Object Oriented Programming

I/O File

Lab 10

Content

- **File I/O**
  - Writing text files
  - Reading text files
  - Paths and File considerations

File I/O

- **To read data from or write data to a file, we must create one of the Java stream objects and attach it to the file.**

- **A stream is a sequence of data items.**

- **Java has two types of streams:**
  - an input stream.
  - an output stream.

- **An input stream has a source from which the data items come, and an output stream has a destination to which the data items are going.**

Streams

- **Input streams can flow from the keyboard or from a file**

`System.in` is an input stream that connects to the keyboard

- Scanner keyboard = new Scanner(System.in);

- **Output streams can flow to a screen or to a file**

`System.out` is an output stream that connects to the screen

- System.out.println("Output Stream");

- **There are two(2) types of files we may encounter.**

```
TextFiles
```
- Contain human readable information.
- Can be modified using a text editor.
- Also called ASCII files.

```
BinaryFiles
```
- Not human readable information. Designed to be read by a computer.

Writing to Text Files

- **To write to a Text File we need 2 objects**
  - FileOutputStream
  - PrintWriter

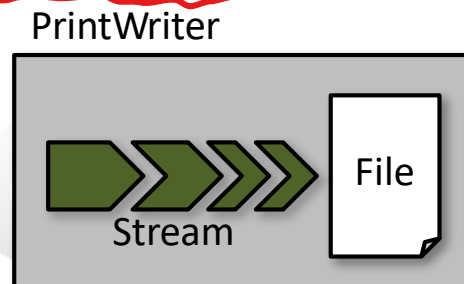 *PrintWriter writer = new PrintWriter(new FileOutputStream("filename.txt"))*

- **FileOutputStream connects to the file and creates a stream.**

  PrintWriter can be used to write text to the stream.

  It is possible that the file may not exist so we must use a try-catch block

- **We must import**
  - java.io.PrintWriter
  - java.io.FileOutputStream
  - java.io.FileNotFoundException

PrintWriter

File

Stream

Writing to Text Files

- **How to write to a text file**

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
// ...
try {
    PrintWriter outputStream = new PrintWriter(new FileOutputStream("myfile.txt"));
    outputStream.println("the quick ...")
    outputStream.close();
} catch (FileNotFoundException e){
    System.out.println("Error opening file");
}
// ...
```

local variable

must

Close the file when finished to flush the write buffer

```java
import java.io.*;

class Ch12TestPrintWriter {
    public static void main (String[] args) throws IOException {

        //set up file and stream
        File outFile = new File("sample.data");
        FileOutputStream outFileStream
                    = new FileOutputStream(outFile);
        PrintWriter outStream = new PrintWriter(outFileStream);

        //write values of primitive data types to the stream
        outStream.println(987654321);
        outStream.println("Hello, world.");
        outStream.println(true);

        //output done, so close the stream
        outStream.close();
    }
}
```

[Handwritten annotations: "error X 출력환불?", "정수", "string", "t개 모두 string", "bool", "string 으로 반환"]

Writing to a Text File

- **When a program is finished writing to a file, it should always close the stream connected to that file** *outputStreamName.close();*

    - This allows the system to release any resources used to connect the stream to the file

    - If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly

    - PrintWriter does not automatically write the data to the file, it just buffers it. Calling flush() tells PrintWriter, and PrintWriter tells FileWriter to write the data to the file.

    - close() also does a flush, so if you call close(), you don't need to call flush().

- **When performing file I/O, a FileNotFoundException can be thrown**
  - In this context it actually means that the file could not be created
  - This type of exception can also be thrown when a program attempts to open a file for reading and there is no such file

- **It is therefore necessary to enclose this code in exception handling blocks**
  - The file should be opened inside a try block
  - A catch block should catch and handle the possible exception

Writing to Text Files

- **PrintWriter allows us to use the methods**
  - print: outputs to the file but does not end the line
  - println: outputs to the file and ends the line.
  - printf: formats output to the file.

Display 10.2   Some Methods of the Class PrintWriter

public void println(*Argument*)

The *Argument* can be a string, character, integer, floating-point number, boolean value, or any combination of these, connected with + signs. The *Argument* can also be any object, although it will not work as desired unless the object has a properly defined toString() method. The *Argument* is output to the file connected to the stream. After the *Argument* has been output, the line ends, and so the next output is sent to the next line.

public void print(*Argument*)

This is the same as println, except that this method does not end the line, so the next output will be on the same line.

public PrintWriter printf(*Arguments*)

This is the same as System.out.printf, except that this method sends output to a text file rather than to the screen. It returns the calling object. However, we have always used printf as a void method.

public void close()

Closes the stream's connection to a file. This method calls flush before closing the file.

public void flush()

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke flush.

Reading from Text Files

- **To read from a Text File we need 2 objects**
  - Scanner
  - FileInputStream

    Scanner scanner = new Scanner(new FileInputStream("filename.txt"));

- **Scanner provides the simplest method of reading from a text file. It is possible that the file may not exist so we must use a try-catch block**

- **We must import**
  - java.util.Scanner
  - java.io.FileInputStream
  - java.io.FileNotFoundException

Scanner
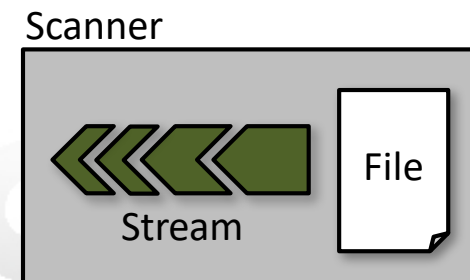
File

Stream

Reading from Text Files

- ## How to read from a text file

import java.util.Scanner;

import java.io.FileInputStream;

import java.io.IOException;

----

try {

    Scanner inputStream = new Scanner(new FileInputStream("myfile.txt"));

    int someInt = inputStream.nextInt();

    String someString = inputStream.nextLine();

    inputStream.close();

} catch (IOException e){

    System.out.println("Error, file not found or may be open.");

}

//...

→ 변수에 값을 저장하기 반드시 close()할필요 X
권장사항

Close the file when finished to release resources.

Sample Textfile Input

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

class Ch12TestScanner {
    public static void main (String[] args) throws IOException {
        //open the Scanner
        File inFile = new File("sample.data");
        FileInputStream inFileStream = new FileInputStream(inFile);
        Scanner scanner = new Scanner(inFileStream);
        //get integer
        int i = scanner.nextInt();

        //Do something with i
        //similar process for other data types
        scanner.close();
    }
}
```

Reading from Text Files

- **Pitfall: if you attempt to read beyond the end of a text file then this will result in the exception NoSuchElementException.**

  - To prevent this from happening we need to test for the end of the file.
  - The methods such as hasNextLine(), hasNextInt(), hasNextByte(), hasNextDouble() are used to determine if more data is available.
  - These may be used in a loop as follows

```
27    while (inputStream.hasNextLine( ))
28    {
29        line = inputStream.nextLine( );
```

File Names

- **The rules for how file names should be formed depend on a given operating system, not Java**
  - When a file name is given to a java constructor for a stream, it is just a string, not a Java identifier (e.g., "fileName.txt")
  - Any suffix used, such as .txt has no special meaning to a Java program
  - it is assumed that the file is in the same directory or folder as the one in which the program is run
  - If it is not in the same directory, the **full or relative path** name must be given

Self-Test (1)

- **프로젝트 명: Project10_1**
  - git commit –m "Project10_1"

- **HasNextLineDemo 클래스를 작성할 것**
  - Scanner와 FileInputStream을 이용하여 BladeRunner.txt를 읽어 들임 (BladeRunner.txt는 프로젝트 폴더에 생성되어 있음)
  - PrintWriter와 FileOutputStream을 이용하여 NumberedRunner.txt를 씀
  - BladeRunner.txt 텍스트 파일 내용의 매 line 마다 맨 앞에 line number를 추가한 뒤 NumberedRunner.txt 텍스트 파일 내용에 써야 함
  - 파일의 끝 부분인지 확인하는 메소드를 활용할 것
  - 프로그램 종료 전에 파일을 close 하여 자원 할당을 해제할 것

Self-Test (1) (Contd)

- **왼쪽은 읽어 들이는 파일 (BladeRunner.txt)**

- **오른쪽은 쓰는 파일 (NumberedRunner.txt)**

Reading from Text Files

- **To read from a Text File with BufferedReader we need 2 objects**
  - FileReader
  - BufferedReader

  BufferedReader bufReader = *new* *BufferedReader(*new* *FileReader("filename.txt"));*

- **FileReader connects to the file and creates a stream.**
  BufferedReader can be used to read from the stream.
  It is possible that the file may not exist so we must use a try-catch block

- **We must import**
  - java.io.BufferedReader
  - java.io.FileReader
  - java.io.IOException
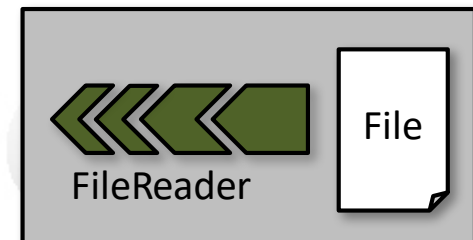
Reading from Text Files

- **How to read from a text file**

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
// …
try {
        BufferedReader bufReader = new BufferedReader(new FileReader("myfile.txt"));
        String str = bufReader.readLine();
        System.out.println(str);
        bufReader.close();
} catch (IOException e){
        System.out.println("Error, file not found or may be open.");
}
// …
```

한점

Close the file when finished to release resources.

Reading from Text Files

- **Pitfall: A program using a BufferedReader may throw two kinds of exceptions**
  - FileNotFoundException
  - IOException.

```
try
{
    BufferedReader inputStream =
        new BufferedReader(new FileReader("morestuff2.txt"));

    String line = inputStream.readLine();

    inputStream.close();
}
catch(FileNotFoundException e)
{
    System.out.println("File morestuff2.txt was not found");
    System.out.println("or could not be opened.");
}
catch(IOException e)
{
    System.out.println("Error reading from morestuff2.txt.");
}
```

```java
import java.io.*;
class Ch12TestBufferedReader {

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

        //set up file and stream
        File inFile = new File("sample3.data");
        FileReader fileReader = new FileReader(inFile);
        BufferedReader bufReader = new BufferedReader(fileReader);
        String str;

        str = bufReader.readLine();
        int i = Integer.parseInt(str);

        //similar process for other types
        bufReader.close();
    }
}
```

Reading Numbers

- **Unlike the Scanner class, the class BufferedReader has no methods to read a number from a text file**
  - Instead, a number must be **read in as a string**, and **then converted to a value** of the appropriate numeric type using one of the wrapper classes.
  - To read in a single number on a line by itself, first use the method **readLine**, and then use **Integer.parseInt**, **Double.parseDouble**, etc. to convert the string into a number.
  - If there are multiple numbers on a line, **StringTokenizer** can be used to decompose the string into tokens, and then the tokens can be converted as described above.

Reading from Text Files

- **BufferedReader allows us to use the methods**
    - read: reads a single character
    - readLine: reads a line. Returns null if the reading beyond eof
    - skip: skips n characters.

Display 10.8   **Some Methods of the Class BufferedReader**

```
public String readLine() throws IOException
```

Reads a line of input from the input stream and returns that line. If the read goes beyond the end of the file, null is returned. (Note that an EOFException is not thrown at the end of a file. The end of a file is signaled by returning null.)

```
public int read() throws IOException
```

Reads a single character from the input stream and returns that character as an int value. If the read goes beyond the end of the file, then −1 is returned. Note that the value is returned as an int. To obtain a char, you must perform a type cast on the value returned. The end of a file is signaled by returning −1. (All of the "real" characters return a positive integer.)

```
public long skip(long n) throws IOException
```

Skips n characters.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

Testing for End of File

- The method **readLine** of the class **BufferedReader** returns **null** when it tries to read beyond the end of a text file.

- A program can test for the end of the file by testing for the value **null** when using **readLine**

- The method **read** of the class **BufferedReader** returns **-1** when it tries to read beyond the end of a text file. A program can test for the end of the file by testing for the value **-1** when using **read**

Path Names

- **The way path names are specified depends on the operating system**
  - A typical UNIX path name that could be used as a file name argument is **"/user/sallyz/data/data.txt"**
  - A **PrintWriter** connected to this file is created as follows:

  PrintWriter outputStream = new PrintWriter(new FileOutputStream ("/user/sallyz/data/data.txt"));

Path Names

- **A typical Windows path name is the following:**

    C:₩dataFiles₩goodData₩data.txt

- **A PrintWriter connected to this file is created as follows:**

    PrintWriter outputStream = new PrintWriter(new FileOutputStream
    ("C:₩₩dataFiles₩₩goodData₩₩data.txt"));

- **Note that in Windows ₩₩ must be used in place of ₩, since a single backslash denotes an the beginning of an escape sequence**

The File Class

- **The File class is like a wrapper class for file names**
  - The constructor for the class **File** takes a name, as a string argument, and produces an object that represents the file with that name
  - The **File** object and methods of the class **File** can be used to determine information about the file and its properties

```java
1   import java.util.Scanner;
2   import java.io.File;
3   import java.io.PrintWriter;
4   import java.io.FileOutputStream;
5   import java.io.FileNotFoundException;
6   public class FileClassDemo
7   {
8       public static void main(String[] args)
9       {
10          Scanner keyboard = new Scanner(System.in);
11          String line = null;
12          String fileName = null;

13          System.out.println("I will store a line of text for you.");
14          System.out.println("Enter the line of text:");
15          line = keyboard.nextLine();
16          System.out.println("Enter a file name to hold the line:");
17          fileName = keyboard.nextLine();
18          File fileObject = new File(fileName);

20          while (fileObject.exists())
21          {
22              System.out.println("There already is a file named "
23                                          + fileName);
24              System.out.println("Enter a different file name:");
25              fileName = keyboard.nextLine();
26              fileObject = new File(fileName);
27          }
```

```
28          PrintWriter outputStream = null;
29          try
30          {
31              outputStream =
32                  new PrintWriter(new FileOutputStream(fileName));
33          }
34          catch(FileNotFoundException e)
35          {
36              System.out.println("Error opening the file " + fileName);
37              System.exit(0);
38          }

39          System.out.println("Writing \"" + line + "\"");
40          System.out.println("to the file" + fileName);
41          outputStream.println(line);

42          outputStream.close();
43          System.out.println("Writing completed.");
44      }
45  }
```

If you wish, you can use `fileObject` *Instead of* `fileName` *as the argument to* `FileOutputStream`*.*

Sample Dialogue

The dialogue assumes that there already is a file named `myLine.txt` *but that there is no file named* `mySaying.txt`*.*

The File Class

```
I will store a line of text for you.
Enter the line of text:
May the hair on your toes grow long and curly.
Enter a file name to hold the line:
myLine.txt
There already is a file named myLine.txt
Enter a different file name:
mySaying.txt
Writing "May the hair on your toes grow long and curly."
to the file mySaying.txt
Writing completed.
```

Some Methods in the Class File

Display 10.12   **Some Methods in the Class** File

File is in the java.io package.

public File(String *File_Name*)

Constructor. *File_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File_Name* is referred to as the **abstract path name**.

public boolean exists()

Tests whether there is a file with the abstract path name.

public boolean canRead()

Tests whether the program can read from the file. Returns true if the file named by the abstract path name exists and is readable by the program; otherwise returns false.

(continued)

Some Methods in the Class File

Display 10.12   Some Methods in the Class File

```java
public boolean setReadOnly()
```

Sets the file represented by the abstract path name to be read only. Returns true if successful; otherwise returns false.

```java
public boolean canWrite()
```

Tests whether the program can write to the file. Returns true if the file named by the abstract path name exists and is writable by the program; otherwise returns false.

```java
public boolean delete()
```

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns true if it was able to delete the file or directory. Returns false if it was unable to delete the file or directory.

(continued)

Display 10.12   **Some Methods in the Class** File

```
public boolean createNewFile() throws IOException
```

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns true if successful, and returns false otherwise.

```
public String getName()
```

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

```
public String getPath()
```

Returns the abstract path name as a String value.

```
public boolean renameTo(File New_Name)
```

Renames the file represented by the abstract path name to New_Name. Returns true if successful; otherwise returns false. New_Name can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

(continued)

- **프로젝트 명: Project10_2**
    - git commit –m "Project10_2"

- **Candidate 클래스는 후보자의 정보와 관련된 데이터와 메소드를 가지고 있음 (이미 작성되어 있음)**
    - 생성자를 통해 투표 수는 random으로 결정됨
    - toString() 메소드는 해당 후보자의 이름과 투표 수를 출력함

- **Calculator 클래스는 Candidate 클래스 타입의 배열을 생성하고 관련 동작을 수행함 (이미 작성되어 있음)**
    - addName(String name): 생성된 Candidate 클래스 타입 배열에 후보자를 추가하는 메소드
    - toStringAllList(): Candidate 클래스 타입 배열에 있는 내용을 문자열로 반환함

Self-Test (2) (Contd)

- **Election 클래스를 작성할 것**

- **candidate.txt 에는 매 line마다 candidate의 이름만 작성되어 있음**
    - Calculator 클래스를 통해 생성된 calculator 객체를 사용하여 candidate.txt 에 작성되어 있는 이름을 객체의 Candidate 클래스 타입 배열 변수에 추가 할 것 (addName 메소드 활용)
    - 파일의 끝 부분인지 확인하는 메소드를 활용할 것
    - 프로그램 종료 전에 파일을 close 하여 자원 할당을 해제할 것

Self-Test (2) (Contd)

- **election.txt에는 투표 결과를 써야 함**
  - 파일 생성은 File 클래스를 통해 이루어짐
  - PrintWriter와 FileOutputStream을 활용할 것
  - Calculator 클래스를 통해 생성된 calculator 객체를 사용하여 election.txt에 투표 결과를 쓸 것 (toStringAllList() 메소드 활용)
  - 프로그램 종료 전에 파일을 close 하여 자원 할당을 해제할 것

Self-Test (2) (Contd)

candidate.txt - 메모장

파일(F)　편집(E)　서식(O)

```
Moon
Hong
Ahn
Yoo
Sim
```

election.txt - 메모장

파일(F)　편집(E)　서식(O

```
Name: Moon
Votes: 76
Name: Hong
Votes: 43
Name: Ahn
Votes: 33
Name: Yoo
Votes: 20
Name: Sim
Votes: 32
```