Unit-05

Using I/O

5.1 Console and File I/O

Console:

Console I/O refers to input and output operations performed through the console or terminal. These operations allow a program to interact with the user via standard input (keyboard) and standard output (scree

In Java, the Scanner class is commonly used for reading input from the console. `System.out.print` and `System.out.println are used` to write output to the console.

Example Programm:

```
import java.util.Scanner;

public class ConsoleInput{

   public static void main(String[] args) {

      Scanner scanner = new Scanner(System.in);

      System.out.print("Enter your name: ");

      String name = scanner.nextLine();

      System.out.print("Enter your age: ");

      int age = scanner.nextInt();

      System.out.println("Hello, " + name + ". You are " + age + " years old.");

      scanner.close();   }  }
```

- **Scanner scanner = new Scanner(System.in);**: Creates a `Scanner` object to read from the standard input.
- **scanner.nextLine();**: Reads a line of text from the console.
- **scanner.nextInt();**: Reads an integer from the console.
- **scanner.close();**: Closes the scanner object to release the resources.

- **System.out.print**: Prints text without a newline at the end.
- **System.out.println**: Prints text with a newline at the end.

File I/O:

File I/O refers to input and output operations performed on files stored in the filesystem. These operations allow a program to read from and write to files.

To write to a file, you can use classes like FileWriter and BufferedWriter. To read from a file, you can use classes like FileReader and BufferedReader.

Example Program:

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;


public class FileReadExample {

    public static void main(String[] args) {

        try {

            FileReader reader = new FileReader("output.txt");

            BufferedReader buffer = new BufferedReader(reader);

            String line;

            while ((line = buffer.readLine()) != null) {

                System.out.println(line);

            }

            buffer.close();

        } catch (IOException e) {

            e.printStackTrace();

        }   }  }
```

- **FileWriter fileWriter = new FileWriter("output.txt");**: Creates a `FileWriter` object to write to the specified file.
- **BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);**: Wraps the `FileWriter` object to buffer the output.
- **bufferedWriter.write("text");**: Writes text to the file.
- **bufferedWriter.newLine();**: Writes a newline character to the file.
- **bufferedWriter.close();**: Closes the writer to release the resources and ensure data is properly saved.

- **FileReader fileReader = new FileReader("input.txt");**: Creates a `FileReader` object to read from the specified file.
- **BufferedReader bufferedReader = new BufferedReader(fileReader);**: Wraps the `FileReader` object to buffer the input.
- **bufferedReader.readLine();**: Reads a line of text from the file.
- **bufferedReader.close();**: Closes the reader to release the resources.

Console I/O allows programs to interact with users through the console, while file I/O allows programs to interact with files stored on the filesystem. Properly handling these operations, including opening and closing files, is crucial to ensure data integrity and resource management.

5.2 Opening and closing files

Opening a File:

Opening a file means preparing it for reading or writing. This process involves specifying the file path and the mode (read, write, append, etc.) in which the file should be accessed.

Closing a File:

Closing a file means terminating the connection to the file and releasing any resources associated with it. This is important to ensure that all data is properly saved and that no resources are leaked, which could lead to file corruption or resource exhaustion.

5.3 Scanner Class

The `Scanner` class in Java is a part of the `java.util` package. It provides methods to read input from various sources, such as the keyboard (standard input), files, strings, and other input streams. The `Scanner` class is particularly useful for parsing primitive types and strings using regular expressions.

Commonly Used Methods

- **Reading Strings and Lines**:
  - `next()`: Reads the next token as a string.
  - `nextLine()`: Reads the next line of input.
- **Reading Primitive Types**:
  - `nextInt()`: Reads the next token as an integer.
  - `nextDouble()`: Reads the next token as a double.
  - `nextBoolean()`: Reads the next token as a boolean.
- **Checking for More Input**:
  - `hasNext()`: Checks if there is another token available.
  - `hasNextInt()`: Checks if the next token can be interpreted as an integer.
  - `hasNextLine()`: Checks if there is another line in the input.

5.4 Byte Streams and Character Streams

In Java, input and output (I/O) operations are performed using streams. Streams represent a sequence of data and can be used to perform operations on files, network connections, or other I/O sources. There are two main types of streams in Java: Byte Streams and Character Streams.

Byte Streams

Byte streams handle I/O of raw binary data. They read and write data in bytes (8-bit). Byte streams are useful for binary data such as images, audio files, and any file format that isn't plain text.

Classes for Byte Streams

1. **InputStream**: The superclass for all byte input streams.
   - **FileInputStream**: Reads bytes from a file.
   - **BufferedInputStream**: Buffers input for efficiency.
   - **DataInputStream**: Allows reading of primitive data types.
2. **OutputStream**: The superclass for all byte output streams.
   - **FileOutputStream**: Writes bytes to a file.
   - **BufferedOutputStream**: Buffers output for efficiency.
   - **DataOutputStream**: Allows writing of primitive data types.

Character Streams

Character streams handle I/O of character data (16-bit Unicode). They are used for reading and writing text files. Character streams automatically handle character encoding and decoding.

Classes for Character Streams

3. **Reader**: The superclass for all character input streams.
    - **FileReader**: Reads characters from a file.
    - **BufferedReader**: Buffers input for efficiency.
    - **InputStreamReader**: Bridges byte streams to character streams.
4. **Writer**: The superclass for all character output streams.
    - **FileWriter**: Writes characters to a file.
    - **BufferedWriter**: Buffers output for efficiency.
    - **OutputStreamWriter**: Bridges byte streams to character streams.

5.5 Reading and Writing Byte Streams

Reading and writing byte streams in Java involves handling raw binary data using the `InputStream` and `OutputStream` classes and their subclasses. Byte streams are suitable for binary data such as images, audio files, and other types of non-text files.

Reading Bytes from a File:

```
import java.io.FileInputStream;

import java.io.IOException;


public class ByteStreamReadExample {

   public static void main(String[] args) {

      try (FileInputStream fis = new FileInputStream("input.dat")) {

         int byteData;

         while ((byteData = fis.read()) != -1) {

            System.out.print((char) byteData);  } }

               catch (IOException e) {

         e.printStackTrace();   }  }  }
```

Writing Bytes to a File:

```
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamWriteExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.dat")) {
            String data = "Hello, World!";
            fos.write(data.getBytes()); }
              catch (IOException e) {
            e.printStackTrace();   }  }  }
```

5.6 Reading and Writing Character Streams

Reading and writing character streams in Java involves handling text data using the `Reader` and `Writer` classes and their subclasses. Character streams are specifically designed to handle 16-bit Unicode characters, making them ideal for text files and character-based data.

Reading Characters from a File:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CharacterStreamReadExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("input.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);  }  }
             catch (IOException e) {
            e.printStackTrace();  }  }   }
```

Writing Characters to a File:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class CharacterStreamWriteExample {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {
            bw.write("Hello, World!");
            bw.newLine();
            bw.write("This is a character stream example.");  }
             catch (IOException e) {
            e.printStackTrace(); } }  }
```

5.7 Random Access Files

Random access files in Java allow you to read from and write to a file at any position, rather than sequentially. This is useful for applications that require frequent updates or access to specific parts of a file without reading it from the beginning. The `RandomAccessFile` class in the `java.io` package provides this functionality.

The some Features of RandomAccessFile:

- **Bidirectional I/O**: You can read and write to the same file.
- **Random Access**: You can move the file pointer to any position within the file.
- **Read/Write Operations**: Supports reading and writing primitive data types and strings.

The different `mode` to access random file:

- `"r"`: Read-only mode.
- `"rw"`: Read and write mode.
- `"rws"`: Read and write mode with synchronous file and metadata updates.

- `"rwd"`: Read and write mode with synchronous file content updates.

Different Methods to access random file:

- **`seek(long pos)`**: Sets the file pointer to the specified position.
- **`read()`**: Reads a byte of data.
- **`read(byte[] b)`**: Reads bytes into an array.
- **`write(int b)`**: Writes a byte of data.
- **`write(byte[] b)`**: Writes bytes from an array.
- **`readInt()`**: Reads an integer.
- **`writeInt(int v)`**: Writes an integer.
- **`close()`**: Closes the random access file.

Writing to a Random Access File:

```
import java.io.IOException;
import java.io.RandomAccessFile;
public class RandomAccessFileWriteExample {
    public static void main(String[] args) {
        try (RandomAccessFile raf = new RandomAccessFile("example.dat", "rw")) {
            // Write data at the beginning of the file
            raf.writeUTF("Hello, World!");
            raf.writeInt(12345);
            raf.seek(0);
            raf.writeUTF("Java I/O");
        } catch (IOException e) {
            e.printStackTrace();   }  }  }
```