Middlesex University London

# Lecture 16

# File writing, reading/ Binary input, output

# OBJECTIVES

- In the end of this lecture you should be able to read from and write to text and binary file using Java
- You will create a file, add data and read data from it in a desire way.

# TODAY:

- File handling in Java/File methods/File Operations in Java
- Scanner
- BufferedWriter
- BufferedReader
- ObjectOutputStream
- ObjectInputStream

# FILE WRITING AND READING

- There are two main ways to write and read a file
  - **String** format
  - **Binary** format
- Data stored in a **text file** are represented in human-readable form.
- Data stored in a **binary file** are represented in binary form.
- You cannot read binary files. Binary files are designed to be read by programs.
- For example, the Java source programs are stored in text files and can be read by a text editor, but the Java classes are stored in binary files and are read by the JVM. The advantage of binary files is that they are more efficient to process than text files.

# FILE HANDLING IN JAVA

- File handling implies how to read from and write to file in Java.
- The basic I/O package: java.io allows to do all Input and Output tasks in Java
- The File () – Java Representation of a File

```
File f = new File ("myfile.txt");
```

# FILE OPERATIONS IN JAVA

- Create a file
- Get File information
- Write to a file
- Read a file

# CREATE A FILE

```
public class CreateFile {
    public static void main(String[] args) {

        try{

            File f = new File ("mytextfile.txt");
            if(f.createNewFile()){
                System.out.println("file created");
            }else{
                System.out.println("file already exists");
            }

        }catch (IOException e){
                System.out.println("file not found");
        }



    }

}
```

It would create a file in the same directories.
Or by using full path
"C:\\Users\\MyName\\mytextfile.txt"
On mac
"/Users/name/ mytextfile.txt"

# JAVA FILE METHODS

| Method name: | Description: |
|---|---|
| canRead() | Tests whether file is readable or not |
| canWrite() | Test whether the file is writable or not |
| createNewFile() | Creates an empty file |
| delete() | Deletes a file |
| exists() | Tests whether the file exists or not |
| getName() | Return the name of the file |
| getAbsolutPath() | Return the absolute path of the file |
| length() | Return the size of the file in bites |
| list() | Return the array of the file in directory |
| mkdir() | Creates a directories |

# GET FILE INFORMATION

```
8  import java.io.File;
   import java.util.ArrayList;
10
   public class FileInformation {
12     public static void main(String[] args) {
13           File f = new File ("mytextfile.txt");
14           if(f.exists()){
15               System.out.println("File name: "+ f.getName());
16               System.out.println("Absolut path "+ f.getAbsolutePath());
17               System.out.println("WriteTable: "+f.canWrite());
18               System.out.println("Readable: "+f.canRead());
19               System.out.println("File size in bytes: "+f.length());
20           }else{
21               System.out.println("file does not exists");
22           }
23     }
24
25 }
26
```

# WRITE TO A FILE

```
 7
 8  import java.io.FileWriter;
 9  import java.io.IOException;
10
   public class WriteToFile {
12      public static void main(String[] args) {
13          try{
              FileWriter writer = new FileWriter("mytextfile.txt");
15            writer.write("Great poem");
16            writer.close();
17             System.out.println("Seccesfully wrote to the file");
18          }catch(IOException e){
19              System.out.println(""+e.toString());
20          }
21
22      }
23
24  }
25
```

# READ A FILE

```
19  */
    public class ReadFromFile {
21      public static void main(String[] args) throws FileNotFoundException {
            ArrayList <String> text = new ArrayList<String> ();
23
24      try{
25              File f = new File ("mytextfile.txt");
                Scanner sc = new Scanner (f);
27
28             while (sc.hasNextLine()){
29                 String data = sc.nextLine();
30                 text.add(data);
31             }
32          sc.close();
33
34          }  catch(IOException e){
35              System.out.println(""+e.toString());
36          }
37
            for (int i = 0; i < text.size(); i++) {
39              System.out.println(text.get(i));
40
41          }
42
43
44
45
46      }
```

# PRINTWRITER

- PrintWriter() - is used to write data to a file in a text format.

```
PrintWriter output = new PrintWriter("myfile.txt");
```

- Advantage of PrintWriter is that it provides a powerful printf method which allows for formatting of the text output
- Disadvantage it is unbuffered which will affect performance on large files

- Example: Check related files

# SCANNER

- Scanner () - also used for reading data from a file

```
Scanner input = new Scanner(new File("myfile.txt" ));
```

  - Scanner class provides an ability to token the values from a file, so methods nextInt will produce a next integer value of the file
  - The Scanner class is unbuffered therefore it will read the data character at the time

- Example: Check related files

# BUFFERED VS UNBUFFERED

- The main difference between buffered and unbuffered reading and writing is the **efficiency**
- The unbuffered reading and writing will use the operating system to perform writing and reading of each character of sequence
  - The input and output hard drive operations consume more CPU time. Therefore they are considered more expensive than CPU operations.
- The buffered reading and writing will first buffer the data, and once the buffer is full it will complete its task either display the text or write it to the file

# BUFFEREDWRITER

- Is used to effectively **write data into a file** by first buffering the data before writing it to the file
- BufferedWriter will save the data using a text format
- BufferedWriter doesn't use tokens so the data needs to be formatted prior to writing.

```
BufferedWriter writer = new BufferedWriter( new FileWriter("myfile.txt",true));
```

- Example: Check related files

# BUFFEREDREADER

- **BufferedReader** has same properties as BufferedWriter but instead of writing the data to a text file it will **read it.**

```
BufferedReader bufferedReader = new BufferedReader(new FileReader ("myfile.txt"));
```

- Example: Check related files

# WHAT IS STREAM?

- A stream can be defined as a sequence of data.


- The **Input**stream is used to **read** data from source
- The **Output**stream is used to **write** data to the destination

# SERIALIZATION

- Object serialization is the process of converting an object into a series of bytes so they can be written to disk
- Object to Disk -> Serialization
- Disk to Object -> Deserialization
- To add this capability to Java Class. It have to implement Serializable in the class definition

```
public class Sandwich implements Serializable
```

# SERIALIZATION

- An Object An object is eligible for serialization if and only if its class implements the java.io.Serializable interface. Serializable is a marker interface (contains no methods) that tell the Java Virtual Machine (JVM) that the objects of this class is ready for being written to and read from a persistent storage or over the network.
- By default, the JVM takes care of the process of writing and reading serializable objects. The serialization/deserialization functionalities are exposed via the following two methods of the object stream classes:
- ObjectOutputStream.**writeObject**(Object): writes a serializable object to the output stream. This method throws NotSerializableException if some object to be serialized does not implement the Serializable interface.
- ObjectInputStream.**readObject**(): reads, constructs and returns an object from the input stream. This method throws ClassNotFoundException if class of a serialized object cannot be found.
- Both methods throw InvalidClassException if something is wrong with a class used by serialization, and throw IOException if an I/O error occurs. Both NotSerializableException and InvalidClassException are sub classes of IOException.

# OBJECTOUTPUTSTREAM

- **ObjectOutputStream** is used for purpose of **saving Java object to a text file**.
- The data is saved in a textfile is in a binary format.
- To save an object to a file, the class object has to implement Serializable interface

```
FileOutputStream fos = new FileOutputStream("object.dat");
ObjectOutputStream output= new ObjectOutputStream(fos);
```

```
Sandwich sandwich = new Sandwich("White","Edam");
output.writeObject(sandwich);
```

- Example: Check related files

# OBJECTINPUTSTREAM

- ObjectInputStream is used for purpose of reading java object to a text file in a binary format.
- To read an Java object from a text file, the class object has to implement Serializable interface

```
FileInputStream fio = new FileInputStream( "object.dat" );
ObjectInputStream input= new ObjectInputStream(fio);
Sandwich sandwich = (Sandwich) input.readObject();
System.out.println(sandwich.getBread());
System.out.println(sandwich.getCheese());
```

- Example: Check related files

# MASTERING YOUR SKILLS

- File handling
- PrintWriter
- Scanner
- BufferedWriter
- BufferedReader
- ObjectOutputStream
- ObjectInputStream