Object Oriented Programming

(CSC 302)

Lecture Note 7

File I/O

4th December, 2019

How to work with file I/O

- A package for working with directories and files

  java.nio.file

  - To work with a directory or file, you use a Path object. A Path object can include a root component as well as directory names and a file name.

- A static method of the Paths class

  - get(String[, String]...) - Returns a Path object for the string or series of strings that specify the path.

- Methods of the Path interface

  - **getFileName()** - Returns a Path object for the name of the file or directory.
  - **getName(**int**)** - Returns a Path object for the element at the specified index.
  - **getNameCount()** - Returns an int value for the number of name elements in the path.
  - **getParent()** - Returns a Path object for the parent path if one exists. Returns a null value if a parent does not exist.
  - **getRoot()** - Returns a Path object for the root component of the path. Returns a null value if a root does not exist.
  - **toAbsolutePath()** - Returns a Path object for the absolute path to the file or directory.
  - **toFile()** - Returns a File object for the path to the file.

- Static methods of the Files class
  - **exists(**Path) - Returns a true value if the path exists.
  - **notExists(**Path**)** - Returns a true value if the path does not exist.
  - **isReadable(**Path**)** - Returns a true value if the path exists and is readable.
  - **isWritable(**Path**)** - Returns a true value if the path exists and is writable.
  - **isDirectory(**Path**)** - Returns a true value if the path exists and refers to a directory.
  - **isRegularFile(**Path**)** - Returns a true value if the path exists and refers to a regular file.
  - size(Path) - Returns a long value for the number of bytes in the file.
  - **newDirectoryStream(**Path**)** - Returns a DirectoryStream<Path> object that you can use to loop through all files and subdirectories of the directory.

- **createFile(**Path**)** - Creates a new file for the specified Path object if one doesn't already exist. Returns a Path object for the file.
- **createDirectory(**Path**)** - Creates a new directory for the specified Path object if the directory doesn't already exist and all parent directories do exist. Returns a Path object for the directory.
- **createDirectories(**Path**)** - Creates a new directory represented by the specified Path object including any necessary but non-existent parent directories. Returns a Path object for the directory.
- **delete(**Path**)** - Deletes the file or directory represented by the Path object. A directory can only be deleted if it's empty.

- Code examples that work with directories and files
  - Java 7 introduced the **java.nio.file** package (also known as NIO.2). This package provides an improved way to access the default file system and is designed to replace the functionality that was available from the **java.io.File** class.
  - The java.nio.file package provides support for many features that aren't provided by the java.io.File class.
  - When coding **paths**, you can use a front slash to separate directory names. This works equally well for Windows and other operating systems.
  - To identify the name and location of a file, you can use an absolute path name to specify the entire path for a file. You can also use a relative path name to specify the path of the file relative to the **current working directory** (CWD). This is usually but not always the directory that the application was started from.
  - Although the java.nio.file package was introduced with Java 7, the java.nio package was introduced with Java 4.

- Code that creates a directory if it doesn't already exist

```
String dirString = "c:/jsiit/java_netbeans/files";
Path dirPath = Paths.get(dirString);
  if (Files.notExists(dirPath)) {
      Files.createDirectories(dirPath);
}
```

- Code that creates a file if it doesn't already exist

```
String fileString = "products.txt";
Path filePath = Paths.get(dirString, fileString);
if (Files.notExists(filePath)) {
      Files.createFile(filePath);
}
```

- Code that displays information about a file

```
System.out.println("File name:          " + filePath.getFileName());
System.out.println("Absolute path:    " + filePath.toAbsolutePath());
System.out.println("Is writable:        " + Files.isWritable(filePath));
```

- Code that displays the files in a directory

```
if (Files.exists(dirPath) && Files.isDirectory(dirPath)) {
        System.out.println("Directory: " + dirPath.toAbsolutePath());
        System.out.println("Files: ");
        DirectoryStream<Path> dirStream = Files.newDirectoryStream(dirPath);
        for (Path p: dirStream) {
        if (Files.isRegularFile(p))
                System.out.println("  " + p.getFileName());
        }
}
```

- An introduction to file input and output
  - An input file is a file that is read by an application. An output file is a file that is written by an application. Input and output operations are often referred to as I/O operations or file I/O.
  - A stream is the flow of data from one location to another. To write data to a file from internal storage, you use an output stream. To read from a file into internal storage, you use an input stream.
  - To read and write text files, you use character streams. To read and write binary files, you use binary streams.
  - Streams are not only used with disk devices, but also with input devices like keyboards and network connections and output devices like PC monitors and network connections.

- How files and streams work
  - **Two types of files**

    Text - A file that contains characters. The fields and records in this type of file are often delimited by special characters like tab and new line characters. Web pages are also plain text files that use a special markup language known as HTML.

    Binary - A file that may contain characters as well as other non-character data types that can't be read by a text editor. Common binary file types include images, movies, and applications.
  - **Two types of streams**

    Character - Used to transfer text data to or from an I/O device.

    Binary - Used to transfer binary data to or from an I/O device.

- A file I/O example
  - The java.io package contains dozens of classes that can be used to work with different types of streams that have different functionality.
  - To get the functionality you need for a stream, you often need to combine, or layer, two or more streams.
  - To make disk processing more efficient, you can use a buffered stream that adds a block of internal memory called a buffer to the stream.
  - When working with buffers, you often need to flush the buffer. This sends all data in the buffer to the I/O device. One way to do that is to use a try-with-resources statement to automatically close the I/O stream after you use it.

- How to import all necessary packages
  - import java.io.*;
  - import java.nio.file.*;
- Get a Path object for the filePath

```
productsPath = Paths.get("products.txt");
File productsFile = productsPath.toFile();
```

- Write data to the file

```
try (PrintWriter out = new PrintWriter(
                        new BufferedWriter(
                        new FileWriter(productsFile)))) {
        out.println("java\tBeginning Java Programming\t57.50");
} catch (IOException e) {
        System.out.println(e);
}
```

- Read data from the file

```
try (BufferedReader in = new BufferedReader(
                            new FileReader(productsFile))) {
        String line = in.readLine();
        System.out.println(line);
} catch (IOException e) {
        System.out.println(e);
}
```

- How to work with I/O exceptions

  A subset of the IOException hierarchy

  IOException EOFException

  FileNotFoundException

- Common I/O exceptions
  - IOException - Thrown when an error occurs in I/O processing.
  - EOFException - Thrown when code attempts to read beyond the end of a file.
  - FileNotFoundException - Thrown when code attempts to open a file that doesn't exist.

- Path productPath = Paths.get("products.txt");

```
if(Files.exists(productsPath)) {          // prevent the FileNotFoundException
   File productsFile = productsPath.toFile();
   try(BufferedReader in = new BufferedReader(
                           new FileReader(productsFile))) {
      String line = in.readLine();
      while(line != null) {               //prevent the EOFException
         System.out.println(line);
         line = in.readLine();
      }
   }
    catch(IOException e) {
      System.out.println(e);
    }
} else {
   System.out.println(
        productsPath.toAbsolutePath() + " doesn't exist");
 }
```

- The loop that prevents the EOFException

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

- Another way to code this loop

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

- How to work with text files
  - A subset of the Writer hierarchy

Writer <<abstract>>

BufferedWriter

PrintWriter

OutputStreamWriter

FileWriter

  - The **Writer** class is an abstract class that's inherited by all of the classes in the Writer hierarchy. If the output file doesn't exist when the **FileWriter** object is created, it's created automatically. If it does exist, it's overwritten by default. If that's not what you want, you can specify true for the second argument of the constructor to append data to the file.
  - If you specify true for the second argument of the **PrintWriter** constructor, the *autoflush* feature flushes the buffer each time the *println* method is called.
- Classes used to connect a character output stream to a file

PrintWriter	contains the methods for writing data to a text stream
→BufferedWriter	creates a buffer for the stream connects the stream to a file
→FileWriter	connects the stream to a file

- Constructors of these classes

  PrintWriter(Writer[, booleanFlush]**)**

  BufferedWriter(Writer**)**

  FileWriter(File[, booleanAppend]**)** - throws IOException

  FileWriter(StringPathName[, booleanAppend]**)** - throws IOException

- How to connect without a buffer (not recommended)

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

- A more concise way to code the previous example

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

- How to connect to a file with a buffer

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

- How to connect for an append operation

```
PrintWriter out = new PrintWriter( new BufferedWriter(
new FileWriter("products.txt", true)));
```

- How to connect for an append operation

```
PrintWriter out = new PrintWriter( new BufferedWriter(
new FileWriter("products.txt", true)));
```

How to connect with the autoflush feature turned on

```
PrintWriter out = new PrintWriter( new BufferedWriter(
new FileWriter("products.txt")), true);
```

- How to write to a text file
  - To write a character representation of a data type to an output stream, you use the print and *println* methods of the **PrintWriter** class. If you supply an object as an argument, these methods call the *toString* method of the object.
  - To create a delimited text file, you delimit the records or rows in the file with one delimiter, such as a new line character, and you delimit the fields or columns of each record with another delimiter, such as a tab character.
  - To flush all data to the file, you can use a try-with-resources statement to automatically close the stream when you're done using it. You can also use the flush or close methods of the stream to manually flush all data to the file.

- Common methods of the PrintWriter class
  - **print(**argument**)** - Writes the character representation of the argument type to the file.
  - **println(**argument**)** - Writes the character representation of the argument type to the file followed by the new line character. If the autoflush feature is turned on, this also flushes the buffer.
  - **flush()** - throws IOException - Flushes any data that's in the buffer to the file.
  - **close()** - throws IOException - Flushes any data that's in the buffer to the file and closes the stream.
- Code that appends a string and an object to a text file

```
// open an output stream for appending to the text fie
PrintWriter out = new PrintWrite(new BufferedWriter(new FileWriter("log.txt", true)));
// write a string and an object to the file
out.print("This application was run on ");
LocalDateTime currentDateTime = LocalDateTime.now();
out.println(currentDateTime);
// flush data to the file and close the output stream
out.close();
```

- Code that writes a Product object to a delimited text file

```
// open an output stream for overwriting a text file
PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(productsFile)));
//write the Product object to the file
out.print(product.getCode() + "\t");
out.print(product.getDescription() + "\t");
out.print(product.getPrice());
// flush data to the file and close the out stream
out.close();
```

- How to connect a character input stream to a file
  - The **Reader** class is an abstract class that's inherited by all of the classes in the Reader hierarchy. To learn more about the Reader hierarchy, check the documentation for the Java API. All classes in the java.io package that end with Reader are members of the Reader hierarchy.
  - Although you can read files with the **FileReader** class alone, the **BufferedReader** class improves efficiency and provides better methods for reading character input streams.

- A subset of the Reader hierarchy

  Reader <<abstract>>

  BufferedReader

  InputStreamReader

  FileReader

- Classes used to connect to a file with a buffer

  BufferedReader contains the methods for reading data from the stream

  →**FileReader** connects the stream to a file

- Constructors of these classes

  BufferedReader(Reader**)** - throws None

  FileReader(File**)** - throws FileNotFoundException

  FileReader(StringPathName**)** - throws FileNotFoundException

- How to connect a character input stream to a file

  BufferedReader in = new BufferedReader(new FileReader("products.txt"));

- How to read from a text file
- Common methods of the BufferedReader class

  readLine() – trhows - IOException - Reads a line of text and returns it as a string.

  close() – throws - IOException - Closes the input stream and flushes the buffer.
- Code that reads the records in a text file

```
// read the records of the file
String line;
while((line = in.readLine()) != null) {
        System.out.println(line);
}
// close the input stream
in.close();
```

- Code that reads a Product object from a delimited text file

```
// read the next line of the file
String line = in.readLine();
```

- // parse the line into its columns

```
String[] columns = line.split("\t");
String code = columns[0];
String description = columns[1];
String price = columns[2];
```

- // create a Product object from the data in the columns

```
Product p = new Product(code, description, Double.parseDouble(price));
```

- // print some Product object data

```
System.out.println(p.getDescription() + " (" + p.getPriceFormatted() + ")");
```

- // close the input stream

```
in.close();
```

Student Management System