JAVA Programming

Course Instructor: **Dr. N Nandini Devi**

School of Computer Science

UPES Dehradun

TOPICs to be discussed

- File Handling in Java
  - Introduction to File Class
  - File Operations
    - File Creation
    - Getting File Information
    - Writing to a File
    - Reading from a File
    - Deleting a File
- File Handling with Buffered I/O Classes
- Random Access Files
- File Permissions and Security
- Directory Operations

Let’s START ...!!!

File Class in Java

- In **Java**, with the help of the File class, we can work with files. This File class is inside the java.io package. The File class can be used by creating an object of the class and then specifying the name of the file.
- A File is a named location in the memory used to store related information.
- In simple words, file handling means reading and writing data to a file.

Why File Handling is Required?

- File Handling is an integral part of any programming language as file handling enables us to store the output of any particular program in a file and allows us to perform certain operations on it.

Useful methods in File class

Method Name	Description	Return Type
canRead()	It tests whether the file is readable or not.	Boolean
canWrite()	It tests whether the file is writable or not.	Boolean
createNewFile()	It creates an empty file.	Boolean
delete()	It deletes a file.	Boolean
exists()	It tests whether the file exists or not.	Boolean
length()	Returns the size of the file in bytes.	Long
getName()	Returns the name of the file.	String
list()	Returns an array of the files in the directory.	String[]
mkdir()	Creates a new directory.	Boolean
getAbsolutePath()	Returns the absolute pathname of the file.	String

File Operations

- We can perform the following operation on a file:
  - ❑ Creating a File
  - ❑ Getting File Information
  - ❑ Writing to a File
  - ❑ Reading from a File
  - ❑ Deleting a File

Creating a File

- To create a file in Java, you can use the **createNewFile()** method.
- If the file is successfully created, it will return a Boolean value true and false if the file already exists.

```
import java.io.*;
class MainClass {
    public static void main(String[] args) {
        try {
            //Creating file object
            File f = new File("myfile.txt");
            if(f.createNewFile()){
                //Getting the file name
                System.out.println("File created: " + f.getName());
            }else{
                System.out.println("File already exists.");
            }
        }catch(IOException e){
            System.out.println("An error has occurred.");
            e.printStackTrace();
        }
    }
}
```

Getting File Information

- The operation is performed to get the file information. We use several methods to get the information about the file like name, absolute path, readability, writability, and length.

```
import java.io.File;
class FileInfo {
   public static void main(String[] args) {
      File f = new File("myfile.txt"); //Creating file object
      if(f.exists()){
         //Getting file name
         System.out.println("File Name: " + f.getName());
         //Getting the path of the file
         System.out.println("File path: " + f.getAbsolutePath());
         //Checking whether the file is writable or not
         System.out.println("Is file writeable? " + f.canWrite());
         //Checking whether the file is readable or not
         System.out.println("Is file readable? " + f.canRead());
         //Getting the length of the file in bytes
         System.out.println("The size of the file in bytes is: " + f.length());

      }else{
         System.out.println("The file does not exist.");
      }
   }
}
```

Writing to a File

- We use the FileOutputStream class along with its write() method to write some bytes to the file.
- We use the FileWriter class along with its write() method to write some text to the file.

```
import java.io.*;
class FileOutputStreamDemo{
   public static void main(String[] args) {
      String data = "Java File Writing";

      String filePath = "output.txt";
      try(FileOutputStream fos = new
                FileOutputStream(filePath)){
         fos.write(data.getBytes());
         System.out.println("Data written to
                     the file successfully.");

     }catch(IOException e){
         e.printStackTrace();
     }
  }
}
```

```
import java.io.*;
class FileWriterDemo{
   public static void main(String[] args) {
      String data = "Java File Writing";

      String filePath = "output.txt";
      try(FileWriter fw = new
                FileWriter(filePath)){
         fw.write(data);
         System.out.println("Data written to
                     the file successfully.");

     }catch(IOException e){
         e.printStackTrace();
     }
  }
}
```

Reading from a File

- We use the FileInputStream class along with its read() method to read from the file byte by byte.
- We use the Scanner class to read contents from a file.

```
import java.io.*;
class FileInputStreamDemo{
   public static void main(String[] args) {
      String filePath = "input.txt";
      try(FileInputStream fis = new
                  FileInputStream(filePath)){

         int content;
         while((content = fis.read()) != -1){
                     //Read one byte at a time

            System.out.print((char)content);
         }
      }catch(IOException e){
         e.printStackTrace();
      }
   }
}
```

```
import java.io.*;
Import java.util.Scanner;
class FileReaderDemo{
   public static void main(String[] args) {
      try{
         File Obj = new File("myfile.txt");
         Scanner Reader = new Scanner(Obj);
         while(Reader.hasNextLine()){
            String data = Reader.nextLine();
            System.out.println(data);
         }
         Reader.close();
      }catch(FileNotFoundException e) {
            e.printStackTrace();
      }
   }
}
```

Deleting a File

- We use the delete() method in order to delete a file.

```
import java.io.File;
class MainClass {
   public static void main(String[] args){
      File obj = new File("myfile.txt");           //Creating the file object
      if(obj.delete()) {                                          //Deleting
the file
         System.out.println("Deleted file: "
                           + obj.getName());    //Getting the deleted file name
      }else{
         System.out.println("Failed in
                                          deleting the file.");
//Unsuccessful delete
      }
   }
}
```

File Handling with Buffered I/O

- **Buffered I/O** in **Java** improves efficiency by buffering input and output streams.
- This can be particularly useful when reading from or writing to files, as it minimizes the number of I/O operations.

- **BufferedReader:**

  Reads text from a character-based input stream, buffering characters for efficient reading. The **readLine()** method reads one line at a time, which is efficient for text-based files.

- **BufferedWriter:**

  Writes text to a character-based output stream, buffering characters for efficient writing. The **write()** method writes data, and we can use **newLine()** to add line breaks.

BufferedReader

- **BufferedReader** buffers the input, which reduces the number of I/O operations by reading large chunks of data at once. This makes it more efficient than reading data byte-by-byte or character-by-character.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
class BufferedReaderExample{
    public static void main(String[] args) {
       try (BufferedReader reader = new BufferedReader
                              (new FileReader("input.txt"))) {

          String line;
          while((line = reader.readLine()) != null){
             System.out.println(line);
          }
       }catch(IOException e){
          System.out.println("An unexpected error

                             has occurred");
          e.printStackTrace();
       }
   }
```

BufferedWriter

- **BufferedWriter** buffers the output, reducing the number of I/O operations by writing larger chunks of data at once, which is faster than writing character-by-character.
- It has a **newLine()** method, that allows us to easily add line breaks while writing text files.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
class BufferedWriterExample{
    public static void main(String[] args) {
       try (BufferedWriter writer = new BufferedWriter
                        (new FileWriter("output.txt"))) {

           writer.write("Hello, World!");
           writer.newLine(); //Adds a new line
           writer.write("This is a line written using
                                        BufferedWriter.");

       }catch(IOException e){
           System.out.println("An unexpected error
                                          has occurred.");
           e.printStackTrace();
       }
   }
```

Random Access Files (RAF)

- In **Java, Random Access Files** allow us to read from and write to a file at any specific location. Unlike sequential file handling, **RandomAccessFile** enables non-linear access, meaning you can move to any position in the file and read or write data.

  - **Read and Write:**

    RandomAccessFile supports both reading and writing within the same file, making it versatile for various use cases.

  - **File Pointer:**

    It maintains a file pointer that you can set to any position within the file using the **seek(long pos)** method, allowing random access to file contents.

  - **Modes of Operation:**

    RandomAccessFile has two modes – "r" for read-only access and "rw" for both read and write access.

Random Access Files (Example)

```
import java.io.RandomAccessFile;
import java.io.IOException;
class RandomAccessFileDemo{
   public static void main(String[] args) {
      try(RandomAccessFile sourceFile = new RandomAccessFile("source.txt", "r");
          RandomAccessFile destFile = new RandomAccessFile("destination.txt", "rw")){
         //Set the pointer to the beginning of the source file
          sourceFile.seek(0);
         //Read and write data in chunks for efficiency
         byte[] buffer = new byte[1024];                //Buffer of 1 KB
         int bytesRead;
         while((bytesRead = sourceFile.read(buffer)) != -1){     //Read from source
             destFile.write(buffer, 0, bytesRead);               //Write to destination
         }
         System.out.println("Data copied from source.txt to destination.txt.");
      }catch(IOException e){
         System.out.println("Data copy failed...");
         e.printStackTrace();
      }
   }
}
```

Advantage of RAF over regular File I/O Streams

- Unlike regular file streams, **RandomAccessFile** allows you to move the file pointer to any specific byte offset using **seek()**. This enables you to read from or write to any position in the file without processing all preceding data sequentially.
- **RandomAccessFile** supports both reading and writing in the same instance, unlike FileInputStream and FileOutputStream, which are designed solely for reading or writing.
- With **RandomAccessFile**, you can overwrite data at any point in the file without needing to rewrite the entire file. This is useful for applications like databases or index files where specific records may need to be updated frequently.
- **RandomAccessFile** provides methods to read and write primitive data types (writeInt, writeDouble, writeChar, etc.), making it easier to handle structured binary data directly. In contrast, regular file streams operate primarily with byte data, requiring manual conversion if you're working with structured data.

File Permissions and Security

- The File class in **Java** provides basic methods to control file permissions, such as read, write, and execute. These permissions control how the application can interact with the file but don't directly translate to OS-level user/group permissions.

```
import java.io.File;
class FilePermissionDemo {
   public static void main(String[] args) {
       File file = new File("myFile.txt");
       //Set read-only
       file.setReadOnly(); //Equivalent to removing write permission
       //Allow write
       file.setWritable(true);
       //Allow execute
       file.setExecutable(true);

       //Check permissions
       System.out.println("Can Read: " + file.canRead());
       System.out.println("Can Write: " + file.canWrite());
       System.out.println("Can Execute: " + file.canExecute());
   }
}
```

Directory Operations

- **Java** provides several ways to work with directories using the File class.

```
import java.io.File;
class DirectoryDemo {
   public static void main(String[] args) {
       File dir = new File("myDirectory");
       if(!dir.exists()){
           dir.mkdir();
       }else{
           System.out.println("Directory already exists...");
       }
       String[] files = dir.list();
       if(files != null){
           for(String file:files){
               System.out.println(file);
           }
       }else{
           System.out.println("Directory is empty...");
       }
   }
}
```

Summary

Today, we learned about

- Java File Handling (File class, File operations)
- Efficient File Handling using Buffered I/O classes
- Random Access Files
- File permission and security
- Directory Operations in Java


Thank You!