Handout #53
Nick Parlante

Files

Text File

The simple "text file" is one of the oldest and simplest types of file. For that reason, text files are one of the most universally standard ways to store information. A text file is something you could type up in a word processor, and save with the "plain text" option. A text file is made of a sequence of characters, organized as a series of horizontal lines of text. There is no notion of bold or italic or color applied to the characters; they are just plain text characters, as we would store in a String. Historically, text files have been made of the familiar roman keyboard alphabet – abcdef..xyz012..89!@#$.. – with the extremely standard ASCII encoding (American Standard Code for Information Interchange). More recently, text has grown to include the idea of unicode characters like ø and π, but the encodings for those are not yet as standard as ASCII.

The .java files where we write our Java code are examples of text files. The text file is such a simple, flexible way to store information, it will be with us for a long time to come. If you want to store information in a simple, non-proprietary way, the text file is a great choice. XML files, which are a very modern way to store information, are a type of text file.

Text File Example

```
This is the first line of my 4 line text file,
and this here is the 2nd.

The above line is blank!
```

The above example text file is 4 lines long. Each line is made of plain characters and the end of each line is marked by an invisible "end-of-line" (i.e. EOL) character that signals the end of that line. For example, there is an EOL character just after the "file," in the first line, and after the "blank!" in the last line. The EOL character is essentially what you get when type the "return" key. The third line is blank which is valid. A blank line contains no characters other than the ending EOL. There are actually a few different EOL characters in use on different operating systems, but Java will shield us from that detail.

Text Reading Overview

The general topic of reading and writing files is known as "input/output" or just "i/o". We "open" a file to access its bytes, and read through its bytes in a "streamed" style, starting at the beginning of the file and reading through all the lines in order until reaching the end of the file. If we want to see a particular part of the file, we start at the beginning and read forward to the part we want. It is possible to read a file in more of a "random-access" style, but the streamed style is the most common.

Reading a file in Java is conceptually simple, however it involves a lot of syntactic overhead. First we will look at the main points of how it works, and then we will look at the full syntax.

We will use a BufferedReader object to read in a file's data. The BufferedReader object responds to a readLine() method that returns a String from the file. The first time the method is called, it returns the first line from the file. The next time it is called, it returns the second line from the file, and so on returning the next line from the file each time we call the readLine() method. The String does not include the EOL character; readLine() strips that out. Eventually, when we have gotten every line from the file, readLine() returns null to signal that there are no more lines.

For example, calling readLine() on a BufferedReader to read the text file shown above will return the String "This is the first line of my 4 line text file," the first time readLine() is called. The next call to readLine() will return "and this here is the 2nd.". The Third call will return "" (the empty string) since the line contains zero characters other than the EOL. The fourth call returns "The above line is blank!" and all subsequent calls to readLine() return null, signaling that we have reached the end of the file.

The strategy to read the whole file is to set up a while/if/break loop that calls readLine() again and again, exiting when readLine() returns null. We name the BufferedReader variable "in" since we are reading the data from the file in to memory.

Although it is not required, it is tidy to call close() on the file reader when we are done with it. This can close out the bookkeeping in the computer associated with the open file. The close() is optional since when a program exits, all of its open files are automatically closed. So if we do not close with the close() method, it will happen later anyway.

The code below shows all the key steps...

- Setting up the BufferedReader
- Calling readLine() in a loop to get each line. Do something with each line (compute something with it, print it, etc.)
- Detecting when readLine() returns null and exiting the loop
- Closing the reader when done reading

```
BufferedReader in = <set up BufferedReader object for the file we want>

while (true) {
  String line = in.readLine();  // Get the next line as a String, or null
                                // if there are no more

  if (line == null) { // exit the loop when there are no more lines
    break;
  }
  // Do something with the line (print in this case)
  System.out.println("Line from the file:" + line);
}
// get here when we have seen every line in the file

// Close() the file now that we are done with it  (optional)
in.close();
```

The above shows all the important parts of the file-reading loop. Now we look at the longer version with all the other syntax filled in.

Text Reading Full Example

The BufferedReader and other classes we need are in the "java.io" package, so we add an "import java.io.*;" statement at the top of our program.

Creating the BufferedReader is a two step process. First, we create a FileReader object, passing to its constructor the name of the file we want to open in the file system. Given a simple filename to read such as "file.txt", Java by default looks for a file with that name in the program's directory alongside the .java files.

The FileReader object knows how to read text files, but it does not have all the features we need. Therefore, we create a BufferedReader built on the FileReader, as shown below. The BufferedReader supports the readLine() method we want to use, and it does "buffering" which is a technique that uses a scratch area of memory to speed up the file reading process.

```
// Create FileReader, passing the name of the file we want to read
FileReader filer = new FileReader("file.txt");

// Create a BufferedReader based on the FileReader
BufferedReader in = new BufferedReader(filer);


// The above can be written all on one line like this...
BufferedReader in = new BufferedReader(new FileReader("file.txt"));
```

FileReader and BufferedReader work with files made of plain text. For non-text files (.jpeg, .mp3, ...) use FileInputStream instead. For 106A, we will only directly read text files. Also, there are ways that Java code can specify files in a directory other than the program directory, however we will not do that in CS106A. For simplicity, we will always position our data files in the program directory and rely on the simple default behavior.

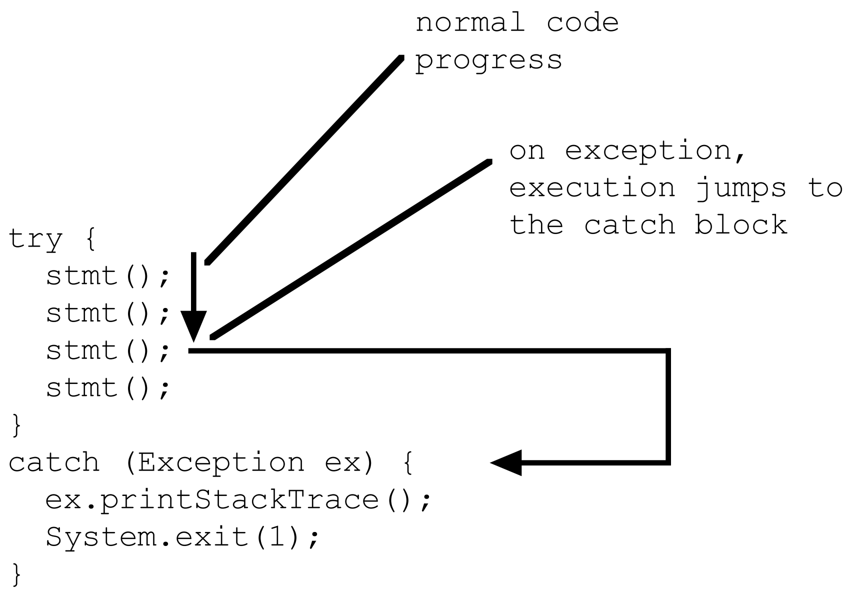
Exceptions

An exception occurs at runtime when a line of code tries to do something impossible such as accessing an array using an index number that is out of bounds of the array or dereferencing a pointer that is null. We have seen exceptions like that interrupt our programs many times as we debug our code.

We must deal with exceptions a little in our file-reading code since the code can run into some real error conditions, such as when the file want to read is not there. We will not explore all the features of exceptions. Instead, we will use exceptions in a simple, specific way in our file-reading code.

An exception halts the normal progress of the code and searches for some error handling code that matches the exception. Most often, the error handling code will print some sort of warning message and then possibly exit the program, although it could take some more sophisticated corrective action.

Java uses a "try/catch" structure to position error-handling code to be used in the event of an exception. The code to run goes in a "try" section, and it runs normally. If any line in the try section hits an exception at runtime, the program looks for a "catch-block" section for that type of exception. The normal flow of execution jumps from the point of the exception to the code in the catch-block. The lines immediately following the point of the exception are never executed.



```
try {
  stmt();
  stmt();
  stmt();
  stmt();
}
catch (Exception ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

For the file-reading code, some of the file operations such as creating the FileReader, or calling the readLine() method can fail at runtime with an IOException. For example, creating the FileReader could fail if there is no file named "file.txt" in the program directory. The readLine() could fail if, say, the file is on a CD ROM, our code is halfway through reading the file, and at that moment our pet parrot hits the eject button and flies off with the CD. The readLine() will soon throw an IOException since the file has disappeared midway through reading the file.

Our file-reading method will use a simple try/catch pattern for exception handling. All the file-reading code goes inside the "try" section. It is followed by a single catch-block for the possible IOException. The catch prints an error message using the built-int method printStackTrace(). The "stack trace" will list the exception at the top, followed by the method-file-line where it occurred, followed by the stack of earlier methods that called the method that failed.

In the example stack trace below, the method hide() in the Foo class has failed with a NullPointerException. The offending line was line 83 in the file Foo.java. The hide() method was called by main() in FooClient on line 23.

```
java.lang.NullPointerException
   at Foo.hide(Foo.java:83)
   at FooClient.main(FooClient.java:23)
```

In production code, the catch will often exit the whole program, using a non-zero int exit code to indicate a program fault (e.g. call System.exit(1)). Alternately, the program could try to take corrective action in the catch-block to address the situation. For CS106A, we will always just call printStackTrace(), and then let the program continue running as best it can with the lines after the try/catch. This can make it easier to examine things in the debugger after the exception. If no exception occurs during the run, the catch-block is ignored; the catch is only used in the event of an exception.

The method below shows the standard text-file-reading code: a while/readLine/break structure to read all the lines from a text file, all positioned inside a try/catch in case of errors during the read. In this case, the code in the loop body prints the lines of text as they are read, and counts the total number of lines and the number of blank lines. More generally, any desired operation for the text goes in the loop body. The outer try/while/catch code does not need to change.

```
   // Classic file reading code --  the standard while/readLine loop
   // in a try/catch.
   // This code prints all the lines, and counts the lines and blank lines
   public void echo(String filename) {
      try {
         // Create reader for the given filename
         BufferedReader in = new BufferedReader(new FileReader(filename));

         // While/break to send readLine() until it returns null
         int count = 0;
         int blanks = 0;
         while (true) {
            String line = in.readLine();

            if (line == null) {
               break;
            }
            // Here, do the desired operation on the lines from the
            // the file. This code counts the lines and blank lines,
            // and prints the text to the console.

            count++;
            if (line.equals("")) {
               blanks++;
            }
            System.out.println(line);
         }

         in.close();

         System.out.println("lines: " + count + " blank lines:" + blanks);
      }
      catch (IOException except) {
         // The code above jumps to here on an IOException,
         // otherwise this code does not run.
         // Good simple strategy: print stack trace, maybe exit
         except.printStackTrace();
         // System.exit(1);  // could do this too
      }

   }
```

There is perhaps some similarity between the file-reading code and the old for-all loop. In both cases, there is some standard syntax that goes on the outside (rather a lot of standard code in the file-reading case). Inside the loop, there is a place for the key code that we want to run for each iteration.

Files and ArrayList

A text file can contain any number of lines. An ArrayList can contain any number of elements. There is a natural synergy between these two, where we read all of the lines out of a text file and add them to an ArrayList. This is just a slight modification to the standard file-reading code above. We create the ArrayList before entering the loop. Inside the loop, we do an add() operation to add each String line to the list. At the end, we have built an ArrayList of Strings of the whole file.

```
   // Builds and returns an ArrayList containing all the lines of text
   // read from the given text file.
   public ArrayList readList(String filename) {
      ArrayList lines = new ArrayList();
      try {
         BufferedReader in = new BufferedReader(new FileReader(filename));
         while (true) {
            String line = in.readLine();
            if (line == null) {
               break;
            }
            // Add each line to the list
            lines.add(line);
         }
         in.close();
      }
      catch (IOException except) {
         except.printStackTrace();
      }
      return lines;
   }
```

Writing a Text File

Reading a text file is far more common than writing a text file, since on average programs seem to read more data than they produce (much program output is in the form of graphics and whatnot on screen instead of output text files).

The code to write a text file has the same structure as the code to read a text file. We create a PrintWriter that will create and write to the new file in the file system. As with reading files, if the filename is something like "file.txt", by default the file will be created in the program's directory. If there is an existing file with the same name, it will be overwritten (erased) by the new file. The PrintWriter is built on a BufferedWriter and a FileWriter, although that is just a detail.

What is important is that PrintWriter responds to the familiar methods print() and println() used by System.out. We write a loop that calls println() on the PrintWriter as needed to write the text lines we want to the file. We call close() when done writing, and all the i/o code is wrapped in a try/catch as usual.

```
  // Demonstrates standard code to write a text file.
  // Given a filename, writes lines 0, 1, 2, ...999
  // to a text file with that name.
  public void writeFile(String filename) {
    try {
      // Create a PrinterWriter for the file (built on BufferedWriter
      // and FileWriter).
      // (PrinterWriter responds to print() and println() like System.out)
      PrintWriter out = new PrintWriter(
         new BufferedWriter(new FileWriter(filename)));

      // Print the text we want to the file
      for (int i=0; i<1000; i++) {
        out.println(i);
      }

      out.close();
    }
    catch (IOException except) {
      except.printStackTrace();
    }
  }
```