Lesson 13:
Working with Files and Databases

Wholeness of the Lesson

Java provides convenient tools for reading and writing files, and for accessing data stored in a database. The relationship between stored data and an executing program parallels the relationship between awareness and its interaction with the world; that interaction is most successful and rewarding if awareness is broad (corresponding to a well-designed program) and is well integrated with the laws of nature, with ways of manifest existence (JDBC).

File I/O in Java

1. Java supports reading and writing text streams and binary streams. In this lesson we focus on the API for text streams.

2. Background knowleddge: Internally, Java represents each character with the UTF-16 encoding, which is a 16-bit encoding (= 2 bytes represented by 4 hex digits). For instance, in this encoding, 'A' is represented by `\u0041` (which is the pair of bytes 0, 65 in base 10). So, with this way of encoding characters, the String "ABC" would be represented by the byte sequence [0, 65, 0, 66, 0, 67]. However, the default encoding for the Western world is not UTF-16 but ISO-Latin, which encodes "ABC" as [65, 66, 67]. So, though Java's internal represenation of characters uses UTF-16, for purposes of reading and writing text, it uses ISO-Latin encoding as the default.

Readers

1. Reader is the superclass of all "readers" in Java, which offer the ability to read in streams of unicode characters in various convenient ways.

2. InputStreamReader converts raw bytes from some input source to character data, using, by default, the ISO-Latin encoding. BufferedReader organizes data stored in a Reader object to be read in convenient ways.

   Example. Suppose we have a file text.txt containing the line of text "This is test text.".

```
try {
      //System.in(Standard Stream) read input from the keyboard.
      //System.in is a byte stream with no character stream features
      InputStream stream1 = System.in;
      InputStream stream2 = new FileInputStream("text.txt");
      //Buffered input streams read data from a memory area(buffer);
      //Similarly, buffered output streams write data to a buffer.
      BufferedReader reader1 =
         new BufferedReader(new InputStreamReader(stream1));
      BufferedReader reader2 =
         new BufferedReader(new InputStreamReader(stream2));
      System.out.println(reader2.readLine());
      System.out.print("Type something: ");
      System.out.println(reader1.readLine());
      reader1.close();
      reader2.close();
      stream1.close();
      stream2.close();
   }
   catch(IOException e) {
      System.out.println(e.getMessage());
}
```

```
//output
This is test text.
Type something: hi
hi
```

3. If there is no explicit need to convert from raw bytes to characters (as there is when reading from System.in), the concept of an "input stream" is absorbed into the functionality of readers, so the developer never needs to work with the low level of bytes.

   Use the following Examples as models for how to read text files.

   Example:

```
   //uses a FileReader(character stream classes)
try {
      FileReader reader = new FileReader("text.txt");
      BufferedReader bufreader = new BufferedReader(reader);
      String line = null;
      while( (line = bufreader.readLine()) != null){
         System.out.println(line);
      }
      bufreader.close();
      reader.close();
   }
   catch(IOException e) {
      e.printStackTrace();
}
```

Example:

```
   //uses a Scanner
try {
      Scanner sc = new Scanner(new File("text.txt"));
      String line = null;
      while(sc.hasNextLine()) {
         line = sc.nextLine();
         System.out.println(line);
      }
      sc.close();
   }
   catch(IOException e) {
      e.printStackTrace();
}
```

Writers

1. Similarly, there is an OutputStreamWriter that converts raw bytes to a Unicode character stream as output. Convenience methods in a PrintWriter make it possible to format output using print, println, and printf methods, familiar from System.out.

   Example:

```
try {
      OutputStream stream1 = System.out;
      OutputStream stream2 = new FileOutputStream("text2.txt");
      PrintWriter writer1 =
         new PrintWriter(new OutputStreamWriter(stream1));
      PrintWriter writer2 =
         new PrintWriter(new OutputStreamWriter(stream2));
      writer1.println("output to console");
      writer2.println("output to file");
      writer1.close();
      writer2.close();
      stream1.close();
      stream2.close();
   }
   catch(IOException e) {
      System.out.println(e.getMessage());
   }
```

2. When there is no explicit need to work with bytes directly, PrintWriter can be used without reference to the conversion class OutputStreamWriter.

   The following example is a model for writing text data.

   Example:

   // use PrintWriter

```
try {
   PrintWriter writer = new PrintWriter("text3.txt");
   writer.printf("output to %s", "file");
   writer.close();
}
catch(IOException e) {
   System.out.println(e.getMessage());
}
```

Example:

```
// use FileWriter
FileReader inputStream = null;
FileWriter outputStream = null;
try {
   inputStream = new FileReader("source.txt");
   outputStream = new FileWriter("dest.txt");
   int c;
   while ((c = inputStream.read()) != -1) {
      outputStream.write(c);
   }
} catch (IOException e) {
   …
} finally {
   try {
      inputStream.close();
      outputStream.close();
   } catch (IOException e) {
      …
   }

}
```

The `File` Class

1. The `File` class is an abstraction that represents either a file or a directory on the native system's directory system.

2. Methods available in `File` include:

```
boolean isFile
boolean isDirectory
boolean exists
String getAbsolutePath
String getParent
File getParentFile
boolean mkdir
boolean mkdirs
boolean delete
```

MAIN POINT

Reading a File in Java is accomplished by using a FileReader (or Scanner). Writing to a File is accomplished by using a FileWriter. More generally, "input" in human life is handled by the senses; "output" is handled by the organs of action. Both have their source in the field of pure creative intelligence.

Interacting with a Database

1. JDBC provides an API for interacting with a database using SQL – part of the jdk distribution.

2. To interact efficiently with a database, you use the database vendor's *driver* that allows communication between the JVM and the database. Java comes with JavaDB, which is adapted from Apache's Derby dbms.

Set-up steps for using JDBC

1. Install the database software. In this course we will use JavaDb.

2. Install the JDBC driver. This is the platform specific software that the JDBC API uses to communicate with your database. (JavaDb driver is automatically installed when Java is installed.)

   For example, MySql has a custom JDBC *driver* (most database systems provide such a driver for use in Java programs). The configuration step is to add the driver package as a *jar file* to your project.

Coding steps for JDBC

1. Assemble your database url that will tell JDBC the name of your data source– typically, store this database url as a constant.

2. In your code, first load the driver (if the dbms does not provide automatic registration – JavaDB does provide automatic registration).

3. Then create a connection object – you pass in the url at this stage. Typically try to reuse the connection object because it is costly to open up this connection, especially when you do it frequently.

4. Create a Statement object.

5. Prepare a String consisting of an SQL query, and pass it to the Statement instance, using either executeQuery(String sql) or updateQuery(String sql). executeQuery is for reads, updateQuery for updating or inserting.

6. Return value of these query methods in a ResultSet instance. A ResultSet encapsulates the rows of data that have been read. A pointer is (semantically) present that points to position –1 when you first get the ResultSet. Loop through using the next() method, which returns a boolean, to read the rows in the table. Read a row by specifying the column names you wish to look at. The getString(columnName) method returns the value in the current row of the result set having the specified column name, if that value is of String type. Some other options are getInt, getDouble, getDate.

7. When you have finished with the database, close the Statement instance with the close() method. When you are done with the Connection instance, close it with its close() method.

MAIN POINT

JDBC provides an API for interacting with a database using SQL. To interact efficiently with a database, you typically use the database vendor's driver that allows communication between the JVM and the database. This is reminiscent of the Principle of Diving – once the initial conditions have been met, a good dive is automatic. (Here, the initial conditions are correct configuration of the data source and code to load the database driver; once the set up is right, interacting with the database is "effortless".)

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Expansion of consciousness leads to expanded territory of influence

1. Since Java is an OO language, it supports storage and manipulation of data within appropriate objects.
2. To work with real data effectively, Java supports interaction with external data stores (databases) through the use of various JDBC drivers, and the JDBC API.

3. **Transcendental Consciousness**: TC is the field of truth, the field of *Sat.* "Know that by which all else is known."
4. **Wholeness moving within Itself**: In Unity Consciousness, the final truth about life is realized in a single stroke of knowledge.