CS211 Recitation 5: Reading from a file; URLs and links

Introduction

We assume that everyone has done some reading of a file in a earlier class, either in C++, Java, or Matlab.

Material on reading from files, read Sec. 2.6.3 of Weiss. For ProgramLive, read Sec. 5.7.2.

The core Java language has no facilities for input-output (I/O). Instead, all IO is handled by classes in the Java API.

There are two issues in handling reading (or writing) a file. First, how do we let the user tell us what file to read or write? Second, how do we hook up to that file and read or write it?

We handle the first one first:

Class JFileChooser

An instance of class JfileChooser (in package java.io) is associated with a dialog box on your monitor, which the user can navigate in order to choose a file. When the user has chosen the file, the dialog box closes and the program resumes. Here's the sequence of Java statements to do this:

```
JFileChooser jd = new JFileChooser();
jd.setDialogTitle("Choose input file");
jd.showOpenDialog(null);
File f = jd.getSelectedFile();
```

The first two statements are self-explanatory. The third one causes the dialog box to open; the statement finished executing only when the user closes the dialog box. When the third statement finishes, object jd contains information about the file the user selected. The fourth statement then retrieves an object that represents the file selected and stores it in variable j —**null** is stored in j if the user canceled the dialog without selecting a file. So we have to test f before using it as a description of a file.

Class File is also in package java.io.

That is all there is to the best way of obtaining a file from the user.

Input streams

Consider the following statement, assuming that f, of class File, contains a description of a file:

```
FileReader fr= new FileReader(f);
```

This expression associates a "file reader" fr with the file described by f. Using methods of object fr, we are able to read a character of the file at a time. This is too low-level for us, and we would rather be able to read a line at a time. Java provides a class BufferedReader for this purpose. Thus, execution of

```
BufferedReader br= new
    BufferedReader(new FileReader(f));
```

stores in br an object that can read a line at a time from file f. Evaluation of

```
br.readLine()
```

reads and returns the next line of file f, as a String, without the end-of-line character; if there is no next line, it returns **null**.

That's all there is to reading a file. On the last page, we give a program that gets a file from the user and then prints the length of each line.

Notice the **throws** clauses, which are needed because some IO exceptions are "checked".

Notice also the call br.close() when the file processing has finished. Always close a file in this fashion when finished with it.

Take a careful look at function getReader. Whenever you want to obtain a file from a user and create a BufferedReader to read from it, copy this method into your file and use it! That is the easiest way to deal with input from files.

Finally, look at the loop with initialization that process the input file. The first line is read outside the loop to truthify the invariant. This is needed because the file may be empty, in which case there is no first line.

URLs

URL stands for *uniform resource locator*. URLs are used on the internet to define files and the protocols with which they should be processed. Here's an example of a URL:

http://www.cs.cornell.edu/Courses/cs211/2001fa/index.html

It consists of

1. An identification of a service or protocol (http);
2. A domain name or host (www.cs.cornell.edu), which is associated with a computer that is attached to the internet; and
3. A path on that computer (Courses/cs211/2001fa/index.html).

A URL can have other components; here is the general form:

<protocol>://<authority><path>?<query>#<fragment>

We won't discuss or use the authority or the query.

Protocols

Here are the protocols one usually sees:

1. http: This stands for HyperText Transport Protocol, which is the most used protocol for accessing files over the internet. Generally, the files are html files, but they could be any files, including text files, .jpg files, and .gif files. A domain name must be given.
2. file: This is used for a file that is on the local computer. The domain name is generally not present.
3. ftp: This stands for File Transfer Protocol, a protocol that provides for files and directories of files to be transferred from one computer to another.
4. mailto: Used to bring up a window that one can use to send mail. In this case, the // and domain name are omitted, and instead of a path one has an email address, e.g. consider this URL: mailto:gries@cs.uga.edu

Domain names, or hosts

The appearance of "//" in the URL signals the beginning of a domain name, which is a name that has been registered as being assigned to or associated with a particular computer. Here are examples of domain names:

1. www.cs.cornell.edu
2. www.cs.toronto.edu
3. www.datadesk.com

It used to cost about $75 per year to register and maintain a domain name. It's a lot less now. Here is a link to the web page (http://www.icann.org/) for the company makes up rules for domain names and registers them. The domain-name part of a URL is often called the "host" —that name is used later, so remember it.

Ports

A URL can optionally specify a "port", which is the port number to which the TCP connection is made on the remote host machine. If the port is not specified, the default port for the protocol is used instead. For example, the default port for http is 80. An alternative port could be specified as:

http://www.ncsa.uiuc.edu:8080/demoweb/url-primer.html

We won't deal with ports, since they are usually not given in the URLs that we deal with.

Absolute paths

The path part of a URL is a path on the computer to the file that the URL describes. For example,

/~gries/Logic/Introduction.html

indicates that the hard drive of the computer whose domain name was given has a folder (directory) ~gries; in that folder is a folder named Logic, and in that folder is a file named Introduction.html.

The character / is used to separate entities on the path, regardless of the operating system that is running on the computer --Unix-like, Windows, or Macintosh.

If the file name is missing at the end (so that the last entity is a folder), then a default file is chosen, usually index.html or index.htm. This default depends on (and can be changed) the computer on which the file resides. On some computers, we have seen the following defaults: default.html, default.htm, home.html, and home.htm.

Any path of any file or folder on your hard drive can be used in a URL. The form of the beginning of such a path depends on whether a Unix-like, Windows, or Macintosh operating system is being used. You can check this out on your own computer by loading any html file that is on your hard drive into a browser like Netscape Communicator or Internet Explorer and looking at the URL that is displayed. Here are examples for the three kinds of systems:

1. Unix: /home/profs/gries/public_html/index.html
2. Windows: /C:/MyDocuments/test.html
3. Mac: /ProgramLive/Course/IC/web/ICweb.htm

Relative URLs

Within an html file, one can have a relative URL, as in

href="people/faculty/faculty.htm".

The protocol and host are assumed to be the same as that of the current html file, and the path is assumed to be relative to the folder in which the html file appears. One can use ".." in the path to move up in the path of folders, as in all operating systems. For example, if the current folder is /~gries/Logic, then relative path ../NoLogic/test.html refers to the file /~gries/NoLogic/test.html.

Fragments

The following URL has the "fragment" #chap1:

http://java.sun.com/index.html#chapter1

The fragment is often the name of a "target" within the file given by the URL, but there are other uses for it. Technically, the fragment is not part of the URL. For assignments in CS211 this Fall, targets should be discarded if present.

Class URL

Package java.net contains class URL, which is used for dealing with URLs in Java. Below, we provide terse specs for a few of the most useful methods in class URL. We don't discuss the authority, query, and fragment parts of URLs. We also make the following comments.

First, if you have a variable of class URL, you can get a BufferedReader to it using the following procedure —this procedure returns **null** if the file URL is not for an http file or a file file (a file on a local computer). Notice how an

OpenStream is created for it (returns one byte at a time), then an
InputStreamReader (which turns bytes into characters) from that, and finally the
BufferedReader (which had a method to read one line at a time).

This is quite similar to dealing with input files on your computer.

```
    /**  = a reader for URL url (which must not be
        null). If the protocol is not http or file, null is
        returned. */
    private BufferedReader getReader() {
        if (!protocol.equals("http") &&
           !protocol.equals("file")) {
          return null;
        }

        try {
          InputStream is= url.openStream();
          InputStreamReader   isr=
                        new InputStreamReader(is);
          return new BufferedReader(isr);
        } catch (IOException e) {
              return null;
        }
  }
```

When you have an absolute Url as a String s, use "**new** URL(s)" to create an instance of URL for it. But if you already have a URL c that is a directory, say, and you want to create a URL for a file s that is within the directory, use "**new** URL(c,s)".

Finally, given a URL u, use the various getter methods shown below to get components of it, usually as Strings.

```
public class URL {
   /** Constructor:  a URL object constructed its
        representation s. A call URL("...")
        is equivalent to a call URL(null, "..."). */
   public URL(String s)
             throws MalformedURLException

   /* Constructor:  a URL object constructed from
         URL c and String s.
        If c's path component is empty and the
        scheme, authority, and query components are
        undefined, then the new URL is a reference to
        the current document. Otherwise, any
        fragment and query parts present in the s are
        used in the new URL.

        If s contains a protocol component is and it does
        not match the protocol in c, then
        the new URL is created as an absolute URL
         based on the s alone. Otherwise, the protocol
         component is inherited from the context URL.

         If the s's path component begins with a slash
        character "/" then the path is treated as
         absolute, and s's path replaces the c's path.
         Otherwise s's path is treated as a relative
         path and is appended to c's path. The path is
         canonicalized through the removal of
         directory changes made by occurences of ".."
         and ".".
   */
   public URL(URL c, String s)
          throws MalformedURLException

   /* = the file name for this URL */
   public String getFile()

   /* = the host, or domain name, for this URL */
   public String getHost()

   /* = the path for this URL */
   public String getPath()

   /* = the port for this URL (-1 if port is not set) */
   public int getPort()

   /* = the protocol for this URL */
   public String getProtocol()

   /* = a newly opened connection to this URL, as an
        InputStream. Shorthand for
          openConnection().getInputStream()  */
   public final InputStream openStream()
                  throws IOException

   /*  =  a representation of this URL */
   public String toString()
}
```

```
import java.io.*;
import javax.swing.*;
/** Illustrate use of class JFileChooser and reading a file */
public class FileChooserApp {
  /** print the lengths of the lines on a file chosen by the user. */
  public static void main(String[] args) throws IOException {
    BufferedReader br= getReader(); // A link to the user’s file
    if (br == null) {
      return;
    }
    // Read file br and print the length of each line
    String s = br.readLine();
    // { inv: s is last line read and lengths of lines before line s have been printed }
    while (s != null) {
      System.out.println(s.length());
      s = br.readLine();
    }
    br.close();
  }
  /** Obtain a file name from the user, using a JFileChooser, and return a reader that
    that is linked to it. If the user cancels the choice, return null */
  public static BufferedReader getReader() throws IOException {
    JFileChooser jd = new JFileChooser();
    jd.setDialogTitle("Choose input file");
    jd.showOpenDialog(null);
    File f = jd.getSelectedFile();
    if (f == null) {
      return null;
    }
    return new BufferedReader(new FileReader(f));
  }
}
```