L54

Monday, February 7, 2022 12:27 PM

Byte Streams, Character Streams (Readers and Writers) and Object Streams

Byte Streams

Q1. The input and output streams represent sources and destinations from where data can be read and written to. The sources and destinations can be files on a storage device or network sockets or in memory arrays.

The java.io package contains all the main classes related to different kinds of streams which can work on raw bytes, characters and even objects.

The streams which work on **bytes** are called **byte streams**. The one which work on **character data** are called **readers** and **writers**. Java also provides streams which are capable of reading and writing **Java objects** from and to a persistent storage.

The abstract classes InputStream and OutputStream are the super classes for all byte streams.

InputStream provides the below three methods to read input:

1. read() - it reads the next byte of data from the input stream and returns the value of that byte as an int.
2. read(byte[] byteArr) - it reads some number of bytes from the input stream and stores them into the byte array byteArr and returns the count of bytes read.
3. read(byte[] byteArr, int startOffsetInByteArr, int length) - it tries to reads a maximum of length bytes of data from the input stream into an array of byteArr and returns the actual count of bytes it read.

Among the above three methods we should remember that the count of bytes returned by the 2nd and 3rd methods depend on the bytes available and were read.

Similarly there are three corresponding write methods in the OutputStream:

1. write(int singleByte) - it writes the singleByte of data provided as an int value into the method.
2. write(byte[] byteArr) - it writes all the bytes in the byte array byteArr into the output stream.
3. write(byte[] byteArr, int startOffsetInByteArr, int length) - it writes the length bytes stored in byteArr from startOffsetInByteArr into the output stream.

Both the Input and output streams implement Closeable interface, which contains a close() method.

Closing streams after use is very important to free up resources.

See retype the below code which demonstrates how to use the above mentioned read and write methods.

The code first creates dummy text in a StringBuilder. It then uses a ByteArrayInputStream instance, which wraps the byte array containing the content in the StringBuilder instance and presents it as an input stream. The code uses a FileOutputStream to write to a file.

The contents of the newly written file are once again read and printed to console to verify.

```java
package q11354;
import java.util.*;
import java.io.*;
import java.nio.file.*;
public class ByteStreamsDemo {
    public static void main(String[] args) throws IOException {
        StringBuilder sb = new StringBuilder();
        sb.append("This text was written at 1 time\n");
        for (int i = 2; i <= 10; i++) {
            sb.append("This text was written at " + i + " times\n");
        }
        InputStream bais = new ByteArrayInputStream(sb.toString().getBytes());
        OutputStream fos = null;
        String outputFileName = "ByteStreamsDemo.txt";
        try {
            fos = new FileOutputStream(outputFileName);
            byte[] byteArr = new byte[512];
            int bytesRead = 0;
            while ((bytesRead = bais.read(byteArr)) != -1) {
                fos.write(byteArr, 0, bytesRead);
            }
        } finally {
            if (fos != null) {
                fos.close();
            }
        }
        Path outputFilePath = Paths.get(outputFileName);
        byte[] contentArr = Files.readAllBytes(outputFilePath);
        System.out.println(new String(contentArr));
    }
}

```

Character Streams (Readers and Writers)

Q1. **Byte streams** are not the correct choice to work on character data, when such data has to be read for parsing. We use byte streams mainly to carry content without interpretation.

java.io package has two abstract classes called Reader and Writer which are very similar to InputStream and OutputStream.

These two classes are the super classes for all character streams. Unlike byte streams which work on raw bytes, the readers and writers perform automatic conversion from bytes to characters using the system's local character sets.

The abstract classes InputStream and OutputStream are the super classes for all byte streams.

Reader provides the below three methods to read input:

1. read() - it reads the next char of data from the reader and returns the value of that char as an int.
2. read(char[] charArr) - it reads some number of characters from the reader and stores them into the character array charArr and returns the count of characters read.
3. read(char[] charArr, int startOffsetInByteArr, int length) - it tries to reads a maximum of length characters from the reader into charArr and returns the actual count of characters it read.

In the above methods we should remember that in the second and third methods the count of characters returned depends on the characters available and were read.

Similarly there are three corresponding write methods in the Writer:

1. write(int singleChar) - it writes the singleChar provided as an int value into the method.
2. write(char[] charArr) - it writes all the characters in the character array charArr into the output stream.
3. write(char[] charArr, int startOffsetInByteArr, int length) - it writes the length characters stored in charArr from startOffsetInByteArr into the writer.

Both the Input and output streams implement Closeable interface, which contains a close() method.

Closing streams after use is very important to free up resources. The below code demonstrates the try-with-resources syntax which automatically closes the resources that implement the inteface java.lang.AutoCloseable.

See and retype the below code which demonstrates how to use the above mentioned read and write methods.

The code first creates dummy text in a StringBuilder. It then uses a StringReader instance, which wraps the string containing the content in the StringBuilder instance and presents it as a reader. The code uses a FileWriter to write to a file.

The contents of the newly written file are once again read and printed to console to verify.

```java
package q11355;
import java.util.*;
import java.io.*;
import java.nio.file.*;
public class ReaderWriterDemo {
    public static void main(String[] args) throws IOException {
        StringBuilder sb = new StringBuilder();
        sb.append("This text was written at 1 time\n");
        for (int i = 2; i <= 10; i++) {
            sb.append("This text was written at " + i + " times\n");
        }
        Reader reader = new StringReader(sb.toString());
        String outputFileName = "CharStreamsDemo.txt";
        try (FileWriter fw = new FileWriter(outputFileName)) {
            char[] charsArr = new char[512];
            int charsRead = 0;
            while ((charsRead = reader.read(charsArr)) != -1) {
                fw.write(charsArr, 0, charsRead);
            }
        }
        Path outputFilePath = Paths.get(outputFileName);
        byte[] contentArr = Files.readAllBytes(outputFilePath);
        System.out.println(new String(contentArr));
    }
}

```

Object Streams

Q1. Java provides ObjectInputStream and ObjectOutputStream to read and write Java objects. This process writing is called serialization and the process of reading an open back is called deserialization.

An object of any class which implements an interface called Serializable can participate in serialization.

Serializable does not have any methods to implement. Such an interface is called a **marker interface.**

Almost all the container classes like those present in collections and also String, StringBuilder, the wrapper classes for primitives, Date related classes etc implement this interface to facilitate serialization. Both the ObjectInputStream and ObjectOutputStream store and retrieve objects from an underlying byte stream.

Among the various methods the main method in ObjectInputStream to read an object is readObject().

Similarly, the main method in ObjectOutputStream to write an object is writeObject().

See and retype the below code.

Please note that if we do not want a field of a class to be persisted we should mark that field with the transient keyword. For example in the below code you will notice that the fields seatingPosiiton and comments will not have their original values once stored and retrieved.

If a class does not implement Serializable and it is attempted to be persisted using writeObject method, the method throws NotSerializableException, which is a runtime exception.

```java
package q11356;
import java.io.*;
public class SerializationDemo {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Student st1 = new Student("CT1007", "Ganga", 25, 71, "Hard-Working");
        Student st2 = new Student("CT1008", "Yamuna", 26, 51, "Absent-Minded");
        String outputFileName = "ObjectStreamsDemo.txt";
        Student restoredSt1 = null;
        Student restoredSt2 = null;
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(outputFileName));
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream(outputFileName))) {
            System.out.println("Before serialization st1 : " + st1);
            System.out.println("Before serialization st2 : " + st2);
            oos.writeObject(st1);
            oos.writeObject(st2);
            restoredSt1 = (Student) ois.readObject();
            restoredSt2 = (Student) ois.readObject();
        }
        System.out.println("After deserialization st1 : " + restoredSt1);
        System.out.println("After deserialization st2 : " + restoredSt2);
    }
}
class Student implements Serializable {
    private String id;
    private String name;
    private int age;
    private transient int seatingPosition;
    private transient String comments;
    public Student(String id, String name, int age, int seatingPosition, String comments) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.seatingPosition = seatingPosition;
        this.comments = comments;
    }
    public String toString() {
        return "Student[ id=" + id + ", name=" + name + ", age=" + age + ", seatingPosition=" + seatingPosition + ", comments=" + comments + " ]";
    }
}

```