CS601: Principles of Software Development

File Input/Output
File Processing.
Intro to Exceptions.

Olga A. Karpenko

Parts of this presentations are based on the Java Solftware Solutions book by Lewis&Loftus.

Announcements

- ***Withdraw*** deadline is today, Sept 8th
  - Last chance to tuition back
- Lab 1 part 1 is due on Monday

Lab 1 Notes

- Use .equals to compare strings for equality

```
public void func(String s1, String s2)
{
   if (s1.equals(s2))
         // do something
}
```

- Use System.lineSeparator() instead of "\n"

Lab 1 Notes

- To convert date as a string like to the Date object:

```
String dateString = "2016-06-29T17:50:37";
DateFormat format = new
SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
Date d = format.parse(dateString);
```

- toString method of Date will print d as:

  Wed Jun 29 17:50:37 PDT 2016

File Processing in Java

Relevant Classes from Java 8 API

From java.io.*

- File
- Scanner
- BufferedReader
- PrintWriter

From java.nio

- Path
- Paths
- Files
- FileSystem

Path

- In java.nio package
- Represents a “path” in the file system
- Example: `"/Users/okarpenko/Documents/"`
- Methods:
  - getRoot()
  - getParent()
  - getFileName()
  - isAbsolute()
  - toAbsolute()
  - normalize()

Paths

- Has static methods to create Path objects

```
Path p = Paths.get("myfile.txt");

Path p = Paths.get("/Users/okarpenko/
Documents/", "hotelsSanDiego");
```

Files

- In java.nio.file
- Includes helper methods
  - To get attributes of Path objects
  - To list the files within a directory
  - To read lines from the file
  - …

- See `PathExample.java`

File Input

- FileInputStream – reads raw bytes InputStreamReader and FileReader – read characters
  - FileReader uses default encoding
- BufferedReader
  - More efficient due to buffering
  - Can read a line at a time

Reading From the File

```
FileInputStream fs = new FileInputStream(filename),
"UTF-8");
BufferedReader  reader = new BufferedReader(new
   InputStreamReader(fs);
 String line;
 while ((line = reader.readLine()) != null) {
       System.out.println(line);
 }
```

The code above needs to handle exceptions

See `FileIOExample.java`

Reading From the File

- Using Files.newBufferedReader:

```
BufferedReader reader =
Files.newBufferedReader(path,
Charset.forName("UTF-8"));

String line = null;
while ((line = reader.readLine()) != null) {
        System.out.println(line);
}
```

The code above needs to handle exceptions

See `FileIOExample.java`

Writing to a File

- Selected classes:

  PrintWriter, BufferedWriter,
FileOutputStream

Writing to a File

```
PrintWriter writer =
new PrintWriter(new FileWriter("out.txt"));

String line = "hello";
writer.println(line);
writer.flush();
```

- Note: Need to take care of IO exceptions
- See `FileIOExample`: read from the file & write to a different file

DirectoryStream

- An Interface
- If Implemented, enables iteration through the contents of a directory

DirectoryStream: Example

```
Path p = Paths.get(”MyFolder");

DirectoryStream<Path> filesList =
Files.newDirectoryStream(p);

for (Path file: filesList) {

   // process the file

}
```

- See `DirectoryListingExample.java`

References

- http://docs.oracle.com/javase/tutorial/essential/io/index.html

Introduction to Exception Handling

What is an Exception?

- Exception
  - An event that occurs during the execution of a program that disrupts the normal flow
  - Example: tried to open a file, but no such file exists

What is an Exception?

- Exception object
  - Created by the method when an error occurs within a method
  - Contains information about the exception
    - its type (Ex. IllegalArgumentException)
    - the state of the program when the error occurred.

  - Method hands it off to the runtime system

From Java Tutorial: http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exception: examples

- Array Index Out of Bounds
  - ArrayIndexOutofBoundsException

    http://docs.oracle.com/javase/7/docs/api/java/lang/ArrayIndexOutOfBoundsException.html

- File Not Found
  - FileNotFoundException

    http://docs.oracle.com/javase/7/docs/api/java/io/FileNotFoundException.html

- Following a Null Reference
  - NullPointerException

Exceptions

- Exceptions are *thrown* by some statements
- May be *caught* and *handled* by another piece of code

Java Exception Handling

- Enable program to operate even in the presence of an exception
  - Note problem and continue
  - Terminate gracefully

- Allows for grouping of types of exceptions
- Important for building robust software

Exception Handling

- In Java: a predefined set of exceptions that can occur during execution
- A program can deal with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program
- How to handle each exception is important design decision

Exception Handling

- If an exception is ignored -> the program will terminate / print error message
- The message includes a *call stack trace* that:
  - indicates the line on which the exception occurred
  - shows the method call trail

Catching exceptions

- The **try/catch** statement :

```
try {
   //statements that may throw an exception
}
catch(Exception_type name) {
   //do something
}
finally {
   //code that will execute whether or not 
an exception is thrown
}
```

The finally Clause

- Optional
- Is always executed
- No exception -> finally clause is executed after the statements in the try block
- Exception –>finally clause is executed after the statements in the appropriate catch clause

try Statement: Example

```
int a = 5;
int b = 0;
try {
   int c = a / b;
   System.out.println(c);
}
catch (ArithmeticException e) {
   System.out.println("Can't divide by 0.");
}
```

- See `ArithmeticExceptionDemo.java`

Exceptions

- To be continued..