Files

CSE 1310 – Introduction to Computers and Programming

Vassilis Athitsos
University of Texas at Arlington

Files and Text Files

- A file is a collection of data, that is saved on a hard drive.
- A text file contains text (as opposed to images, video, songs, etc.)
- Every file is uniquely identified using two attributes:
  - The file name.
  - The file path.

What You Should Know

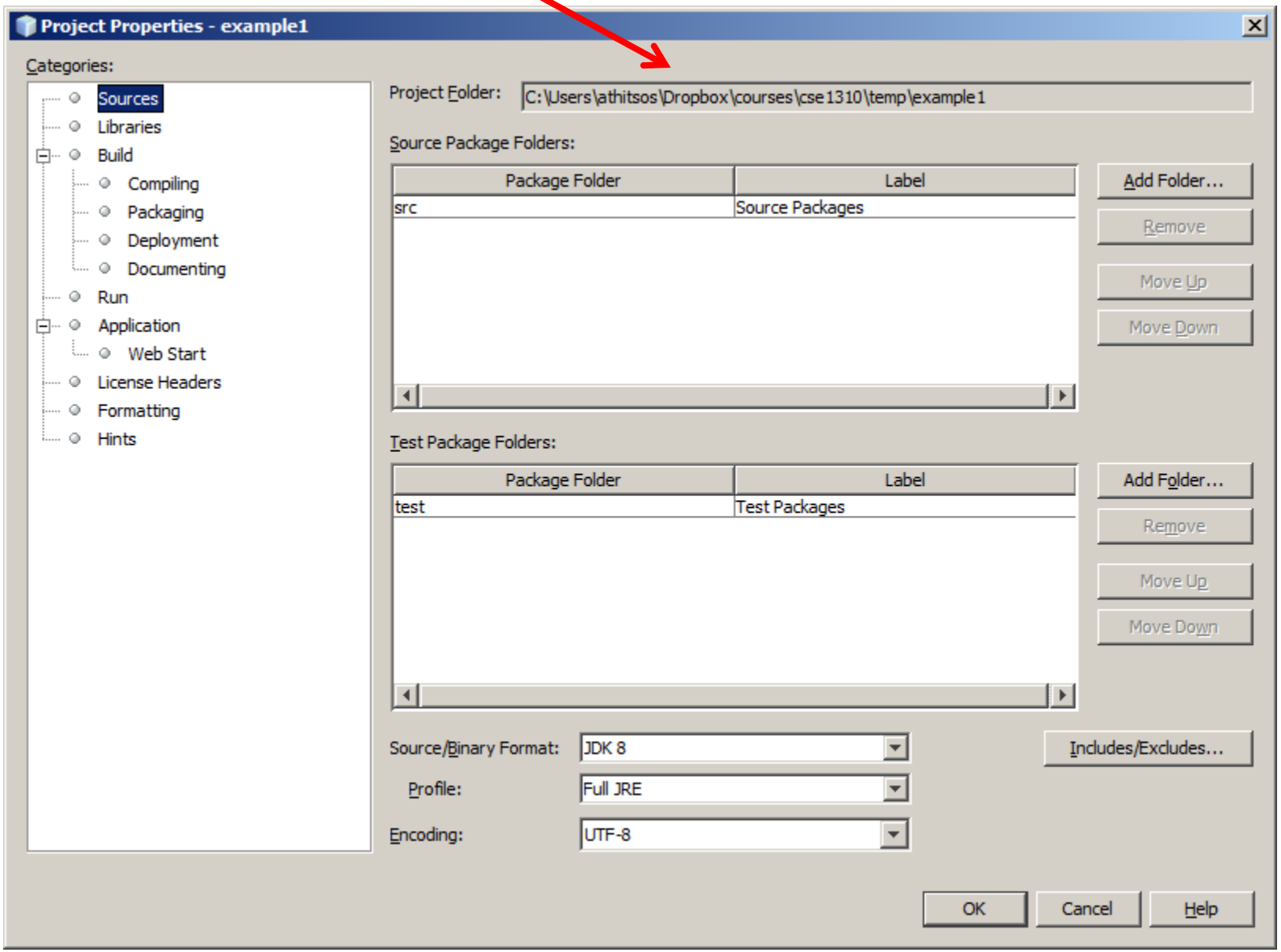
- You are expected to know:
  - How to create a text file.
  - How to edit a text file, to put there the text that you want.
  - How to store a text file in the folder that you want.
  - How to figure out the path of the folder where a file is stored.
- If you do not know how to do these things, talk to the instructor or the TA.

Making Files Visible to Netbeans

- From Netbeans:
  - Click on File->Project Properties.
  - At the top, see where the project folder is.
  - In order for your Java code to see a text file, **<u>you should put the text file on the project folder</u>**.

IF YOU DO NOT FOLLOW THESE STEPS, YOUR FILE-RELATED PROGRAMS WILL NOT WORK.

Folder where your project is saved:



Motivation for Files: Phonebook

- Consider a phonebook application, that allows:
  - Making a new entry (new name and phone number).
  - Modifying an existing entry.
  - Deleting an entry.
  - Looking up the phone given the name.
  - Looking up the name given the phone.
- What can we do and what can we not do, using what we have learned so far?

Motivation for Files: Phonebook

- Consider a phonebook application, that allows:
  - Making a new entry (new name and phone number).
  - Modifying an existing entry.
  - Deleting an entry.
  - Looking up the phone given the name.
  - Looking up the name given the phone.
- We can do all five things listed above. However, at the end of the program, all information vanishes.
- Files provides a solution:
  - data can be saved into files before the program exits.
  - data can be read again from those files when needed.

Motivation for Files: Spreadsheets

- Spreadsheets are one of the most important computer applications.
- Spreadsheets organize data in tables, consisting of rows and columns.

Spreadsheet Example: Class Enrollments

Date	1310-006	4308-001	5360-001
7/14/2015	33	41	21
7/17/2015	41	41	20
7/24/2015	50	42	18
7/29/2015	57	41	19
8/2/2015	58	41	18
8/5/2015	67	41	19
8/7/2015	72	41	17

- An enrollment spreadsheet shows how many students are enrolled each day in each course section.
- Such a spreadsheet typically includes tens of rows (dates) and hundreds of columns (courses).

Spreadsheet Example: Class Enrollments

```
Date,1310-006,4308-001,5360-001
7/14/2015,33,41,21
7/17/2015,41,41,20
7/24/2015,50,42,18
7/29/2015,57,41,19
8/2/2015,58,41,18
8/5/2015,67,41,19
8/7/2015,72,41,17
```

- Here is the table from the previous slide, saved as a text file.
- This is a CSV file.
- CSV stands for "comma-separated values".
- Such files are really easy to read by computer programs.
  - You will soon write such programs.

Motivation for Files: Large Text Processing

- Suppose that we have to write a program that:
  - takes a book (or a set of books) as an input.
  - identifies the most frequent words in that book or set of books.
- Can you think of example applications for such a program?

Motivation for Files: Large Text Processing

- Suppose that we have to write a program that:
  - takes a book (or a set of books) as an input.
  - identifies the most frequent words in that book or set of books.
- Can you think of example applications for such a program?
  - identifying the most important words to learn in a foreign language.
  - identifying the language in which a book was written.
  - identifying and comparing style of different authors, newspapers, centuries, etc.

Motivation for Files: E-mails, Web Pages

- E-mails and web pages are stored as files.
- Programs are needed to:
  - Create such files, and store data appropriately.
  - Read such files, and present data to the user.

Motivation for Files: Images, Video, Songs

- This is beyond the scope of this course, but: Photographs, movies, songs, are all stored in files.
- Programs are needed to read such files and interpret the data:
  - Display photos.
  - Play movies.
  - Play songs.
- Programs are also needed to create such files (store images, video, songs in files).

File Format

```
Date,1310-006,4308-001,5360-001
7/14/2015,33,41,21
7/17/2015,41,41,20
7/24/2015,50,42,18
7/29/2015,57,41,19
8/2/2015,58,41,18
8/5/2015,67,41,19
8/7/2015,72,41,17
```

- The file format specifies how to interpret the data.
- In some cases, we need to know the format in advance, in order to to understand how to process the data.
- In the above example (from the enrollments spreadsheet), the CSV format is followed.

File Format

```
Date,1310-006,4308-001,5360-001
7/14/2015,33,41,21
7/17/2015,41,41,20
7/24/2015,50,42,18
7/29/2015,57,41,19
8/2/2015,58,41,18
8/5/2015,67,41,19
8/7/2015,72,41,17
```

```
Date_1310-006_4308-001_5360-001
7/14/2015_33_41_21
7/17/2015_41_41_20
7/24/2015_50_42_18
7/29/2015_57_41_19
8/2/2015_58_41_18
8/5/2015_67_41_19
8/7/2015_72_41_17
```

- On the right, we see the same data stored in another format (columns separated by _ , i.e., the underscore character).
- We need to know the format in advance, in order to to understand how to process the data shown above.

First Example

- This is our first example of a program reading a file.
- Not much processing of data happens here.
- The code simply reads lines of text and prints them.
- Note that you need to import java.io.File.

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Steps in Reading a File

- First step: **<u>opening a file</u>**. Two substeps:
  - Create a File object.
  - Create a Scanner object.
- We have been using scanner objects for user input.
- Scanner objects created as shown can be used to read data from files.

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Steps in Reading a File

- Creating the scanner object for a file has to be fine using **try ... catch**.
- Why?

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Steps in Reading a File

- Creating the scanner object for a file has to be fine using **try ... catch**.
- Why? Because something could go wrong, and the program would crash.
- For example, the filename could be wrong.

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Steps in Reading a File

- Second step: reading the data.
- There are a lot of ways to do this.
- However, to simplify, we will only use two methods:
  - **hasNextLine()**: check if there is more data to read.
  - **nextLine()**: read the next line of data.

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Steps in Reading a File

- Second step: reading the data.
- Notice that we read the data using a while loop.
  - **hasNextLine()** will return false when there are no more lines to read from the file.

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Steps in Reading a File

- Third step: process the data.
- This task depends on what you want to do.
- In this simple example, we just print the data.

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Steps in Reading a File

- Fourth step: close the file.
- Use the **close()** method.

```
import java.util.Scanner;
import java.io.File;

public class enrollments {
  public static void print_file(String filename)
  {
    File temp = new File(filename);
    Scanner input_file;
    try
    {
      input_file = new Scanner(temp);
    }
    catch (Exception e)
    {
      System.out.printf("Failed to open file %s\n",
                        filename);
      return;
    }

    while(input_file.hasNextLine())
    {
      String line = input_file.nextLine();
      System.out.printf("%s\n", line);
    }
    input_file.close();
  }

  public static void main(String[] args)
  {
    print_file("enrollments.csv");
  }
}
```

Reading a File: Summary

- First step: open the file.
  - Make sure you know how to create a **File** object.
  - Make sure you know how to create a **Scanner** object from a **File** object.
- Second step: read the data.
  - Make sure you know how to use **hasNextLine()** and **nextLine()**.
- Third step: process the data.
  - This is task-dependent, you do whatever is needed.
- Fourth step: close the file.
  - Make sure you know how to use the **close()** method.

Storing File Contents in a Variable

- Many times we need to store the data of the entire file in a variable, to do more processing.
  - For example, to edit a spreadsheet.
- We can store the data from the entire file in an **array list of strings**.
  - A string for each line of text in the file.
- Why an array list and not an array?

Storing File Contents in a Variable

- Many times we need to store the data of the entire file in a variable, to do more processing.
  - For example, to edit a spreadsheet.
- We can store the data from the entire file in an **array list of strings**.
  - A string for each line of text in the file.
- Why an array list and not an array?
  - First, because we don't know from the beginning how many lines the file has.
  - Second, because this way we can insert more lines if we want (for example, to add data to a spreadsheet).

Storing File Contents in a Variable

- Many times we need to store the data of the entire file in a variable, to do more processing.
  - For example, to edit a spreadsheet.
- We can store the data from the entire file in an **array list of strings**.
  - A string for each line of text in the file.
- Let's write a function **read_file** that:
  - Takes as input a filename.
  - Returns an array list of all the lines of text on the file.

```java
public static ArrayList<String> read_file(String filename)
 {
   File temp = new File(filename);
   Scanner input_file;
   try
   {
     input_file = new Scanner(temp);
   }
   catch (Exception e)
   {
     System.out.printf("Failed to open file %s\n",
                       filename);
     return null;
   }

   ArrayList<String> result = new ArrayList<String>();
   while(input_file.hasNextLine())
   {
     String line = input_file.nextLine();
     result.add(line);
   }

   input_file.close();
   return result;
 }
```

The solution is almost identical to our print_file function.
The new lines are shown in red.

Using the **read_file** Function

- Let's rewrite function **file_print**, so that it uses the **read_file** function that we just wrote.

```
public static void print_file(String filename)
 {
   ArrayList<String> lines = read_file(filename);
   if (lines == null)
   {
     return;
   }
   int length = 0;
   for (int i = 0; i < lines.size(); i++)
   {
     String line = lines.get(i);
     System.out.printf("%s\n", line);
   }
 }
```

A Second Example: Length of a File

- Let's write a function **file_length** that:
  - Takes as input a filename.
  - Returns the number of characters in that file.
- Hint: use the **read_file** function that we already have.

A Second Example: Length of a File

```
public static int file_length(String filename)
{
  ArrayList<String> lines = read_file(filename);
  if (lines == null)
  {
    return 0;
  }

  int length = 0;
  for (int i = 0; i < lines.size(); i++)
  {
    String line = lines.get(i);
    length += line.length();
  }

  return length;
}
```

A Third Example: Counting Words

- Let's write a function **count_words** that:
  - Takes as input a filename.
  - Returns the number of words in that file.
- Question: how can we count the number of words in a string of text?

A Third Example: Counting Words

- Let's write a function **count_words** that:
  - Takes as input a filename.
  - Returns the number of words in that file.
- Question: how can we count the number of words in a string of text?
- Words are typically separated by space.
- However, they could also be separated by commas, periods, and other punctuation.
- The **String.split** method can be used to split text into words.

The **split** Method

- The **split** method separates a string into an array of "words" (smaller strings).
- Argument: a string specifying the characters that separate the words.

```
public class example
{
  public static void main(String[] args)
  {
    String text = "Today is a hot summer day.";
    String[] words = text.split(" ");
    for (int i = 0; i < words.length; i++)
    {
      System.out.printf("word[%d] = %s\n",
                        i, words[i]);
    }
  }
}
```

```
Output:

words[0] = Today
words[1] = is
words[2] = a
words[3] = hot
words[4] = summer
words[5] = day.
```

The **split** Method

- You can specify multiple characters that separate words, as seen in this example.
- In the example below, we say that words are separated by comma, space, and dash.
- Important: to specify multiple characters, **<u>you must enclose them in square brackets: []</u>**.

```
public class example {
  public static void main(String[] args)
  {
    String text = "Let's count: One,two,three.";
    String[] words = text.split("[, -]");
    for (int i = 0; i < words.length; i++)
    {
      System.out.printf("word[%d] = %s\n",
                        i, words[i]);
    }
  }
}
```

```
Output:

word[0] = Let's
word[1] = count:
word[2] = One
word[3] = two
word[4] = three.
```

The **split** Method

- You can specify multiple characters that separate words, as seen in this example.
- In the example below, we say that words are separated by comma, space, and dash.
- Important: to specify multiple characters, **<u>you must enclose them in square brackets: []</u>** (see the example argument: **"[, -]"**).

```
public class example {
  public static void main(String[] args)
  {
    String text = "Let's count: One,two,three.";
    String[] words = text.split("[, -]");
    for (int i = 0; i < words.length; i++)
    {
      System.out.printf("word[%d] = %s\n",
                        i, words[i]);
    }
  }
}
```

```
Output:

word[0] = Let's
word[1] = count:
word[2] = One
word[3] = two
word[4] = three.
```

A Third Example: Counting Words

```
public static int count_words(String filename)
{
  ArrayList<String> lines = read_file(filename);
  if (lines == null)
  {
    return 0;
  }

  int counter = 0;
  for (int i = 0; i < lines.size(); i++)
  {
    String line = lines.get(i);
    String[] words = line.split("[,.- ]");
    counter += words.length;
  }

  return counter;
}
```

Here we specify that words are separated by commas, periods, dashes, spaces.

Writing Data to a File

- We saw that, to read from a file, we associate a file with a **Scanner** object.
  - Then, we read from a file exactly as we read user input.
- To write to a file, we associate a file with a **PrintWriter** object.
  - Then, we print to the file exactly as we print user input.
  - We can use printf, println, and so on.

Writing Data to a File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

Step 1: Make sure you import java.io.PrintWriter.

Writing Data to a File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

Step 2: Initialize PrintWriter object to null (so that we can associate it with a file using try … catch).

Writing Data to a File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

Step 3: Associate the PrintWriter object with a file.

We need to use try … catch, to catch the case where for some reason the file could not be opened (invalid filename, read-only file, etc).

Writing Data to a File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

Step 4: Write whatever data you want.

Note that we use **out.printf**, and NOT System.out.printf (which would just print to the screen).

Writing Data to a File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

Step 5: close the file.

If you forget this step, some or all your data may not be saved in the file.

Writing Data to a File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

Question: what happens if your file already contained some data, before you ran this program?

Writing Data to a File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

Question: what happens if your file already contained some data, before you ran this program?

Answer: **<u>the previous data is lost</u>**.

Problem Viewing the Output File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\n");
    out.printf("writing a second line.\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

On Windows, if you open the text file using Notepad, it shows as one long line.

Problem Viewing the Output File

```
import java.io.PrintWriter;

public class file_writing_example {
  public static void main(String[] args)
  {
    String filename = "out1.txt";
    PrintWriter out = null;
    try
    {
      out = new PrintWriter(filename);
    }
    catch (Exception e)
    {
      System.out.printf("Error: failed to open file %s.\n", filename);
      System.exit(0);
    }
    out.printf("writing a line to a file.\r\n");
    out.printf("writing a second line.\r\n");
    out.close();
    System.out.printf("Done writing to file %s.\n", filename);
  }
}
```

On Windows, if you open the text file using Notepad, it shows as one long line.

To fix this problem, use \r\n instead of \n for a newline.

Writing Data to a File: Recap

- Step 1: Make sure you import java.io.PrintWriter.
- Step 2: Initialize a PrintWriter variable called **out** to null (so that we can associate it with a file using try … catch).
- Step 3: Associate the PrintWriter variable with a file.
  - We need to use try … catch, to catch the case where for some reason the file could not be opened (invalid filename, read-only file, etc).
- Step 4: Write whatever data you want.
  - Use **out.printf**, and NOT System.out.printf (which would just print to the screen).
- Step 5: close the file.
  - If you forget this step, some or all your data may not be saved in the file.

Example: Convert to Squares

- Write a function **save_squares(input_name, output_name)** that:
  - Reads numbers from a file named **input_file** that just contains numbers, one per line.
  - Writes to a file named **output_file** the square of each number that it read from the input file.

Example: Convert to Squares

- Logic:
- Read the lines from the input file.
- Open output file.
  - For each line in the input:
  - Convert string to double.
  - Compute the square.
  - Save the square to the output file.
- Close the output file.

```java
public static void save_squares(String input_file, String output_file)
{
  ArrayList<String> lines = read_file(input_file);
  PrintWriter out = null;
  try
  {
    out = new PrintWriter(output_file);
  }
  catch (Exception e)
  {
    System.out.printf("Error: failed to open %s.\n", output_file);
    return;
  }
  for (int i = 0; i < lines.size(); i++)
  {
    double number;
    try
    {
      number = Double.parseDouble(lines.get(i));
    }
    catch (Exception e)
    {
      System.out.printf("Error: %s is not a number.\n", lines.get(i));
      return;
    }
    out.printf("%.2f\r\n", number * number);
  }
  out.close();
}
```

Storing a Spreadsheet in a Variable

- A spreadsheet is a table.
  - You can specify a row and a column, and you get back a value.
- In a spreadsheet:
  - Each value is a string (that we may possibly need to convert to a number).
  - Each row is a sequence of values, thus can be stored an array (or array list) of strings.
  - The spreadsheet data is a sequence of values, thus can be stored as an array (or array list) of rows, thus an array (or array list) of arrays (or array lists) of strings.

Storing a Spreadsheet in a Variable

- Therefore, we have a few choices on how to store spreadsheet data in a variable. Our variable can be:
  - An array of arrays of strings.
  - An array of array lists of strings.
  - An array list of arrays of strings.
  - An array list of array lists of strings.

Storing a Spreadsheet in a Variable

- We will implement two of those options. We will write functions that read a spreadsheet and store its data as:
  - An array of arrays of strings.
  - An array list of array lists of strings.

Storing a Spreadsheet in a 2D Array

- If we store a spreadsheet as an array of arrays of strings, the variable looks like this:

String[][] data.

  - data[i] corresponds to the i-th line of the spreadsheet.
  - data[i] contains an array of strings.
  - data[i][j] holds the spreadsheet value for row i, column j.
- Let's write a function **read_spreadsheet** that:
  - Takes as input a filename.
  - Returns an array of arrays of strings storing all the data in the spreadsheet.

Storing a Spreadsheet in a 2D Array

```
public static String[][] read_spreadsheet(String filename)
{
  ArrayList<String> lines = read_file(filename);

  int rows = lines.size();

  // The row below creates an array of length "rows", that stores
  // objects of type String[]. Those objects are initialized to null.
  String[][] result = new String[rows][];

  for (int i = 0; i < lines.size(); i++)
  {
    String line = lines.get(i);
    String [] values = line.split(",");
    result[i] = values;
  }

  return result;
}
```

Storing a Spreadsheet in a 2D ArrayList

- If we store a spreadsheet as an array list of array lists of strings, the variable looks like this:

ArrayList<ArrayList<String>> data.

  - data.get(i) corresponds to the i-th line of the spreadsheet.
  - data.get(i) is an array list of strings.
  - data.get(i).get(j) holds the spreadsheet value for row i, column j.
- Let's write a function **read_spreadsheet** that:
  - Takes as input a filename.
  - Returns an array list of array lists of strings storing all the data in the spreadsheet.

Storing a Spreadsheet in a 2D ArrayList

```
public static ArrayList<ArrayList<String>> read_spreadsheet(String filename)
{
  ArrayList<String> lines = read_file(filename);
  ArrayList<ArrayList<String>> result = new ArrayList<ArrayList<String>>();

  for (int i = 0; i < lines.size(); i++)
  {
    String line = lines.get(i);
    String [] values = line.split(",");
    ArrayList<String> values_list = new ArrayList<String>();

    for (int j = 0; j < values.length; j++)
    {
      values_list.add(values[j]);
    }
    result.add(values_list);
  }

  return result;
}
```

Example Programs

- On the lectures web page, you can access two programs that use these **read_spreadsheet** functions.
  - nba_leaders_2d_array_version.java stores data as a 2D array of strings.
  - nba_leaders_2d_arraylist_version.java stores data as a 2D array list of strings.
- Both programs have exactly the same functionality:
  - They read a spreadsheet of statistics of about 270 basketball players.
  - Then, the user can specify a column, and the program prints out the player (or multiple players, in case of ties) with the best statistics in that column.