OBJECT-ORIENTED SYSTEMS DESIGN
[Exercise]: File I/O and Recursion

2022/05/18

Department of Computer Science,
Hanyang University

Lecturer: Taeuk Kim
TA: Taejun Yoon, Seong Hoon Lim, Young Hyun Yoo

File I/O

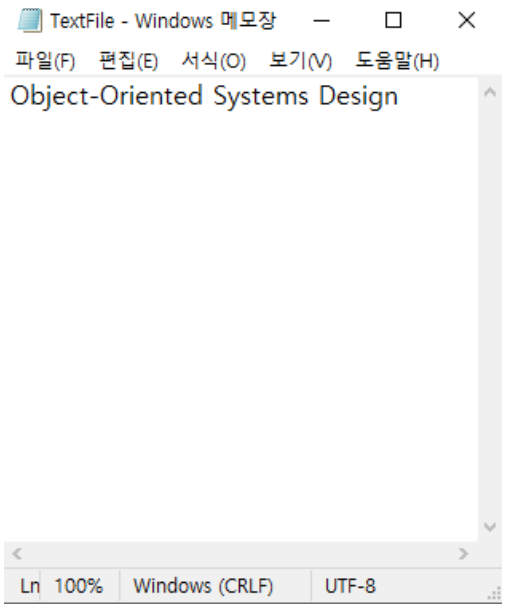
Chapter 10

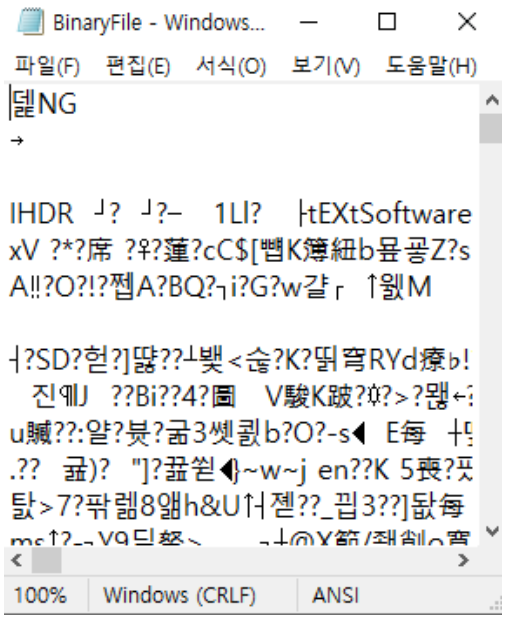
Streams

- **A stream is an object that enables the flow of data between a program and some I/O device or file.**

Text File and Binary File

- **Text File**
  - Files that are designed to be read by human beings, and that can be read or written with an editor are called ***text files.***
- **Binary File**
  - Files that are designed to be read by programs and that consist of a sequence of binary digits are called ***binary files.***



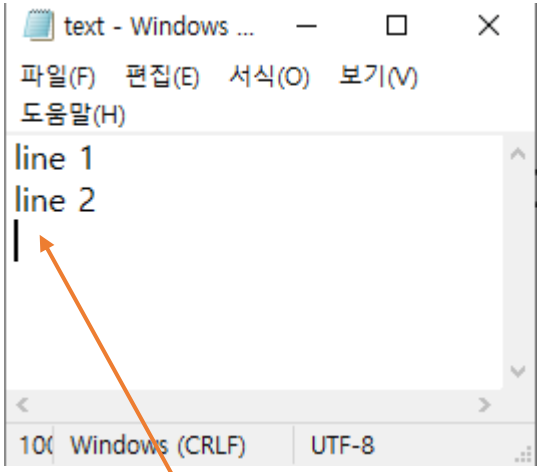


=

Writing to a Text File

- **Using PrintWriter**
  - The class **`PrintWriter`** is a stream class that can be used to write to a text file.

```
public static void main(String[] args) {
    PrintWriter outputStream = null;
    try{
        outputStream = new PrintWriter( new FileOutputStream( name: "text.txt"));
    }catch (FileNotFoundException e){
        System.out.println("Error opening the file text.txt.");
        System.exit( status: 0);
    }
    outputStream.println("line 1");
    outputStream.println("line 2");
    outputStream.close();
}
```



Use **`outputStream`** like **`System.out`**

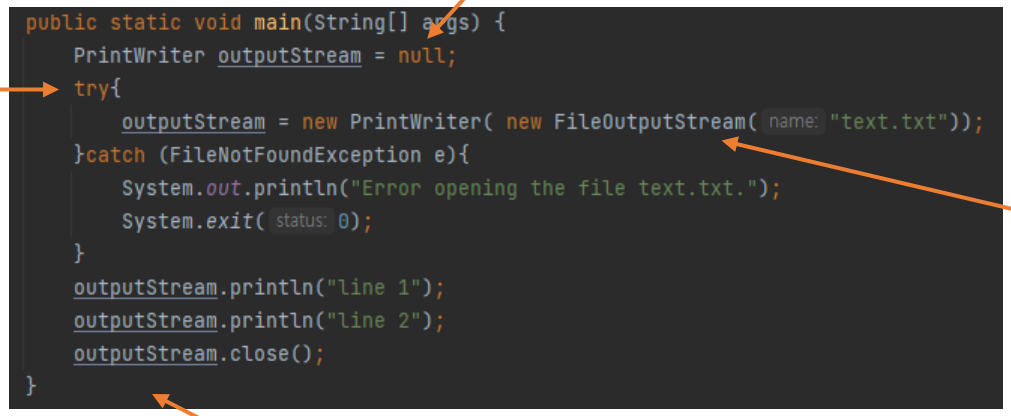
The cursor is on the line 3 because it uses the **`println()`** method.

Writing to a Text File

- **Using PrintWriter**

To use **`PrintWriter`** outside the **`try`** block, you must declare **`PrintWriter`** outside the **`try`** block.

Opening a file can result in an exception, so it should be placed inside a **`try`** block.



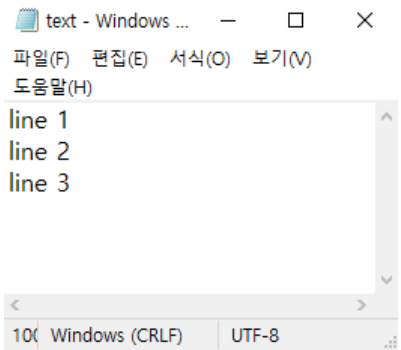
The class **`PrintWriter`** takes **`FileOutputStream`** object as its argument.

When a program is finished writing to a file, it should always close the stream connected to that file.

Appending to a Text File

- To create a `PrintWriter` object and connect it to a text file for ***appending***, a second argument, set to `true`, must be used in the constructor for the `FileOutputStream` object.

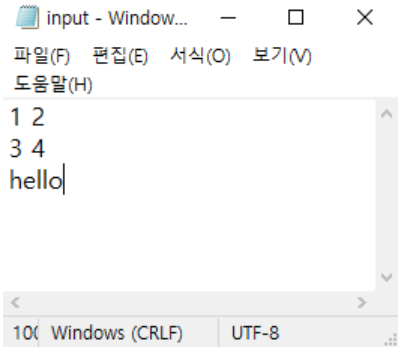
```
public static void main(String[] args) {
    PrintWriter outputStream = null;
    try{
        outputStream = new PrintWriter( new FileOutputStream( name: "text.txt", append: true));
    }catch (FileNotFoundException e){
        System.out.println("Error opening the file text.txt.");
        System.exit( status: 0);
    }
    outputStream.println("line 3");
    outputStream.close();
}
```



Reading From a Text File

- **Using Scanner**
  - The class **`Scanner`** can be used for reading from the keyboard as well as reading from a text file.

```
public static void main(String[] args) {
    Scanner inputStream = null;
    try{
        inputStream = new Scanner(new FileInputStream( name: "input.txt"));
    }catch(FileNotFoundException e){
        System.out.println("input.txt was not found");
    }
    int num1 = inputStream.nextInt();
    int num2 = inputStream.nextInt();
    inputStream.nextLine();
    String line2 = inputStream.nextLine();
    String line3 = inputStream.nextLine();

    System.out.println(+num1+" "+num2);
    System.out.println(line2);
    System.out.println(line3);
```




```
1 2
3 4
hello
```

Reading From a Text File

- **Using Scanner**

Simply replace the argument **`System.in`** (to the **`Scanner`** constructor) with a suitable stream that is connected to the text file.

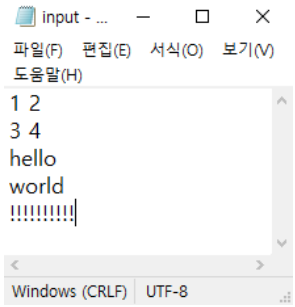
```
public static void main(String[] args) {
    Scanner inputStream = null;
    try{
        inputStream = new Scanner(new FileInputStream( name: "input.txt"));
    }catch(FileNotFoundException e){
        System.out.println("input.txt was not found");
    }
    int num1 = inputStream.nextInt();
    int num2 = inputStream.nextInt();
    inputStream.nextLine();
    String line2 = inputStream.nextLine();
    String line3 = inputStream.nextLine();

    System.out.println(+num1+" "+num2);
    System.out.println(line2);
    System.out.println(line3);
```

The **`nextInt()`** method of the scanner object accepts only an integer, remaining "\n" in the line.

Testing for the End of a Text File with `Scanner`

- Instead of having to rely on an exception to signal the end of a file, the **`Scanner`** class provides methods such as **`hasNextInt`** and **`hasNextLine`**.

```
Scanner inputStream = null;
try{
    inputStream = new Scanner(new FileInputStream( name: "input.txt"));
}catch(FileNotFoundException e){
    System.out.println("input.txt was not found");
    System.exit( status: 0);
}
String line = null;
while(inputStream.hasNextLine()){
    line = inputStream.nextLine();
    System.out.println(line);
}
inputStream.close();
```




```
1 2
3 4
hello
world
!!!!!!!!!!
```

`hasNextLine()` method returns **`true`** if there are remaining lines to read.

Reading From a Text File

- **Using BufferedReader**
  - The class **`BufferedReader`** is a stream class that can be used to read from a text file.
  - An object of the class **`BufferedReader`** has the methods **`read`** and **`readLine`**.

```
public static void main(String[] args) {
    try{
        BufferedReader inputStream = new BufferedReader(new FileReader( fileName: "input.txt"));
        String line1 = inputStream.readLine();
        String line2 = inputStream.readLine();
        String line3 = inputStream.readLine();
        System.out.println(line1);
        System.out.println(line2);
        System.out.println(line3);
    }catch(FileNotFoundException e){
        System.out.println("input.txt was not found");
        System.exit( status: 0);
    }catch (IOException e){
        System.out.println("Error reading from input.txt");
    }
```

Reading From a Text File

- **Using BufferedReader**

```
public static void main(String[] args) {
    try{
        BufferedReader inputStream = new BufferedReader(new FileReader( fileName: "input.txt"));
        String line1 = inputStream.readLine();
        String line2 = inputStream.readLine();
        String line3 = inputStream.readLine();
        System.out.println(line1);
        System.out.println(line2);
        System.out.println(line3);
    }catch(FileNotFoundException e){
        System.out.println("input.txt was not found");
        System.exit( status: 0);
    }catch (IOException e){
        System.out.println("Error reading from input.txt");
    }
```

If you want to use the **`readline()`** method in the **`BufferedReader`** class, you must handle **`IOException`**.

Reading From a Text File

- **Using BufferedReader**
  - Since the **`readline`** method can only read strings, we should use **`StringTokenizer`** class and **`parseInt`** method to read integers from text file.

```
public static void main(String[] args) {
    try{
        BufferedReader inputStream = new BufferedReader(new FileReader( fileName: "input.txt"));
        String line1 = inputStream.readLine();  //String "1 2"

        StringTokenizer stk = new StringTokenizer(line1);
        String temp = stk.nextToken();          //String "1"
        int num = Integer.parseInt(temp);       //Integer 1

        System.out.println(num);

    }catch(FileNotFoundException e){
        System.out.println("input.txt was not found");
        System.exit( status: 0);
    }catch (IOException e){
        System.out.println("Error reading from input.txt");
    }
```

Recursion

Chapter 11

Recursive Methods

- **A *recursive* method is a method that includes a call to itself.**

- **Recursion is based on the general problem solving technique of breaking down a task into subtasks.**
  - In particular, recursion can be used whenever one subtask is a smaller version of the original task.

A Recursive `void` Method

- **Example**

```
public class RecursionDemo {
    public static void main(String[] args) {
        System.out.println("print recursive hello 3");
        printRecursiveHello( n: 3);

        System.out.println("print recursive hello 2");
        printRecursiveHello( n: 2);

    }
    static void printRecursiveHello(int n){
        if (n >0){
            System.out.println("hello "+n);
            printRecursiveHello( n: n-1);
        }
    }
```

```
print recursive hello 3
hello 3
hello 2
hello 1
print recursive hello 2
hello 2
hello 1
```

A Recursive `void` Method

- **Infinite recursion can cause a `StackOverflowError` exception.**

There is no termination condition (or the base case).

```
public class RecursionDemo {
    public static void main(String[] args) {
        printHelloInfinite();
    }
    static void printHelloInfinite(){
        System.out.println("hello");
        printHelloInfinite();
    }
}
```

```
hello
hello
hello
hello
hello
hello
hello
hello
Exception in thread "main" java.lang.StackOverflowError Create breakpoint
```

Recursive Methods that Return a Value

- **Recursive Methods that Return a Value**
  - Recursion is not limited to **`void`** methods.
  - A recursive method can return a value of any type.

factorial

```
static int factorial(int n){
    if (n==1)
        return 1;
    else
        return n * factorial( n: n-1);
}
```

power

```
static int power(int x, int n){
    if(n>0)
        return power(x, n: n-1)*x;
    else
        return 1;
}
```

Recursion Versus Iteration

- **Generate fibonacci numbers**

Iteration

```
public static int fibonacci(int n){
    int fib2 = 0, fib1=1, fib=0;
    int i;

    if(n==0) {
        return 0;
    }else if (n==1){
        return 1;
    }else{
        for(i =2 ;i <=n; i++){
            fib = fib1+fib2;
            fib2 = fib1;
            fib1 = fib;
        }
    }
    return fib;
}
```

Recursion

```
public static int fibonacciRecursive(int n){
    if(n == 0)
        return 0;
    else if (n==1 || n==2)
        return 1;
    else
        return fibonacciRecursive( n: n-1)+fibonacciRecursive( n: n-2);
}
```

Practice

BinarySearch.java

- **Binary search**
  - Binary search is a search algorithm that finds the position of a target value within a sorted array.
  - Binary search is faster than linear search except for small arrays.
  - The array must be sorted first to be able to apply binary search.

- **Pseudocode for binary search**

```
BINARYSEARCH (A, start, end, x)
 if start <= end
  middle = floor((start+end)/2)
  if A[middle]==x
   return middle

  if A[middle]>x
   return BINARYSEARCH (A, start, middle-1, x)

  if A[middle]<x
   return BINARYSEARCH (A, middle+1, end, x)

 return FALSE // in case, element is not in the array
```

Practice.java

- **input.txt**
  - Download a file (**input.txt**) on LMS.
  - The first 100 lines of the file contain the target numbers to be searched.
  - The remaining 100,000 lines correspond to a sequence of integers (whose ranges up to 10,000,000) sorted in ascending order.

- **Main**
  - Get 100 target numbers from the file.
  - Get a sorted array of 100,000 numbers from the file.
  - Find the indices of target numbers in the sequence using binary search.

```
target: 9812270   index: 98051
target: 4458377   index: 44533
target: 9384461   index: 93805
target: 4534765   index: 45293
target: 4683424   index: 46755
target: 2838903   index: 28382
target: 3469845   index: 34759
target: 2298730   index: 23027
target: 7197003   index: 72044
target: 2098784   index: 21106
target: 6287984   index: 62878
target: 8481299   index: 84903
target: 7040290   index: 70457
```

Time for Practice

Get it started, and ask TAs if you are in a trouble.