IMDB Data Set
Topics: Parsing Input using Scanner class

Atul Prakash

IMDB Data Set

- Consists of several files:
  - movies.list: contains <movies, year>
  - actors.list: contains <actor, list of movies>
  - actresses.list: <actress, list of movies>
- aka-titles.list: <title, aliases of the title>
- Bunch of other files, including other crew, fan ratings, awards, running times, etc.

Example Queries

- List of movies released in a given year
- Search for movies, given keywords
- Find actors in a given movie (requires actors.list)
- Find actresses who have acted with a particular actor, e.g., Woody Allen (requires actresses.list and actors.list)

Some References

- IMDB dataset itself in plain-text downloadable files (download just the movies.list.gz for now, and uncompress it).
  - http://www.imdb.com/interfaces
- An example report of some analysis of the IMDB dataset (glance through it):
  - http://had.co.nz/data/movies/description.pdf
- A research paper on correlating Netflix dataset with IMDB to break anonymity of reviewers in Netflix (optional read -- shows an example of privacy risks):
  - http://www.cs.utexas.edu/~shmat/shmat_oak08netflix.pdf

Sample Contents (movies.list)

Format: <unique title, year>

Titles made unique by including the year of the release. Also, articles written after comma. E.g., "The Godfather" is written as "Godfather, The".
Annotations after the title:
(V): direct video release
(TV): made for TV
(VG): video game

More details on title formatting:
http://www.imdb.com/help/show_leaf?titleformat

'Doctor Who': The Tom Baker Years (1991) (V) 1991
'Doctor Who': The Troughton Years (1991) (V) 1991
'Doctor Who': Then and Now (1987) (TV) 1987
'Doctor Who': Thirty Years in the Tardis (1993) (TV) 1993
'Doctor Zhivago': The Making of a Russian Epic (1995) (TV) 1995
'Dog Day Afternoon': After the Filming (2006) (V) 2006
'Dog Day Afternoon': Casting the Controversy (2006) (V) 2006
'Dog Day Afternoon': Recreating the Facts (2006) (V) 2006
'Dog Day Afternoon': The Story (2006) (V) 2006
'Don't Talk' (1942) 1942
'Donnie Darko': Production Diary (2004) (V) 2004
'Dr. Who': Daleks - The Early Years (1993) (V) 1993
'Duel': A Conversation with Director Steven Spielberg (2004) (V) 2004
'Dune': Models and Miniatures (2006) (V) 2006
'Dune': Special Effects (2006) (V) 2006
'Dune': Wardrobe Design (2006) (V) 2006
'E' (1981) 1981

…

Dune (1973) 1973
Dune (1984) 1984
Dune (2010) 2010
Dune 2000 (1998) (VG) 1998
Dune 7 (2002) 2002
Dune Buddies (1978) 1978
Dune Bug (1969) 1969
Dune II: The Building of a Dynasty (1992) (VG) 1992
Dune Surfer (1988) 1988
Dune Warriors (1990) 1990
Dunechka (2004) 2004
Dunera Boys, The (1985) (TV) 1985
Dunes of Destiny (2005) 2005
Dung che sai duk (1994) 1994

Reading in Data

- Several classes in Java to help parse and read input:
  - String.split methods
  - Scanner class
  - StringTokenizer class
  - Pattern and Matcher classes
- Reference with examples:
  - http://www.javapractices.com/topic/TopicAction.do?Id=87
  - Or Google search for:
    - Java practices Parse text input

Example

- Sample input file

```
height = 167cm
mass =  65kg
disposition =  "grumpy"
this is the name = this is the value
```

- Format: key = value

- Strategy: read line by line. Use "=" as a seperator to extract key and value. Store in a list.

Scanner Class

- Scanner class is very powerful for string parsing and may be all you need in many cases
  - For examples illustrated here, see: http://www.javabeat.net/tips/24-parsing-input-using-scanner.html

```
import java.util.Scanner;   // import Scanner class from java.util package
public class ScannerDemo {
    public static void example1() {
        Scanner scanner = new Scanner("EECS 282 is a great class");
        while (scanner.hasNext()){
            System.out.println(scanner.next());
        }
    }
    public static void main(String[] args) {
        example1();
    }
}
```

```
EECS
282
is
a
great
class
```

Field-separated values

```
public static void stringScanningWithSeparator() {
    Scanner scanner = new Scanner("George Washington, President, born 1732");
    scanner.useDelimiter(",");
    while (scanner.hasNext()) {
        System.out.println(scanner.next()); // will print 3 items.
    }
}
```

```
George Washington
 President
 born 1732
```

Here, comma is used as a separator. Can use any string as a separator, e.g., "\t" (tab)

Default seperator: whitespace

What does the following print?

```
public static void stringScanningWithSeparator() {
    Scanner scanner = new Scanner("George Washington, President, born 1732");
    // scanner.useDelimiter(",");
    while (scanner.hasNext()) {
        System.out.println(scanner.next()); // will print 3 items.
    }
}
```

Are spaces printed?

How are commas treated?

Reading Typed Values

```
public static void scanInts() {
    Scanner scanner = new Scanner("1 3");
    int first = scanner.nextInt();
    int second = scanner.nextInt();
    System.out.println("first = " + first + "; second = " + second);
}

Output:
first = 1; second = 3
```

Similar methods: nextBoolean, nextByte, nextDouble, etc. To read String, just use next()

Checking Input Type Before Reading

- Use hasNext*Type* method, where *Type* is the type being read, e.g., hasNextInt, hasNextBoolean, hasNext, hasNextLine

```
public static void scanTypes() {
    Scanner scanner = new Scanner("1 3 5 7 9 11 abc 13 def 15");
    while (scanner.hasNextInt()){
        System.out.println(scanner.nextInt());
    }
}
```

```
1
3
5
7
9
11
```

hasNextInt() returns false on "abc"

Reading Files

- Scanner s = new Scanner(FileObject)
  - Creates a scanner on a file object
- Examples of file objects:
  - System.in: standard input, by default, terminal input
  - File in = new File(*filepath*);

Example

```
public static void scanLinesFromFile() {

    File in = new File("/Users/aprakash/Documents/282/workspace/282/input.txt");
    Scanner scanner = new Scanner(in);
    int linenum = 1;
    while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        System.out.println(linenum + ": " + line);
        linenum++;
    }
    scanner.close();  // important to close scanners on files.
}
```

File Paths on Windows

```
// parse a file containing lines with part1,part2
// For windows, you would do something like
File in = new File("C:\\282\\input.txt");
```

Two backslashes needed because "\" is also used in other ways, e.g., \n, \t. It must be "escaped" with another backslash.

```
// or you can use forward-slashes. Java will convert / to \\:
File in = new File("C:/282/input.txt");
// You can also use relative path names, e.g., "282/input.txt".
// In Eclipse, go to Run->Run Configurations...->Arguments to
// set current directory. Relative paths are with respect to the current
// directory.
```

Closing files

- scanner.close() is important. It closes the scanner object, as well as the file bound to it. Otherwise, file remains open and consumes an operating system resource.
- Always close files when done reading them.

Exceptions in I/O

- Unfortunately, Eclipse gives an error in the above code about an uncaught exception: FileNotFoundException

Reason

```
File in = new File("/Users/aprakash/Documents/282/workspace/282/input.txt");
Scanner scanner = new Scanner(in);
```

What if the file does not exist?
Unlike run-time errors (e.g., divide by 0), Java requires you to handle I/O exceptions

Declaring the thrown exception

```
public static void scanLinesFromFile() throws FileNotFoundException {
    // parse a file containing lines with part1,part2
    // For windows, you would do something like
    // File in = new File("C:\\282\\input.txt");
    // or you can use forward-slashes on Windows. Java will convert:
    // File in = new File("C:/282/input.txt");
    // You can also use relative path names, e.g., "282/input.txt".
    // In Eclipse, go to Run->Run Configurations...->Arguments to set current directory)
    File in = new File("/Users/aprakash/Documents/282/workspace/282/input.txt");
    Scanner scanner = new Scanner(in);
    int linenum = 1;
    while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        System.out.println(linenum + ": " + line);
        linenum++;
    }
    scanner.close();  // important to close scanners on files.
}
```

Alternative -- Catch Exceptions

```
public static void scanLinesFromFile2()  {
    File in = new File("/Users/aprakash/Documents/282/workspace/282/input.txt");
    Scanner scanner;
    try {
        scanner = new Scanner(in);
    } catch (FileNotFoundException e) {
        System.out.println("File not found");
        return;
    }
    int linenum = 1;
    while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        System.out.println(linenum + ": " + line);
        linenum++;
    }
    scanner.close();  // important to close scanners on files.
}
```

Summary

- Scanner class is useful for reading files and parsing them.
- Other useful methods for parsing: String class functions. E.g., trim() method, lastIndexOf(String), substring(int start, int end), and split() methods.