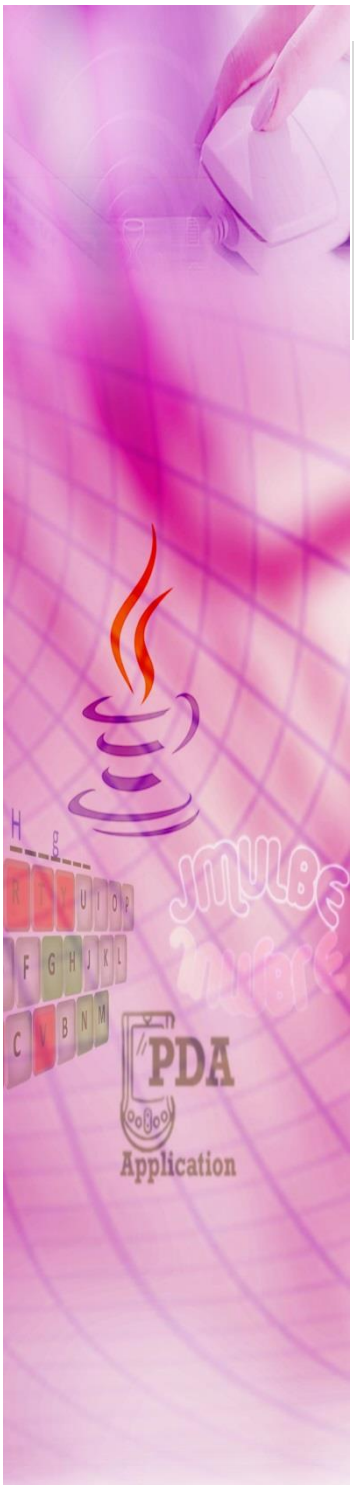


Instructor Inputs

Session 10

Session Overview

This session covers the Working with Output Stream section of Chapter 7 and the following topics of the Introducing NIO section of Chapter 8 of the book, Programming in Java – SG:

- Using the Path Interface and the Paths Class
- Manipulating Files and Directories

In this session, the students will learn the basic concepts of output stream and NIO. In addition, they will learn how to manipulate files and directories by using NIO API.

Working with Output Stream

Handling Tips

Start the session by discussing the scenario of the Banking application. Next, explain the output stream class hierarchy with the help of the diagram given in the SG. Further, while explaining the `FileOutputStream` class, notify the students that if the file in which data needs to be written does not exist, the file will be created before performing the write operation. In addition, explain the various constructors and methods discussed in the SG. Thereafter, while explaining this topic, you can discuss the code given under the Using the FileOutputStream Class subtopic in the Additional Examples topic. Next, while explaining the `BufferedOutputStream` class, notify the students that this class uses buffers to perform write operations. In addition, explain the various constructors and methods discussed in the SG. Further, while explaining this topic, you can discuss the code given under the Using the BufferedOutputStream Class subtopic in the Additional Examples topic. Thereafter, while explaining the `BufferedWriter` class, discuss the various constructors and methods discussed in the SG. Further, while explaining this topic, you can discuss the code given under the Using the BufferedWriter Class subtopic in the Additional Examples topic. Next, while explaining the `FileWriter` class, you can discuss the various constructors and methods discussed in the SG. In addition, while discussing this section, you can discuss the information given in the Additional Inputs topic.

Additional Inputs

ByteArrayOutputStream Class

The `ByteArrayOutputStream` class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it.

DataOutputStream Class

The `DataOutputStream` class lets an application write primitive data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

FilterOutputStream Class

The `FilterOutputStream` class is the superclass of all classes that filter output streams. It is used when you need to buffer, compress/uncompress, and encrypt/decrypt an output stream byte sequence before it reaches its destination. The `FilterOutputStream` class writes the filtered data. The `write()` method in `FilterOutputStream` writes data to the output stream.

Additional Examples

Using the FileOutputStream Class

Consider the following code of the `FileOutputStream` class that demonstrates whether the file that you have provided is a valid file or not:

```
import java.io.*;

public class FileOutputStreamDemo
{
    public static void main(String args[]) throws IOException
    {
        FileDescriptor fd ;
        boolean bool ;
        try (FileOutputStream fos = new FileOutputStream("test.txt"))
        {
            fd = fos.getFD();
            bool = fd.valid();
            System.out.println("Is file valid? " + bool);
        }
        catch (Exception ex)
        {
            System.out.println(ex);
        }
    }
}
```

In the preceding code, the `FileOutputStreamDemo` class has been created. The `getFD()` method is used to return the `FileDescriptor` object that represents the connection to the actual file in the file system being used by this `FileInputStream`. The `valid()` method tests whether the file descriptor object is valid or not. The `System.out.println("IS file valid? " + bool);` statement prints the output.

Using the BufferedOutputStream Class

Consider the following code of the `BufferedOutputStream` class that demonstrates the `write()` method of the `BufferedOutputStream` class:

```
import java.io.*;

public class BufferedOutputStreamDemo
{
    public static void main(String args[]) throws IOException
    {
        ByteArrayOutputStream baos;
        baos = new ByteArrayOutputStream();
        try (BufferedOutputStream bos = new BufferedOutputStream(baos))
        {
            byte[] bytes = {1, 2, 3, 4, 5};
            bos.write(bytes, 1, 3);
            bos.flush();
            for (byte b : baos.toByteArray())
            {
                System.out.print(b);
```



```
            }
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

In the preceding code, the `BufferedOutputStreamDemo` class has been created. The `System.out.print(b);` statement then prints the output.

Using the BufferedWriter Class

Consider the following code that demonstrates how to copy data from one text file to another:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;

public class BufferedWriterDemo {

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

        try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
            try (BufferedWriter bw = new BufferedWriter(new
FileWriter("fileout.txt"))) {

                int i;
                do {
                    i = br.read();
                    if (i != -1) {
                        if (Character.isLowerCase((char) i)) {
                            bw.write(Character.toUpperCase((char) i));
                        } else if (Character.isUpperCase((char) i)) {
                            bw.write(Character.toLowerCase((char) i));
                        } else {
                            bw.write((char) i);
                        }
                    }
                } while (i != -1);
            }
        }
    }
}
```

In the preceding code, the `BufferedReader` class object has been created that reads the data from the file. Further, the object of the `BufferedWriter` class object has been created that writes the data to the file.

Activity 7.1: Working with Input Stream and Output Stream

Handling Tips

Discuss the problem statement with the students.

To perform the activity, 7.1, you need to use the **FileOperation.txt**, **GameMenuFrame.txt**, **GetScoreFrame.txt**, **InputFrame.txt**, **Main.txt**, and **TicTacToeFrame.txt** files, which are provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 07\Activity 7.1\Input Files**

The solution files, **FileOperation. java**, **GameMenuFrame.java**, **GetScoreFrame.java**, **InputFrame.java**, **Main.java**, and **TicTacToeFrame.java**, for this activity are provided at the following location in the TIRM CD:

- **Datafiles For Faculty\Activities\Chapter 07\Activity 7.1\Solution**

Introducing NIO

Handling Tips

Start the session by telling the students that it is an important task to ensure that the I/O operations performed in Java applications are efficient and achieve the maximum processing productivity. Next, tell them that the functionality offered by the `java.io` package for I/O operations had performance-related issues. To overcome the performance-related issues and improve the efficiency of I/O operations, NIO API has been introduced.

Thereafter, tell them that as most of the I/O operations are based on the paths of files, therefore, to facilitate working with the file and directory paths, NIO provides the `Path` interface and the `Paths` class. Then, tell them that in order to work with a path, a reference of the `Path` interface should be created. Further, explain the methods of the `Path` interface, as given in the SG. Thereafter, while explaining this topic, you can discuss the information given under the Using the Path Interface and the Paths Class subtopic in the Addition Inputs topic.

Next, you can tell the students that in order to perform manipulations with the files and directories, they can utilize the `java.nio.file` package. Thereafter, tell them that NIO also facilitates the traversal of a directory structure. Tell them, that a recursive traversal is performed in certain situations, such as whenever there is a requirement to search for files. In order to improve the efficiency of the file traversal process, they can use the `FileVisitor` interface. Thereafter, explain each of the operations, as explained in the SG, and ensure that the students have gained clarity about how to perform each of the operations. While explaining to the students about how to create a file, you can discuss the information given under the Manipulating Files and Directories subtopic under the Additional Inputs topic. After explaining the additional information, you can demonstrate how to retrieve some of the common files by using the code given under the Manipulating Files and Directories subtopic under the Additional Examples topic.

Additional Inputs

Using the Path Interface and the Paths Class

In some cases, you may need to convert the path to a Web browser format. To achieve this, you can use the `toURI()` method of the `Path` class, as shown in the following code:

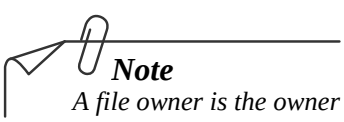
```
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.Paths;

public class URIDemo {

        public static void main(String[] args) {
       Path path = Paths.get("D:/Hello.txt");
        URI path_to_uri = path.toUri();
        System.out.println("Path to URI: " + path_to_uri);
         }
}
```

In the preceding code, `path` is converted to a Web browser format by using the `toUri()` method.

Manipulating Files and Directories

Often there is a requirement to know more details about a file or directory, such as about its visibility and ownership. You can obtain information about the preceding attributes by using the `java.nio.file.attribute` package. The attributes that can be associated with a file or directory are file system dependent. NIO groups the attributes in the form of views. The following views are supported by NIO:

- `BasicFileAttributeView`: This is a view of basic attributes, such as file creation time, last accessed time, and size, which must be supported by all file system implementations.
- `DosFileAttributeView`: This view provides support for standard DOS attributes, namely, readonly, hidden, system, and archive on file systems that support the DOS attributes.
- `PosixFileAttributeView`: This view extends the basic attribute view with attributes supported on file systems that support the Portable Operating System Interface for Unix (POSIX) family of standards, such as Unix.
- `FileOwnerAttributeView`: This view is supported by any file system implementation that supports the concept of a file owner. This view can be used to read or modify the owner of the file.



Note

A file owner is the owner of a file. Every file or directory created on the system has an owner. The file owner has the authority to control and change permissions of the file.

Additional Examples

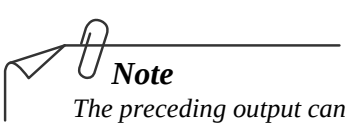
Manipulating Files and Directories

The following code demonstrates how to read attributes of a file by using `BasicFileAttributeView`:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.BasicFileAttributes;
class BasicFileAttributeViewDemo
{

public static void main(String args[])
{
BasicFileAttributes attr = null;
Path path = Paths.get("D:/NIODemo/Hello.java");
try {
attr = Files.readAttributes(path, BasicFileAttributes.class);
} catch (IOException e) {
System.err.println(e);
}
System.out.println("File size: " + attr.size()+" bytes");
System.out.println("File creation time: " + attr.creationTime());
System.out.println("File was last accessed at: " + attr.lastAccessTime());
System.out.println("File was last modified at: " + attr.lastModifiedTime());
System.out.println("Is a directory? " + attr.isDirectory());
System.out.println("Is a regular file? " + attr.isRegularFile());
System.out.println("Is a symbolic link? " + attr.isSymbolicLink());

}
}
```

Once the preceding code is executed, the following output is displayed:

```
File size: 71 bytes
File creation time: 2013-05-19T09:11:25.940355Z
File was last accessed at: 2013-05-19T09:11:25.940355Z
File was last modified at: 2013-05-09T04:56:22.155692Z
Is a directory? false
Is a regular file? true
Is a symbolic link? false
```



Note

The preceding output can vary. In addition, you need to ensure that the D:/NIODemo/Hello.java path exists, else an exception will be raised.

FAQs

- *How are the functionalities of the `Reader/Writer` classes different from those of the `InputStream/OutputStream` classes?*

  Ans: The `Reader/Writer` class hierarchy is character-oriented, and the `InputStream/OutputStream` class hierarchy is byte-oriented.

- *Is it possible to get a list of all the files present in a folder?*

  Ans: Yes. Consider the following code to get a list of files present in a folder:

```
import java.io.*;

public class ListOfFiles
{
    public static void main(String args[]) throws IOException
    {
        String path = ".";
        String files;
        File folder = new File("d://File");
        File[] listOfFiles = folder.listFiles();
        for (int i = 0; i < listOfFiles.length; i++)
        {
            if (listOfFiles[i].isFile())
            {
                files = listOfFiles[i].getName();
                System.out.println(files);
            }
        }
    }
}
```

- *Is it possible to copy the content of one file to another using the classes inside `java.io` package?*

  Ans: Yes, the classes, such as `FileReader` and `FileWriter` can be used to copy the content of one file to another.

- *What is an absolute path?*

  Ans: It is a path that contains the root directory and all other subdirectories that contain a file or folder.

- *How can you determine the file attribute views that are supported by an operating system?*

  Ans. You can determine the file attribute views that are supported by an operating system by using the following code snippet:

```
System.out.println(FileSystems.getDefault().supportedFileAttributeViews());
```

- *How can you hide a file in the Windows operating system by using NIO?*

  Ans: You can use the following code to hide a file in the Windows operating system by using NIO:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.LinkOption.NOFOLLOW_LINKS;
class Demo
{
    public static void main(String args[]) {
        Path path = Paths.get("D:/File1.txt");
        try {
            Files.setAttribute(path, "dos:hidden", true, NOFOLLOW_LINKS);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

  The preceding code hides the `File1.txt` file located in the `D:\` directory. Similarly, if you want to create a read-only file, you need to replace `dos:hidden` with `dos:readonly` in the preceding code.

- *Is the file copy operation in NIO an atomic operation?*

  Ans: No, the file copy operation is not an atomic operation. This operation can result in the `IOException` exception, and then the copy process is aborted.