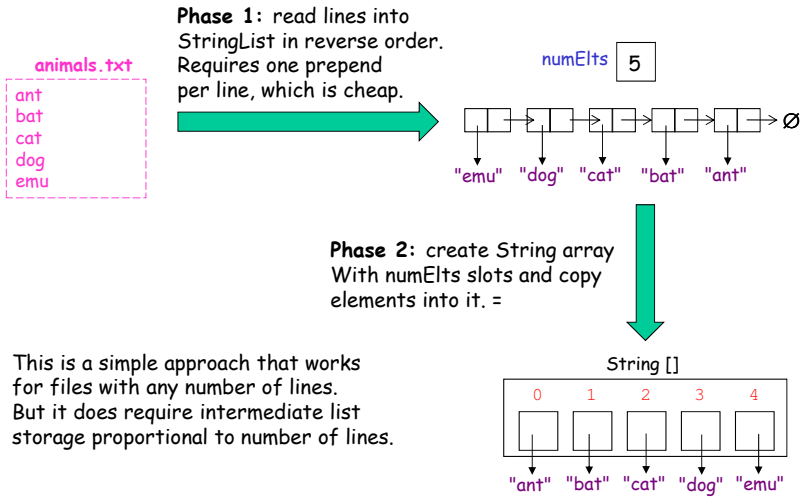
This is an excellent solution used in Java's Vector and ArrayList classes.
You'll see this in CS230.

Approach 6: split intermediate string



This is a simple approach that works for files with any number of lines. But it does require creating an intermediate string. Creating this via concatentation can be *very* expensive (see why in CS230). A better way to create the string is use a StringBuffer.

Approach 7: use intermediate list



This is a simple approach that works for files with any number of lines. But it does require intermediate list storage proportional to number of lines.

Summary of I/O Fundamentals

Output to Standard Output
System.out.print()
System.out.println()

Output to File
PrintWriter writer = new PrintWriter(new File(*someFileName*));
writer.print()
writer.println()

Input from File
Scanner reader = new Scanner(new File(*someFileName*));
Scanner reader = new Scanner(System.in); // Read from standard input
reader.hasNextLine()
reader.nextLine() // Returns a string
reader.nextInt() // Returns an int