Chapter 6: File Processing

WEEK 8:

READING FILES

TOKEN-BASED PROCESSING

LINE-BASED PROCESSING

OUTPUT FILES

File-Reading Basics

- When you store data in your computer, you store it in a file
- File: A collection of information that is stored on a computer and assigned a particular name
- To access a file from inside a Java program, you need to construct an internal object that will represent the file
- The Java class libraries include a class called File that performs this duty
- You construct a File object by passing in the name of a file:

  File f = new File(“filename.txt”)
- Once you’ve constructed the File object, you can call a number of methods to manipulate the file

File Manipulation Methods

```
import java.io.*;   // for File

public class fileMethods {

    public static void main(String[] args) {
        File f = new File("sampleFile.txt");
        System.out.println("exists returns " + f.exists());
        System.out.println("canRead returns " + f.canRead());
        System.out.println("length returns " + f.length());
        System.out.println("getAbsolutePath returns " + f.getAbsolutePath());
    }
}
```

Console

```
<terminated> fileMethods [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 22, 2015, 6:04:47 PM)
exists returns true
canRead returns true
length returns 252
getAbsolutePath returns C:\Users\Emilia\JavaFiles\CSC142\sampleFile.txt
```

- Java has some useful methods to report basic information about files
- exists()
- canRead()
- length()
- getAbsolutePath()

NOTICE: To use File, include the import statement shown

Methods of File Objects

Method	Description
canRead()	Whether or not this file exists and can be read
delete()	Deletes the given file
exists()	Whether or not this file exists on the system
getAbsolutePath()	The full path where this file is located
getName()	The name of this file as a String, without any path attached
isDirectory()	Whether this file represents a directory/folder on the system
isFile()	Whether this file represents a file (nonfolder) on the system
length()	The number of characters in this file
renameTo(file)	Changes this file's name to the given file's name

Reading a File with a Scanner

- The Scanner class we have been using is flexible in that Scanner objects can be attached to many different kinds of input
- Thus far we have been constructing Scanner objects by passing System.in to the Scanner constructor
- Instead, we can pass the constructor a File object:

  Scanner input = new Scanner(new File("sampleFile.txt"));

- A line like the one above will appear in all of your file processing programs
- Unfortunately, you can't compile a program that constructs a Scanner in this manner
- This is because the compiler can't be sure that it will be able to find the specified file
- If the file can't be found, the program will generate an error by throwing what is known as a **FileNotFoundException** – this type of exception is known as a **CHECKED EXCEPTION**

Checked Exceptions

- **Checked Exception:** An exception that **MUST** be **caught or specifically declared** in the header of the method that might generate it
- Because FileNotFoundException is a checked exception, it must be handled in some way
- Java provides a construct known as the try/catch statement for handling such errors (Appendix C)
- Java also allows you to avoid handling the potential error, as long as you clearly indicate the fact that you aren't handling it – to do this, you just need to include a throws clause in the header for the main method to indicated that your main method may generate this exception
- **throws Clause:** A declaration that a method will not attempt to handle a particular type of exception
- **Example:**

```
public static void main(String[] args) throws FileNotFoundException{
    . . .
```

- With this declaration included, your program should now compile

Example of Reading from a File

- Once the Scanner is constructed to read from the file, it can be manipulated like any other Scanner object

NOTE: The while loops keeps reading words from the file as long as there are words to read. At the end of the while loop, there is no input left, so further attempts to read from the file would generate a **NoSuchElementException** indicating that you are attempting to read beyond the end of the input.

```
import java.io.*;   // for File
import java.util.*; // for Scanner

public class fileMethods {

    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("sampleFile.txt"));

        int count = 0;
        while (input.hasNext()){
            String word =  input.next();
            count++;
        }
        System.out.println("total words: " + count);
    }
}
```

Console

```
<terminated> fileMethods [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 22, 2015, 6:49:01 PM)
total words: 47
```

Forgetting to include the File object

```
public class fileMethods {

    public static void main(String[] args) throws FileNotFoundException {

        Scanner input = new Scanner("sampleFile.txt");

        int count = 0;
        while (input.hasNext()){
            String word =  input.next();
            count++;
        }
        System.out.println("total words: " + count);
    }
}
```

```
<terminated> fileMethods [Java Application]
total words: 1
```

This code does not create any error because, in addition to reading from a file, a Scanner object can also be constructed to read from a String.

However, as written, the Scanner is only associated with the text “sampleFile.txt” – which gives a word count of 1.

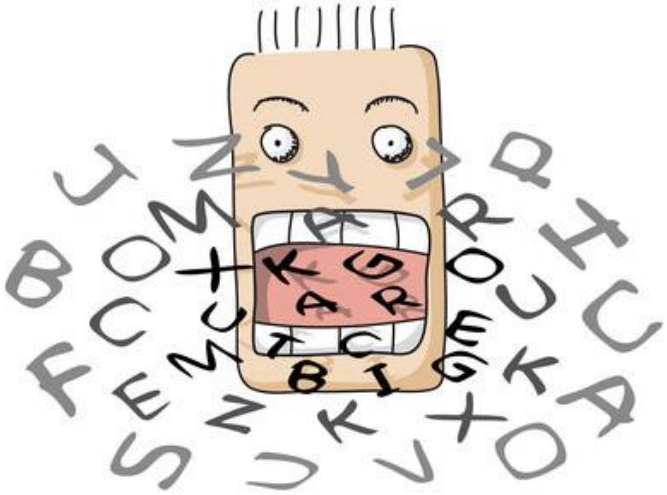
Token-Based Processing

- **Token-Based Processing:** Processing input token by token (i.e. one word at a time or one number at a time)
- Remember that the Scanner class includes a series of **hasNext** methods that parallel the various **next** methods
- By using a while loop, you can iterate through the data in a file checking before reading in a new token
- **Example:**

```
int count = 0;
while (input.hasNextDouble()){
    double next =  input.nextDouble();
    count++;
}
```

Processing a File

- When processing a file, a Scanner object keeps track of the current position in the file.
- You can think of this as an ***input cursor*** or pointer into the file.
- When the Scanner object is first constructed, the cursor points to the beginning of the file
- With each call to one of the next methods, this cursor advances to the next token
- This process is called "**consuming input**"
- The process of consuming input doesn't actually change the file, it just changes the Scanner object so that it is positioned at a different point in the file
- When consuming input, the Scanner skips leading whitespace characters (spaces, tabs, new line characters) until it comes to the next token



CONSUMING INPUT?

Figure from: http://www.callcentrehelper.com/the-right-words-and-phrases-to-use-with-an-angry-customer-30110.htm

No Methods for Scanning “Backwards”

- Scanner objects are designed to process data in a forward manner
- They provide quite a bit of flexibility for looking ahead in an input file
- They provide no way of reading input backwards
- There are no “previous” methods
- If you need to process the items in the input file again, you would have to construct a new Scanner object
- This newly-created Scanner object would be positioned at the beginning of the file

Scanner Parameters

- A Scanner object can be passed as a parameter to a method

```
public class passScanner {

    public static void main(String[] args) throws FileNotFoundException{
        Scanner input = new Scanner(new File("sampleFile.txt"));
        processTokens(input, 2);
        processTokens(input, 3);
        processTokens(input, 2);

    }

    public static void processTokens(Scanner input, int n){
        System.out.println("You requested " + n + " tokens.");
        for (int i = 1; i <= n; i++){
            String nextWord = input.next();
            System.out.println("Token #" + i + " is: " + nextWord);
        }
        System.out.println();
    }

}
```

- **The Scanner object is constructed in the main method and associated with the file "sampleFile.txt"**
- The actual file processing occurs in the ***processTokens*** method
- Thus, the Scanner object created in the main method needs to be passed in to the ***processTokens*** method
- The new calls to ***processTokens*** do not create new Scanner objects – the **SAME** Scanner object is passed in each time – thus its input cursor position reflects the total number of items already processed from the file

Scanner Parameters -- output

- A Scanner object can be passed as a parameter to a method

```
public class passScanner {

    public static void main(String[] args) throws FileNotFoundException{
        Scanner input = new Scanner(new File("sampleFile.txt"));
        processTokens(input, 2);
        processTokens(input, 3);
        processTokens(input, 2);

    }

    public static void processTokens(Scanner input, int n){
        System.out.println("You requested " + n + " tokens.");
        for (int i = 1; i <= n; i++){
            String nextWord = input.next();
            System.out.println("Token #" + i + " is: " + nextWord);
        }
        System.out.println();
    }

}
```

```
<terminated> passScanner [Java App
You requested 2 tokens.
Token #1 is: This
Token #2 is: is

You requested 3 tokens.
Token #1 is: a
Token #2 is: sample
Token #3 is: file

You requested 2 tokens.
Token #1 is: for
Token #2 is: the
```

NOTE: A new call to the processTokens method does NOT “reset” the file text.

Paths and Directories

- Files are organized in a hierarchy
- The directory at the top of the hierarchy is called the **root directory**
- Any directory can contain files as well as other directories (known as "**subdirectories**")
- **Subdirectories** may also contain files and subdirectories of their own
- **File path:** A description of the file's location on a computer, starting with a drive and including the path from the root directory to the directory where the file is stored
- When a program is given a file name for a file of data, without the entire path to the file specified, it looks for the file in the **CURRENT DIRECTORY** (also known as the "**WORKING DIRECTORY**")
- The simple file name is the **"relative path"** of the file – when Java encounters a simple name like this it looks in the current directory
- The fully qualified (or complete) file name could also have been used – in this case you would be using what is known as the **"absolute file path"**

How to Write File Names

- If you are not working with a file in your current ("working") directory, you can access the file you want by specifying its absolute file path
- To do this, you need to know EXACTLY where your file is stored in your system
- **Example:** For a file located in the **C:\data** directory (Windows machine) , you could use a command like the following to associate the file with a Scanner object:

```
Scanner input = new Scanner(new File("C:/data/filename.txt"));
```

- Notice that the path is written with **forward-slash** characters
  - On Windows machines, you would normally use a backslash, but Java lets you use a forward slash
  - This saves you from having to use a \\ ESCAPE for each backslash

Filename via User Input

- What if, instead of writing the file name in the code, you want the user to input the file name?
- You can do this with code similar to the code fragment shown:

```
Scanner console = new Scanner(System.in);
System.out.println(" What is the file name? ");
String name = console.nextLine();
Scanner input = new Scanner(new File(name));
```

- Notice that the code fragment includes **TWO Scanner objects** – one for reading from the **console** and one for reading from the **file**
- You would still need to include the **throws FileNotFoundException** in the header for main, since the user may enter the name of a file that cannot be located
- The user may enter the file name with single backslashes since the Scanner object that reads user input is able to read the file name without escape sequences

A More Complex Input File

- Suppose you are reading in data from a file that contains both employee names and the number of hours each employee worked. The data might look like the following:

Erica	7.5	8.5	10.25	8	8.5
Erin	10.5	11.5	12	11	10.75
Simone	8	8	8		
Ryan	6.5	8	9.25	8	
Kendall	2.5	3			

- Suppose you want to find the total hours worked by each employee – for this program you can assume the name is a simple string, with no spaces
- You will need to read each person's name and then read that person's lists of hours
- You can do this using nested loops

HoursWorked

```
public class HoursWorked {

    public static void main(String[] args) throws FileNotFoundException {

        Scanner input = new Scanner(new File("hours.dat"));
        process(input);
    }

    public static void process (Scanner input){
        while(input.hasNext()){
            String name = input.next();
            double sum = 0.0;
            while(input.hasNextDouble()){
                sum += input.nextDouble();
            }
            System.out.println("Total hours worked by " + name + " = " + sum);
        }
    }
}
```

```
<terminated> HoursWorked [Java Application] C:\
Total hours worked by Erica = 42.75
Total hours worked by Erin = 55.75
Total hours worked by Simone = 24.0
Total hours worked by Ryan = 31.75
Total hours worked by Kendall = 5.5
```

Line-Based Processing

- The practice of processing input line by line (i.e. reading in entire lines of input at a time)
- Most file processing involves a combination of line- and token-based processing
- The Scanner class lets you write programs that include both styles
- For line-based processing, use the hasNextLine and nextLine methods
- One of the advantages of line-based processing is that you don't lose the spacing within the line, as you would if you processed the file using token-based processing
- Another advantage (as we saw earlier with our employees and hours data) is that data in files is often arranged in rows, so reading the files line by line makes sense given the structure of the file – the example on the next slide demonstrates this

HoursWorked, version 2

- In this version, the data file contains employee ID numbers as well, as shown:

```
101     Erica   7.5   8.5   10.25   8    8.5
783     Erin    10.5  11.5  12      11   10.75
114     Simone  8     8     8
238     Ryan    6.5   8     9.25    8
156     Kendall 2.5   3
```

- This version of our data file isn't so simple to read in using token-based processing
- We can read in the first employee's ID using the **nextInt** method, but now we don't have a convenient way to transition from reading the last hours value for one employee to reading the name of the next employee – the ID value is in the way
- Java will read in the next employee's ID number as another "double" value
- Processing this file line-by-line will get rid of this problem – and will guarantee that data for two employees does not get inadvertently combined

HoursWorked, version 2 code

```
public class HoursWorked2 {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("hours2.dat"));
        while(input.hasNextLine()){
            String text = input.nextLine();
            processLine(text);
        }
    }
    public static void processLine (String text){
        Scanner data = new Scanner(text);
        int id = data.nextInt();
        String name = data.next();
        double sum = 0.0;
        while(data.hasNextDouble()){
            sum += data.nextDouble();
        }
        System.out.println("Total hours worked by " + name
                + " (id# " + id + ") = " + sum);
    }
}
```

The main method has a Scanner object to read the file in line-by-line

```
<terminated> HoursWorked2 [Java Application] C:\Program Files\
Total hours worked by Erica (id# 101) = 42.75
Total hours worked by Erin (id# 783) = 55.75
Total hours worked by Simone (id# 114) = 24.0
Total hours worked by Ryan (id# 238) = 31.75
Total hours worked by Kendall (id# 156) = 5.5
```

The process method creates a “mini-Scanner” object to handle each line of input as a string. Every iteration of the while loop in the main method creates a new Scanner object.

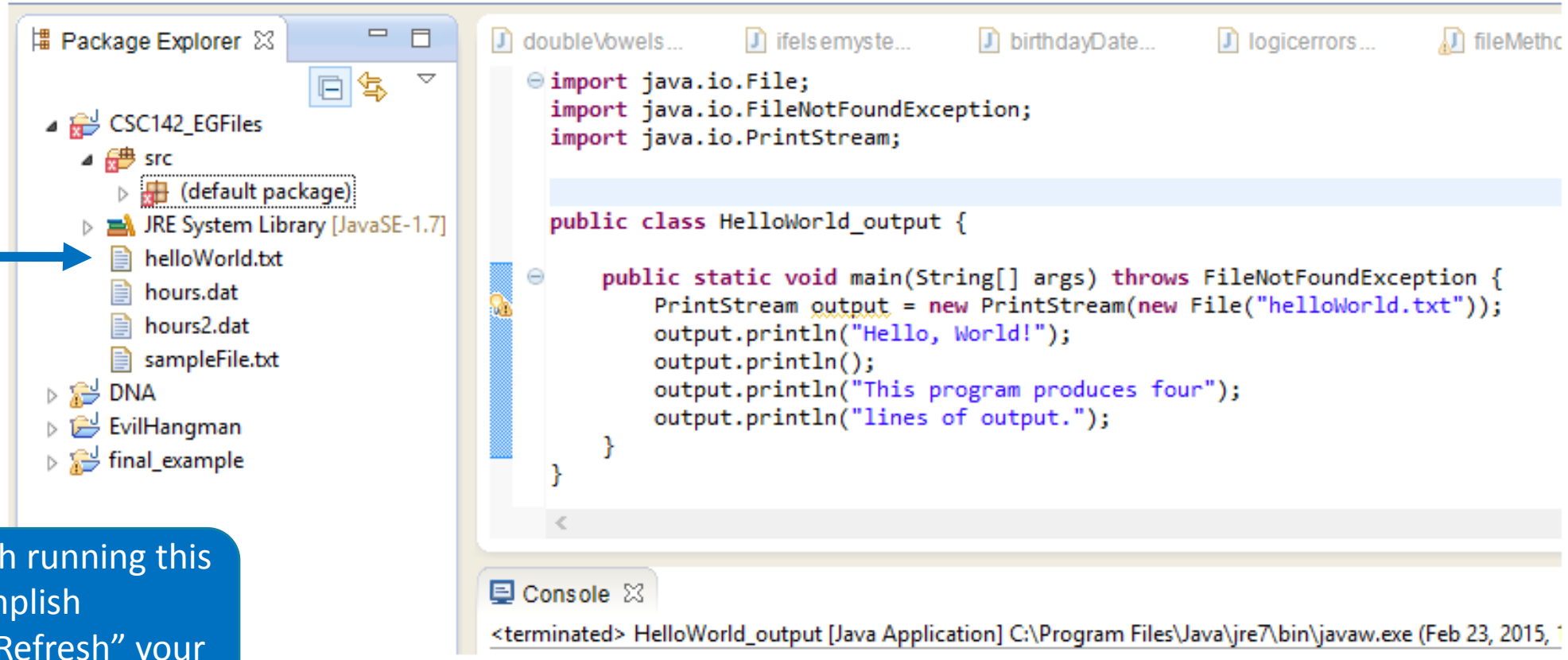
Advanced File Processing: Output Files

- You can **write output to a file** instead of writing it to the console
- System.out is a variable that stores a reference to an object of type **PrintStream** that is tied to the console window
- You can construct other **PrintStream** objects that send their output to other places:

PrintStream output = new PrintStream(new File(“results.txt”))

- This statement tells the computer to create an output file
  - **If a file with this name already existed, it will be “overwritten”**
- Initially, the file that is created as a result of this statement is empty
- To send data to the file (to write to the file) you can use the **print** and **println** commands you are already familiar with – you can also pass **PrintStream** objects to methods to do the processing there

HelloWorld! -- The output file version



It may seem as though running this program didn't accomplish anything, but if you "Refresh" your IDE, you'll see a new file.

Guaranteeing That Files Can Be Read

```
public static void main(String[] args)
        throws FileNotFoundException {
    Scanner console = new Scanner(System.in);
    Scanner input = getInput(console);
    int count = 0;
    while(input.hasNext()){
        String word = input.next();
        count++;
    }
    System.out.println("total words = " + count);
}

public static Scanner getInput(Scanner console)
        throws FileNotFoundException{
    System.out.print("input file name: ");
    File f = new File(console.nextLine());
    while(!f.canRead()){
        System.out.println("File not found. Please try again.");
        System.out.print("input file name: ");
        f = new File(console.nextLine());
    }
    // now we know f is a file that can be read
    return new Scanner(f);
}
```

This scanner gets input from the console.

This scanner gets input from the file.

The getInput method takes a scanner that gets input from the console as an argument.

The getInput method returns a scanner gets input from the file.

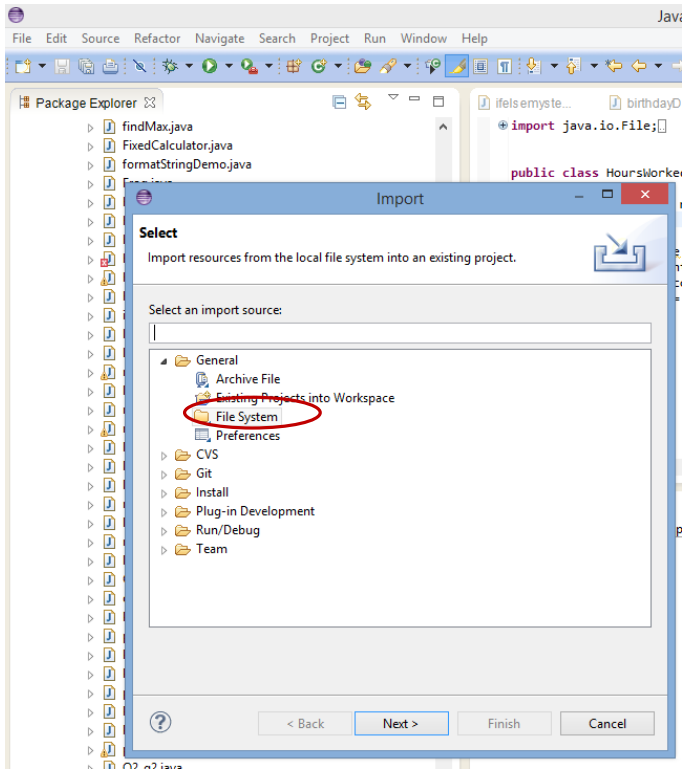
You can write a method getInput that uses a while loop to ensure that the file name entered by a user is, in fact, a valid file name.

Concluding Comments

- **File objects:** Used to represent files in Java. The File class is found in the java.io package
- **Scanner object:** Can be used to read input from a file by passing new File(filename) to the constructor instead of System.in
- **Checked exception:** A program error condition that must be caught or declared in order for the program to compile
- **hasNext methods:** Methods that let you "look ahead" in a file to ensure you don't try to read in data that doesn't exist.
- **Relative path:** Refers to a file that exists in the current ("working") directory or in one of its subdirectories by specifying the path to the file FROM the current location (i.e. provides only a partial file path).
- **Absolute path:** Provides the full file path to a file located somewhere in the file system.
- **Token-based processing:** The practice of processing input token by token (chunks of text separated by whitespace).
- **Line-based processing:** The practice of processing input by reading it in a line at a time.
- **PrintStream object:** Constructed with a File and has the same methods as System.out (such as print and println). Can be used to create an output file, rather than just printing output to the console.

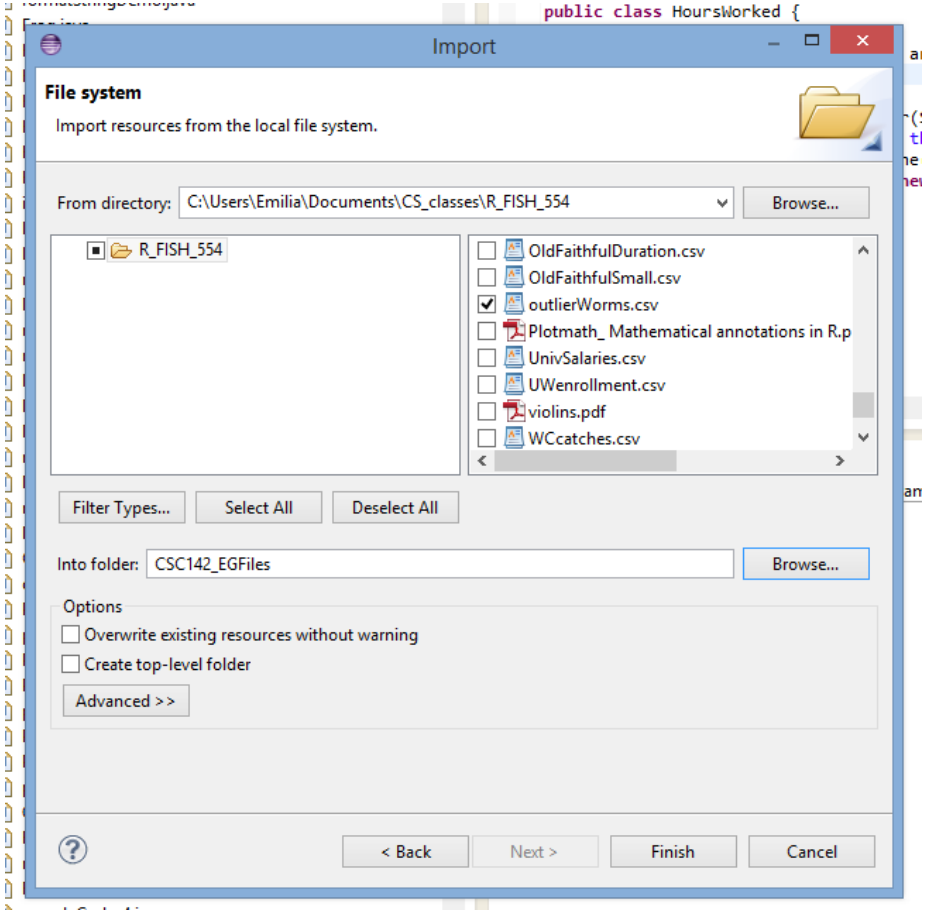
Importing Files to Use in Your Programs

- Under the “File” tab, click on “Import”
- A new window will pop up, as shown:
- Click on “General”
- Click on “File System”
- Click “Next”
- Another window will pop up showing you your file system and allowing you to browse your file system to locate the file you want to import



Importing Files to Use in Your Programs, cont.

- Once you have selected the directory that contains your file, you will be able to see the individual files in the directory
- You need to select the file you want by clicking the box in front of the file name
- You also need to specify what folder you are importing the file in to
- This file should be the Java Project file you're working in
- Click on "Finish"



Importing Files to Use in Your Programs, cont.

- Your imported file should appear under the JRE System Library for your project
- If it appears under src, you will need to move the file for your program to work properly (i.e. with no FileNotFoundException)
- You should now be able to specify the file name using just the name itself as the path