CSE215L Programming Language II Lab

Faculty: Dr. Mohammad Rashedur Rahman (RRn)
Lab Manual 18: File I/O
Lab Instructor: Md. Mustafizur Rahman

Objective:

- To know how I/O is processed in Java
- To understand the different types of I/O in Java, i.e. Text I/O, Binary I/O
- To distinguish between text I/O and binary I/O
- To know about the Serialization and Deserialization in Java

Java I/O:

Files can be classified as either text or binary. A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows is called a *text file*. All the other files are called *binary files*. In simple form, a text file consists of a sequence of characters, and a binary file consists of a sequence of bits. However, Java provides many classes for performing *text I/O* and *binary I/O*.

Text I/O vs. Binary I/O: Computers do not differentiate between binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding. Encoding and decoding are automatically performed for text I/O. On the contrary, binary I/O does not involve encoding or decoding and thus is more efficient than the text I/O.

The File Class: The *File* class contains the methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory. But, the *File* class does not contain the methods for reading and writing file contents. Also, constructing a *File* instance does not create a file on the machine. The following code demonstrates how to create a *File* object and use the methods in the *File* class to obtain its properties.

FileClass.java

```java
import java.io.File;
import java.util.Date;

public class FileClass {
	public static void main(String[] args) {
		File file = new File("src\\manual1\\nsu.jpg");

		System.out.println("Does it exist? " + file.exists());
		System.out.println("The file has " + file.length() + " bytes");
		System.out.println("Can it be read? " + file.canRead());
		System.out.println("Can it be written? " + file.canWrite());
		System.out.println("Is it a directory? " + file.isDirectory());
		System.out.println("Is is a file? " + file.isFile());
		System.out.println("Is it absolute? " + file.isAbsolute());
		System.out.println("Is it hidden? " + file.isHidden());
		System.out.println("Absolute path is " + file.getAbsolutePath());
		System.out.println("Last modified on " +
					new Date(file.lastModified()));
	}
}
```

Text I/O: Text data are read using the *Scanner* class and written using the *PrintWriter* or *FileWriter* class.

File Input (reading data): The *Scanner* class can be used to read text data from a file. The following code shows how to create an instance of the *Scanner* class and reads data from the file *info_1.txt*.

ReadDataMain1.java

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

public class ReadDataMain1 {
	public static void main(String[] args) {
		File file = new File("src\\manual2\\info_1.txt");

		try {
			Scanner input = new Scanner(file);

			while(input.hasNext()) {
				String firstName = input.next();
				String lastName = input.next();
				int id = input.nextInt();

				System.out.println("Name: " + firstName + " " +
							lastName + "\tID: " + id);
			}
```

Info_1.txt

```
Tony Stark 1510001042
Steve Rogers 1430001042
Thor Odinson 1310001042
Natasha Romanoff 1610001042
Stephen Strange 1520001042
```

```
			input.close();
		}
		catch(Exception e) {
			e.printStackTrace();
		}
	}
}
```

Sometimes reading text data from a file depends on how the data is oriented in that file. For example, the following code shows how to deal with the file *info_2.txt* properly. Otherwise, it'll throw an exception called *InputMismatchException*.

ReadDataMain2.java

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

```
public class ReadDataMain2 {
    public static void main(String[] args) throws Exception {
        File file = new File("src\\manual2\\info_2.txt");
        Scanner input = new Scanner(file);

        while(input.hasNext()) {
            String name = input.nextLine();
            int id = input.nextInt();

            System.out.println("Name: " + name + "\tID: " + id);
```

Info_2.txt

```
Tony Stark
1510001042
Steve Rogers
1430001042
Thor Odinson
1310001042
```

```
            if(input.hasNext())
                input.nextLine();
        }

        input.close();
    }
}
```

File Output (writing data): The *PrintWriter* class can be used to create a file and write data to a text file. Following code shows how to create an instance of *PrintWriter* class and write data to the file *about_1.txt*.

WriteDataMain1.java

```
import java.io.*;

public class WriteDataMain1 {
    public static void main(String[] args) {
        String name[] = {"Tony Stark", "Steve Rogers", "Thor Odinson"};
        int id[] = {1510001042, 1430001042, 1310001042};

        File file = new File("src\\manual3\\about_1.txt");

        try {
            PrintWriter output = new PrintWriter(file);

            for(int i=0; i<3; i++) {
                output.println(name[i]);
                output.println(id[i]);
            }

            output.close();
        }
        catch(Exception e) {
            e.printStackTrace();
```

```
                }
            }
}
```

The *close()* method must be used to close the file otherwise, the data may not be saved properly. However, JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

try** (declare and create resources)* ***{
Use the resource to process the file;
}

The following code shows how to use try-with-resource syntax.

WriteDataMain2.java

```java
import java.io.*;

public class WriteDataMain2 {
	public static void main(String[] args) throws Exception {
		String name[] = {"Tony Stark", "Steve Rogers", "Thor Odinson"};
		int id[] = {1510001042, 1430001042, 1310001042};

		File file = new File("src\\manual3\\about_2.txt");

		try(PrintWriter output = new PrintWriter(file);){
			for(int i=0; i<3; i++)
				output.println(name[i] + " " + id[i]);
		}
	}
}
```

There is another class available, called *FileWriter* which also can be used to write data to a text file. The difference between *FileWriter* and *PrintWriter* is, *FileWriter* class allows us to append the data in a text file while *PrintWriter* class don't allow to append. Following code shows how to create an instance of *FileWriter* class and write data to the file.

WriteDataMain3.java

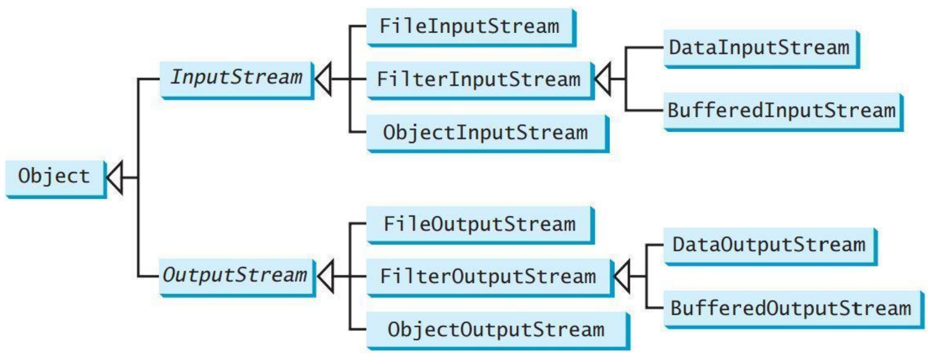
```java
import java.io.*;

public class WriteDataMain3 {
	public static void main(String[] args) throws Exception {
		String name[] = {"Tony Stark", "Steve Rogers", "Thor Odinson"};
		int id[] = {1510001042, 1430001042, 1310001042};

		File file = new File("src\\manual3\\about_3.txt");

		try(FileWriter fw = new FileWriter(file, true);){
			for(int i=0; i<3; i++)
				fw.write(name[i] + " " + id[i] + "\n");
			fw.write("-------------------------\n");
		}
	}
}
```

Binary I/O: Java provides several classes for binary I/O, as shown in the following figure. The abstract *InputStream* is the root class for reading binary data, and the abstract *OutputStream* is the root class for writing binary data.



FileInputStream/FileOutputStream: *FileInputStream* is for reading bytes from files and *FileOutputStream* is for writing bytes to files. Also *FileOutputStream* class allows us to append the data in a binary file by passing true to the append parameter of the constructor. The following code shows how to use binary I/O to write ten byte values from 1 to 10 to a file named *temp.dat* and reads them back from the file.

FileStream.java

```
import java.io.*;

public class FileStream {
	public static void main(String[] args) throws IOException {
		File file = new File("src\\manual4\\temp.dat");

		try (FileOutputStream output = new FileOutputStream(file);){
			for(int i=1; i<=10; i++)
				output.write(i);
		}

		try(FileInputStream input = new FileInputStream(file);){
			int value;
			while((value = input.read()) != -1)
				System.out.print(value + " ");
		}
	}
}
```

DataInputStream/DataOutputStream: *DataInputStream* and *DataOutputStream* are child classes of *FilterInputStream* and *FilterOutputStream* classes respectively. *Filter streams* are streams that filter bytes for some purpose. Unlike basic byte input stream, these filter classes' enable us to read integers, doubles, and strings instead of bytes and characters. So, *DataInputStream* reads bytes from the stream and converts them into appropriate primitive type values or strings. *DataOutputStream* converts primitive type values or strings into bytes and outputs the bytes to the stream. Following code shows how to use *DataInputStream* and *DataOutputStream* classes to filter bytes.

DataStream.java

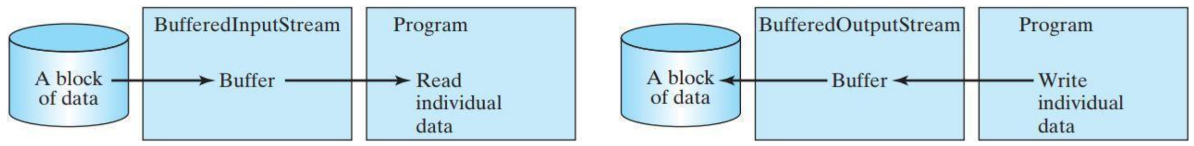
```
import java.io.*;
```

```
public class DataStream {
	public static void main(String[] args) throws IOException {
		File file = new File("src\\manual5\\temp.dat");

		try(
			DataOutputStream output =
				new DataOutputStream(new FileOutputStream(file));
		){
			output.writeUTF("Sujit");
			output.writeInt(24);
			output.writeDouble(50.5);
		}

		try(
			DataInputStream input =
				new DataInputStream(new FileInputStream(file));
		){
			System.out.println(input.readUTF());
			System.out.println(input.readInt());
			System.out.println(input.readDouble());
		}
	}
}
```

BufferedInputStream/BufferedOutputStream:*BufferedInputStream*/*BufferedOutputStream* can be used to speed up input and output by reducing the number of disk reads and writes, as shown in the following figure.



However, we can improve the performance of the *DataStream* program by adding buffers in the stream which is as follows,

```
DataOutputStream output = new DataOutputStream(
             new BufferedOutputStream(new FileOutputStream(file)));

DataInputStream input = new DataInputStream(
             new BufferedInputStream(new FileInputStream(file)));
```

Object I/O, Serialization and Deserialization: *ObjectInputStream*/*ObjectOutputStream* classes can be used to read/write *serializable* objects. However, *Serialization* in Java is a mechanism of writing the state of an object into a byte-stream. And the reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. But, not every object can be written to an output stream. Objects that can be so written are said to be *serializable*. A serializable object is an instance of the *java.io.Serializable* interface, so the object's class must implement Serializable interface for serializing the object. Following code shows how to serialize the object and read/write serializable objects.

Student.java

```
import java.io.Serializable;

public class Student implements Serializable {
	private String name;
```

```
        private int id;

        public Student(String name, int id) {
                this.name = name;
                this.id = id;
        }

        . . . .
}
```

SerializationMain.java

```
import java.io.*;

public class SerializationMain {
        public static void main(String[] args) throws Exception {
                Student obj1 = new Student("Sujit Debnath", 1510000042);
                Student obj2 = new Student("Siyam Chowdhury", 1510001042);

                File file = new File("src\\manual7\\object.dat");

                try(ObjectOutputStream output =
                                new ObjectOutputStream(new FileOutputStream(file));
                ){
                        output.writeObject(obj1);
                        output.writeObject(obj2);
                }

                try(ObjectInputStream input =
                                new ObjectInputStream(new FileInputStream(file));
                ){
                        Student s1 = (Student)input.readObject();
                        Student s2 = (Student)input.readObject();
                        System.out.println(s1.toString());
                        System.out.println(s2.toString());
                }
        }
}
```

Tasks:

1. Write a program that takes integers from the user and writes them into a file until the user inputs a negative number. The program should then read the file and print the sum and average of the numbers.

2. Create a Quiz class with id and mark. Now write a program that reads a file containing records of Quiz objects and initialize an array. The program should then print all the objects in the Quiz array and print the id of the student who obtained the highest mark.
   Sample File:
   ```
   113098 20
   115089 15
   345678 12
   234566 18
   ```
   Program Output:
   ```
   ID:113098 mark:20
   ID:115089 mark:15
   ID:345678 mark:12
   ID:234566 mark:18
   Highest mark obtained by ID:113098
   ```

3. Create an Employee class with name and id. Write 3 Employee objects to a file using serialization. Then using deserialization, read the Employee objects back from the file and print the name and id for each object.