Tutorial
Java File I/O

Mark G. Eramian

University of Saskatchewan

Opening a File

- In Java, files are abstracted by creating an instance of the `java.io.File` class.
- It requires the filename as the parameter.
- Creating an `File` object does not open the file.

```java
import java.io.File;

public class SomeClass {
    public static void main(String args[]) {
        /** Code snippet to open a file. */
        File myFile = File('filename.txt');
    }
}
```

Reading Text Files

- Text files can be opened and read using the `java.util.Scanner` class.
- An instance of `Scanner` can be created from an instance of `File`.

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

public class SomeClass {
    public static void main(String args[]) {
        /** Code snippet to open a file. */
        File myFile = File('filename.txt');
        Scanner infile = new Scanner(myFile);
    }
}
```

Reading Text Files

- The file is not actually "opened" until you instantiate Scanner so you should use a try-catch block to make sure the file was opened successfully. For example:

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

public class SomeClass {
    public static void main(String args[]) {
        Scanner infile = null;
        try {
            infile = new Scanner(new File('filename.txt'));
        }
        catch (FileNotFoundException e) {
            System.out.println("Error: file not found.");
            return;
        }
    }
}
```

Using Scanner

- *Tokens* are sequences of characters from text file that are separated by delimeters.
- `Scanner` can be used to read the next token from an open file.
- The default delimiters are whitespace.
- `Scanner` has methods to read the next token and convert it to a particular data type.
- Example: the `Scanner.nextInt()` method reads the next token from the file and tries to convert it to an integer, throwing an exception if that is not possible.
- `Scanner` also has methods to test whether the next token in the file can be interpreted as a particular data type, e.g. `hasNexint()`.

Example: Read a List File of Integers

- Recall from CMPT 141: a list file is a file with one data item (token) per line.
- This code below a file with one integer per line and adds them up.

```java
public static void main(String args[]) {
    Scanner infile = null;
    try {
        infile = new Scanner(new File('filename.txt'));
    }
    catch (FileNotFoundException e) {
        System.out.println("Error: file not found.");
        return;
    }
    int sum = 0;
    while( infile.hasNextInt() ) {
        sum = sum + infile.nextInt();
    }
    infile.close()
}
```

Error Checking

- Previous example works as long as every line has a valid number.
- If a non-integer is encountered, the file reading stops and later valid integers are not read.
- We can use the `hasNext()` method, which returns true as long as there is a another token to be read regardless of its interpretation, to distinguish between reaching the end of the file and reaching an invalid token.
- This lets us continue adding integers in the remainder of the file even if we encounter a non-integer at some point.

Example: Read a List File with Better Error Checking

```
public static void main(String args[]) {
    Scanner infile = null;
    try {
        infile = new Scanner(new File('filename.txt'));
    }
    catch (FileNotFoundException e) {
        System.out.println("Error: file not found.");
        return;
    }
    int sum = 0;
    while( infile.hasNext() ) {
        try {
            sum = sum + infile.nextInt();
        }
        catch( InputMismatchExcepition e ) {
            System.out.println("Warning: non-integer token.");
        }
    }
    infile.close()
}
```

Other Useful Scanner Methods

- `hasNextFloat()`, `hasNext()`: check for/read float tokens
- `hasNextLine()`, `nextLine()`: check for/read entire lines (without tokenizing). Useful for reading human-readable text, or when you want to read a line and tokenize it yourself.
- `next()`: return the next token, whatever it is (this is how you read string-valued tokens).
- `useDelimeter(pattern)`: sets the delimeter to exactly `pattern`.
- Full listing of `Scanner` methods in the Java 1.8 docs.

Reading Other File Formats

- Recall from CMPT 141 *tabular files*: fixed number of data items per line separated by delimeters; data items on one line may be of different types, but i-th data item on a line is always the same type.
- Two approaches in Java:
  - Write logic to read each line, reading each token with appropriate `Scanner.nextInt()`, `Scanner.nextFloat()`.
  - Use `Scanner.nextLine()` to read each line, use `String.trim()` and `String.split()` in much the same way as `strip()` and `split()` are used in python.
  - In the latter approach, all tokens will be strings, and you have to make the appropriate type conversion yourself after splitting. This is harder in Java because: no list comprehensions!

Exercise 1

- Write java code to read a file where each line contains (in order):
  - Month (string)
  - Year (string)
  - Min temp (float)
  - Maz temp (float)
- Items on each line are separated by whitespace.
- Store the data as a `Java.util.ArrayList` of `TempRecord` objects (which just stores the four data items and has getters/setters).
- Our solution will use the first approach.

Writing Files

- There are a mind-wrenching number of ways to write data to a file in Java.
- We'll use the `BufferedWriter` class.

```java
import java.io.FileWriter;
import java.io.IOException;
import java.io.BufferedWriter;

public class WriteFileExample {
    public static void main(String args[]) {
        // Open a file for writing
        BufferedWriter outfile = null;
        try {
            outfile = new BufferedWriter(new FileWriter("filename.txt"));
        }
        catch (IOException e) {
            System.out.println("Error: file cannot be opened.");
            return;
        }
    }
}
```

Writing Files for Writing

- There are a mind-wrenching number of ways to write data to a file in Java.
- We'll use the BufferedWriter class.

```java
import java.io.FileWriter;
import java.io.IOException;
import java.io.BufferedWriter;

public class WriteFileExample {

    public static void main(String args[]) {
        // Open the file

        BufferedWriter outfile = null;
        try {
            outfile = new BufferedWriter(new FileWriter("stuff.txt"));
        }
        catch (IOException e) {
            System.out.println("Error: file cannot be opened.");
            return;
        }
    }
}
```

Writing data to Files

- The `BufferedWriter.write(s)` method will write the string `s` to the file opened for writer.
- Only strings may be written. Other data types must be converted before being passed to `BufferedWriter.write()`.
- All desired whitespace, including newlines, must be explicitly written.
- All `BufferedWriter.write()` calls have to be in a try-catch block because the method can throw the checked exception `IOException`.
- You must close the file with the `BufferedWriter.close()` method (also must be in try-catch block) or not all of the data you wrote will actually make it to disk.

Exercise 2

Write code to write the values of the following variables to a file:

```java
String s = "The answer to the ultimate question of life,"+
           "the universe, and everything is:";
int fortyTwo = 42;
float fortyTwoPointZero = 42.0;
```

Make sure that each variable's value is on a separate line.

Concluding Remarks

- File I/O in java is much uglier than in Python.
- There are other ways to do file I/O beyond what is shown here. This is just the basics.
- What you've seen here is enough to get you through CMPT 280.
- Online tutorials on Java File I/O are numerous and, in the course instructor's opinion, universally awful.