**INTERNITY FOUNDATION**

**TASK 2**

**Submitted By:**

**Amisha Singhal**

**Java Batch**

**Java I/O Fundamentals  
Read and write data from the console**

### Ans- Console object

Access to the console depends upon the underlying platform and also upon the way in which the Java program is invoked. If it's started from the command line without redirecting the standard input and output streams, then its console will exist.

If there's a console, it will be available with the **System.console()** method. If it's not available, an invocation of that method will return null.

**Reading a line of text from the console:**

**Console console = null;**

**String s = null;**

**try**

**{**

**console = System.console();**

**if (console != null)**

**{**

**s = console.readLine();**

**System.out.println(s);**

**}**

**} catch (Exception ex)**

**{**

**ex.printStackTrace();**

**}**

**For writing,** we have two equivalent methods that accept a [format string](https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax):

**Console console = System.console();**

**if (console != null)**

**{**

**console.format("[%s]", "Hi");**

**console.printf("[%s]", "Hi again");**

**console.format("Or just hi");**

**}**

### Standard Streams

Java supports three standard streams: Standard Input, accessed through System.in; Standard Output, accessed through System.out; and Standard Error, accessed through System.err. These objects are defined automatically and do not need to be opened.

**To read using streams:**

**try {**

**BufferedReader bufferRead = new BufferedReader(new InputStreamReader(System.in));**

**String s = bufferRead.readLine();**

**System.out.println(s);**

**}**

**catch(IOException ex)**

**{**

**ex.printStackTrace();**

**}**

To write, standard output and standard error are both for output and they are defined as PrintStream objects:

**System.out.println("Hi");**

**System.out.print("Hi again\n");**

**Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the**[**java.io**](http://java.io/)**package.**

**Ans-**[**File**](https://docs.oracle.com/javase/8/docs/api/java/io/File.html)**:** This class is a representation of a file or a directory pathnames. Its instances are immutable, that is, once created, the abstract pathname represented by the object will never change.

You can create an instance this way:

**File file = new File("c:\\file.txt");**

[**FileReader**](https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html) **and** [**FileWriter**](https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html)**:** FileReader and FileWriter are character-based, they are intended for reading and writing text.

**Eg for FileReader:**

**Reader reader = new FileReader("c:\\file.txt");**

**int data = reader.read();**

**while(data != -1){**

**char dataChar = (char) data;**

**data = reader.read();**

**}**

**reader.close();**

**Eg for FileWriter:**

**Writer writer = new FileWriter("c:\\file.txt");**

**writer.write("Hello World Writer");**

**writer.close();**

FileReader extends from InputStreamReader, so it can work with an InputStream. At the same time, FileWriter extends from OutputStreamReader, so it can work with an OutputStream.

[**FileInputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html)

FileInputStream reads the contents of a file as a stream of bytes. It's a subclass of InputStream. It can be created either with a String or a File object that represent a path.

**Eg:**

InputStream input = new FileInputStream("c:\\text.txt");

int byteData = input.read();

while(byteData != -1) {

byteData = input.read();

}

input.close();

[**FileOutputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html)

FileOutputStream writes the contents of a file as a stream of bytes. It's a subclass of OutputStream. It can be created either with a String or a File object that represent a path.

**OutputStream fos1 = new FileOutputStream("c:\\text.txt"); //overwrites file**

**OutputStream fos2 = new FileOutputStream("c:\\text.txt", true); //appends to file**

**OutputStream fos3 = new FileOutputStream("c:\\text.txt", false); //overwrites file**

[**BufferedReader**](https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html)

BufferedReader provides buffering to a character-input stream. Rather than read one byte at a time, BufferedReader reads a larger block at a time. To add buffering to an InputStream just wrap it in a BufferedInputStream:

Reader r = new BufferedReader(new FileReader("c:\\file.txt"));

[**BufferedWriter**](https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html)

BufferedWriter provides buffering to a character-output stream. To add buffering to a Writer just wrap it in a BufferedWriter:

BufferedWriter input = new BufferedWriter(new FileWriter("c:\\file.txt"));

[**ObjectInputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html)

ObjectInputStream allows you to read serialized objects (their class must implement java.io.Serializable) from an InputStream instead of only bytes. You wrap an InputStream in an ObjectInputStream like this:

**class Test implements java.io.Serializable {}**

**ObjectInputStream input = new ObjectInputStream(new FileInputStream("obj.data"));**

**Test object = (Test) input.readObject();**

**input.close();**

[**ObjectOutputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html)

ObjectOutputStream allows you to write serialized objects (their class must implement java.io.Serializable) from an OutputStream instead of only bytes. You wrap an OutputStream in an ObjectOutputStream like this:

**class Test implements java.io.Serializable {}**

**ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream("obj.data"));**

**Test object = new Test();**

**output.writeObject(object);**

**output.close();**

[**PrintWriter**](https://docs.oracle.com/javase/8/docs/api/java/io/PrinterWriter.html)

PrintWriter class allows you to write formatted data or data from primitive types instead of bytes to an underlying Writer.

**Eg:**

**PrintWriter writer = new PrintWriter(new FileWriter("c:\\file.txt") );**

**writer.print(true);**

**writer.print(1);**

**writer.print(1.23);**

**// printf and format methods do the same**

**writer.printf("%s", "Hi");**

**writer.format("%s", "Hi");**

**writer.close();**

**Java File I/O (NIO.2)**

**Use Path interface to operate on file and directory paths**

**Ans-** The [**Path**](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html) interface defines an object that represents the path to a file or a directory. Path stores the name elements as a sequence.

A Path object has a root component and a hierarchical sequence of names separated by backslashes (in Windows) or slashes (Unix/Linux).

To create a Path object you can use the get method of the Paths class like this:

**Path p1 = Paths.get("c:\\file.txt"); // using an absolute path**

**Path p2 = Paths.get("c:\\data", "examples\\file.txt");**

**Use Files class to check, read, delete, copy, move, manage metadata of a file or directory**

**Ans-** The class [Files](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html) contains static methods that operate on files and directories.

### Check files

The **Files.exists()** method checks if a given Path exists in the file system,

**Eg:**

**Path path = Paths.get("data/logging.properties");**

**boolean exists = Files.exists(path, new LinkOption[]{ LinkOption.NOFOLLOW\_LINKS});**

The LinkOption parameter is used to indicate how symbolic links are handled if the file is a symbolic link. By default, symbolic links are followed.

### Read files

The methods for reading a file are:

**public static byte[] readAllBytes(Path path) throws IOException**

Read all lines from a file. Bytes from the file are decoded into characters using the specified charset.

**public static List<String> readAllLines(Path path) throws IOException**

### Delete files

The **Files.delete()** method can delete a file or directory.

**Eg:** **Path path = Paths.get("file.txt");**

**try {**

**Files.delete(path);**

**} catch (IOException e) {**

**// deleting failed**

**e.printStackTrace();**

**}**

This method will only delete a directory if it is empty.

### Copy files

The **Files.copy()** method copies a file from one path to another. If the destination file already exists, a **java.nio.file.FileAlreadyExistsException is** thrown:

**Path source = Paths.get("file.txt");**

**Path target = Paths.get("file-copy.txt");**

**try {**

**Files.copy(source, destination);**

**} catch(FileAlreadyExistsException fae) {**

**fae.printStackTrace();**

**} catch (IOException e) {**

**// something else went wrong**

**e.printStackTrace();**

**}**

**Move files**

**Files.move()** moves a file. Moving a file is the same as renaming it, but moving an both move it to a different directory and change its name in the same operation. If the destination file already exists, a **java.nio.file.FileAlreadyExistsException** is thrown:

**Path source = Paths.get("file.txt");**

**Path target = Paths.get("file-moved.txt");**

**try {**

**Files.move(source, destination);**

**} catch(FileAlreadyExistsException fae) {**

**fae.printStackTrace();**

**} catch (IOException e) {**

**// something else went wrong**

**e.printStackTrace();**

**}**

**Manage metadata**

It can be used to extract information about the file or directory:

**public static boolean isDirectory(Path path, LinkOption... options)**

**Tests whether a file is a directory.**

**public static boolean isExecutable(Path path)**

**Building Database Applications with JDBC. Describe the interfaces that make up the core of  
the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations.**

**Ans-** **JDBC** stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

* **JDBC-ODBC Bridge Driver,**
* **Native Driver,**
* **Network Protocol Driver, and**
* **Thin Driver**

[**Driver**](https://docs.oracle.com/javase/8/docs/api/java/sql/Driver.html)**:** Provides the API for registering and connecting drivers based on JDBC (every driver class must implement it). Generally used only by the DriverManager class

[**DriverManager**](https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html)**:** To makes a connection with a driver.

[**Connection**](https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html)**:** Provides methods for creating statements and managing connections with a data source and their properties.

[**Statement**](https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html)**:** Object used for executing a static SQL statement and returning the results it produces.

[**ResultSet**](https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html)**:** Object used for retrieving and updating the results of a query.

All JDBC drivers implement four JDBC interfaces:

* Driver
* Connection
* Statement
* ResultSet.

The DriverManager class tracks the loaded JDBC drivers and creates the database connections. When a program creates a database connection with the DriverManager.getConnection() method, the DriverManager, in turn, calls the Driver.connect() method. Every JDBC driver must implement the java.sql.Driver interface. So, the JDBC driver's connect() method checks whether the driver URL is correct, and then, returns the Connection within its connect() method.

**Identify the components required to connect to a database using the DriverManager class including the JDBC URL.**

**Ans-** When working with JDBC, the first thing we need to do is to establish

a connection with a data source (like a DBMS). One way is to use the class [**DriverManager**](https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html)**.**

DriverManager connects an application to a data source by using a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the classpath.

A database URL varies depending on the DBMS used. For example, for MySQL it looks like this:

**jdbc:mysql://localhost:3306/data**

Where localhost is the name of the server that is hosting the database, 3306 is the port number and data is the name of the database.

To connect to a data source using DriverManager, we need to get a [Connection](https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html) object with something like this:

**Properties props = new Properties();**

**props.put("user", userName);**

**props.put("password", password);**

**Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/data");**

**Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections.**

**Ans-** To process an SQL statement with JDBC you have to:

1. **Establishing a connection.**
2. **Create a statement.**
3. **Execute the query.**
4. **Process the ResultSet object.**
5. **Close the connection.**

**Code:**

**String sql = "SELECT id, name FROM users WHERE id = ?";**

**List<User> users = new ArrayList<>();**

**try (Connection con = DriverManager.getConnection("jdbc:mysql://localhost/test?user=admin&password=admin12345");**

**PreparedStatement ps = con.prepareStatement(sql);) {**

**ps.setInt(1, 1001);**

**try (ResultSet rs = ps.executeQuery();) {**

**while(rs.next()) {**

**users.add(new User(rs.getInt("id"), rs.getString("name")));**

**}**

**}**

**} catch (SQLException e) {**

**e.printStackTrace();**

**}**

First, a connection with a MySQL database is established.

With a Connection object, you create a Statement object. There are three different kinds of statements:

* [**Statement**](https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html)**.** Used to implement simple SQL statements with no parameters.
* [**PreparedStatement**](https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html)**.** Used for pre compiling SQL statements that might contain input parameters. It extends Statement.
* [**CallableStatement**](https://docs.oracle.com/javase/8/docs/api/java/sql/CallableStatement.html)**.** Used to execute stored procedures that may contain both input and output parameters. It extends PreparedStatement.

To execute a query, call an execute method from Statement such as:

* **execute().** Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet.
* **executeQuery().** Returns one ResultSet object.
* **executeUpdate().** Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.

With a ResultSet object, you can access the data. It acts as a cursor, pointing to one row of data and positioned before the first row at the beginning. Then you call, for example, the method next() to move the cursor forward by one row an

d you can get the data with getter methods that either take the column index (the first column is 1) or the column name. There are getter methods for a lot of types, for example;

**int getInt(String columnName);**

**long getLong(int columnIndex);**

**long getLong(String columnName);**

**String getString(int columnIndex);**

**String getString(String columnName);**

**BigDecimal getBigDecimal(int columnIndex);**

**BigDecimal getBigDecimal(String columnName);**

When you are finished, call the method Statement.close() to immediately release the resources it's using. When you call this method, its ResultSet objects are also closed. If you're not going to need the connection anymore, you should also close the connection with Connection.close(). You can use a try-with-resources statement to automatically close Connection, Statement, and ResultSet objects, regardless of whether an SQLException has been thrown. Or you can close them manually in the finally block like this:

**try {**

**Connection con = ...**

**Statement stmt = ...**

**// Do something with stmt**

**} catch(Exception e) {**

**e.printStackTrace();**

**} finally {**

**if (stmt != null) { stmt.close(); }**

**if (con != null) { con.close(); }**

**}**

**================================END===============================**