
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Fundamental Programming Practices (FPP)
Professor Paul Corazza

Lecture 13:
Working with Files and Databases

Wholeness of the Lesson

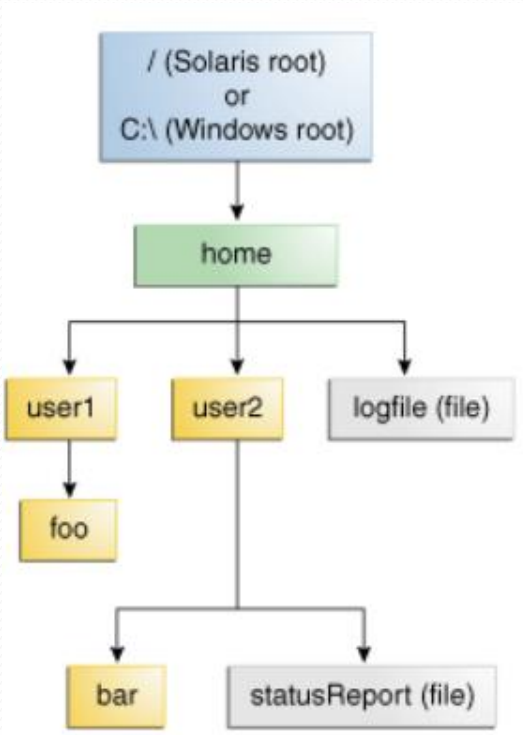
Java provides convenient tools for reading and writing files, and for accessing data stored in a database. The relationship between stored data and an executing program parallels the relationship between awareness and its interaction with the world; that interaction is most successful and rewarding if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

Outline

- File class
- File type and encoding
- I/O Stream concept and classes
- Quick review for DB and SQL
- Work with DB eg. Derby(Java DB)
- Create a JDBC application

The File Class

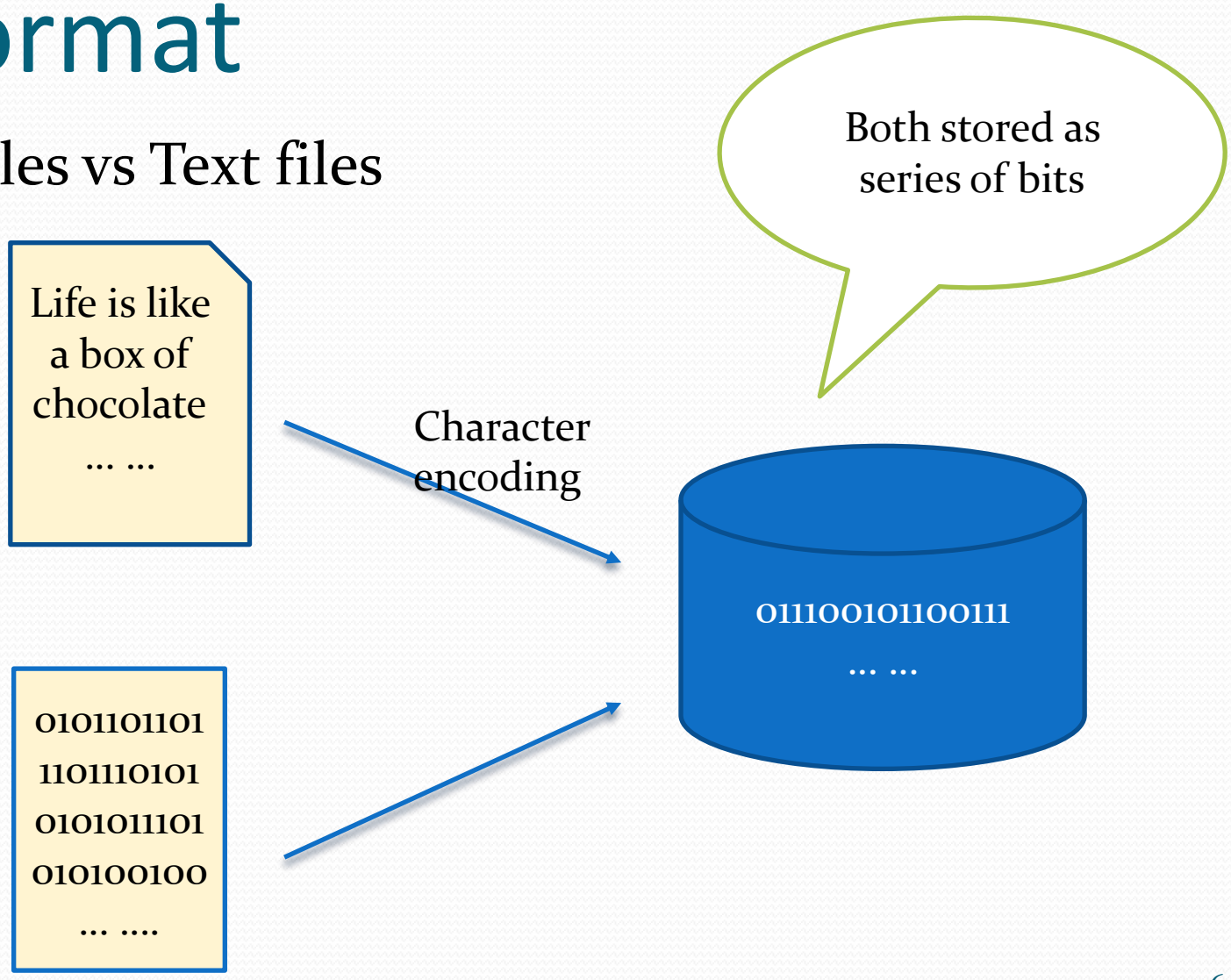
- The `File` class is an abstraction that represents either a file or a directory on the native system's directory system.
- Methods available in `File` include:
  - `boolean isFile`
  - `boolean isDirectory`
  - `boolean exists`
  - `String getAbsolutePath`
  - `String getParent`
  - `File getParentFile`
  - `boolean mkdir`
  - `boolean mkdirs`
  - `boolean delete`



Sample Directory Structure

File Format

- Binary files vs Text files



Character Encoding

- In order for a programming language to interpret byte sequences as characters, it must rely on a *character encoding*. A familiar example of a character encoding is the ASCII table.
- A character encoding matches every character within a certain range to every byte within a certain range. In the ASCII table, the ASCII characters are matched one for one with the byte sequences

0 0 0 0 0 0 0 0 – 0 1 1 1 1 1 1 1

(0 – 127)

Character Encoding – ASC II

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Character Encoding - Unicode

- Unicode uses **a variable number of bytes** per character, such as UTF-8 uses 1 to 4 bytes, UTF-16 uses 1 to 2 16 bit values

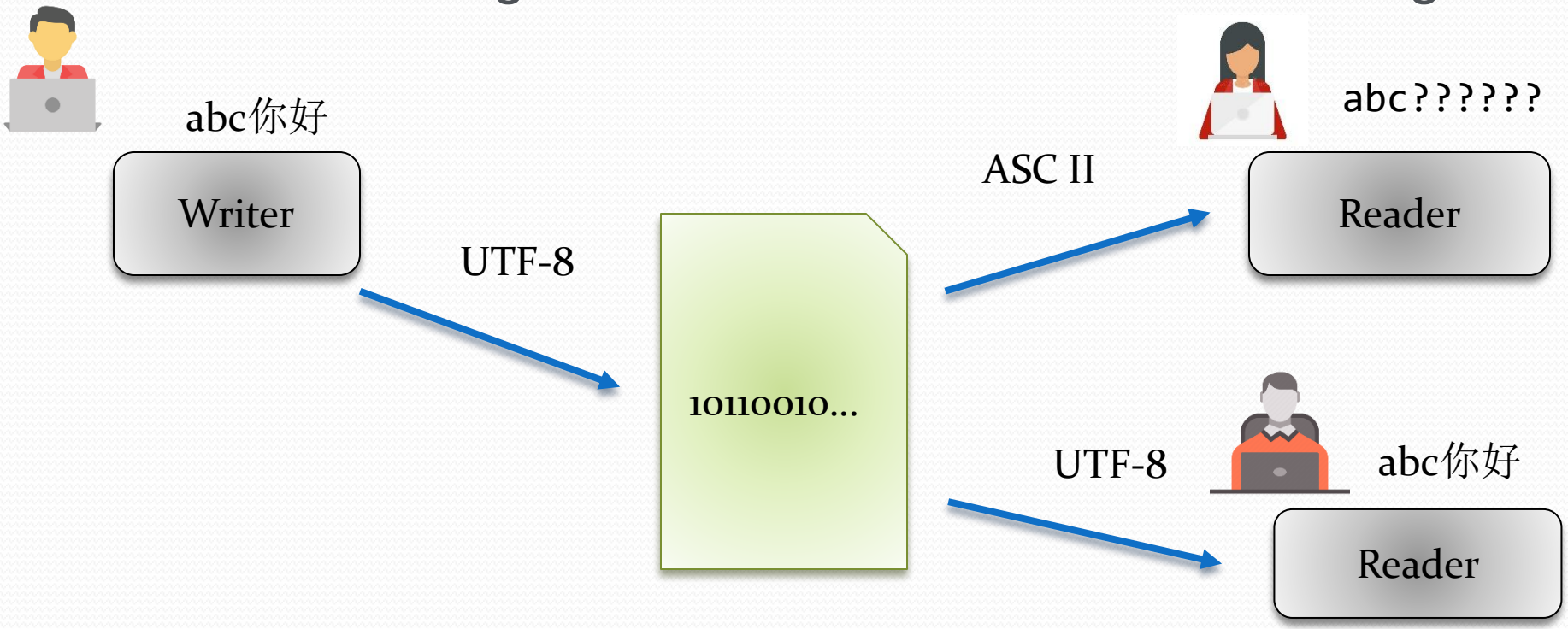
- How to know platform encoding:

  Powershell: [System.Text.Encoding]::Default

  Java: Charset.defaultCharset

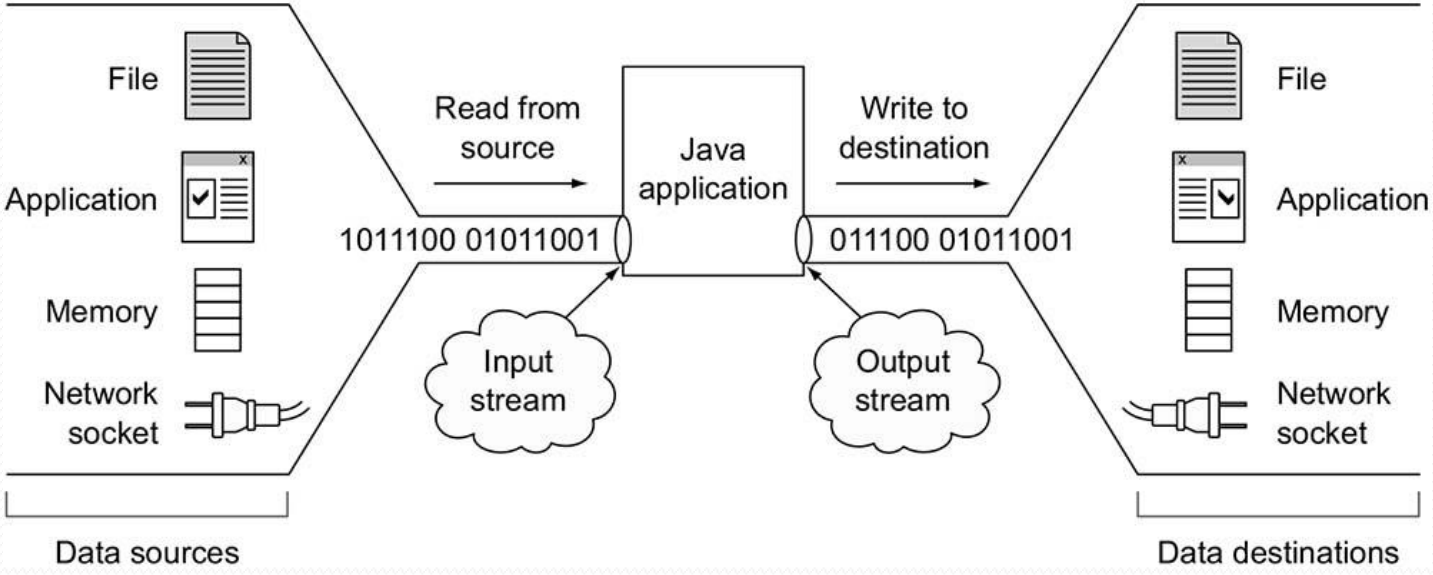
Character Encoding - Mojibake

- The garbled text that is the result of text being decoded using an unintended character encoding.



I/O Stream

- Communication between a Java program and an external device or program is often accomplished using *streams*. A stream is a sequence of bytes.
- Input Stream vs Output Stream
- Character Stream vs Bytes Stream



I/O Stream

- An *input stream* represents data from an input device, like the keyboard for standard input and files that are read from a hard disk.
- An *output stream* represents outbound data directed toward a destination, such as the console (standard output) or a file to be written to disk.

I/O Stream: Character Stream

- A *character stream* is a stream of bytes that has been created using some character encoding (like ISO-8859-1, UTF-8, UTF-16). (Note: UTF-8 and UTF-16 are ways of representing unicode characters). Examples:
  - A text file (created by Notepad for example)
  - Characters entered into *standard input* (the keyboard)

There is no reliable way to detect the characters encoding from a stream of bytes. Some API let you use "default charset" – character encoding preferred by OS. Such as String(byte[]), some use UTF-8 as default Such as Files.readAllLines()

I/O Stream Classes

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

I/O Stream Classes

- Top-level four abstract classes for I/O stream

Abstract class	Character Stream	Byte Stream
Input Stream	Reader	InputStream
Output Stream	Writer	OutputStream

- The other 40+ Java IO stream classes are derived from the four abstract classes. And they're using the four class names as their name suffix.

Eg. FileInputStream, FileWriter

Readers/Writers

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

- `InputStreamReader` converts raw bytes from some input source to character data. `BufferedReader` organizes data stored in a `Reader` object to be read in convenient ways.

(See code on the next slide.)

Reader | Writer Example

```
FileReader in = null;
FileWriter out = null;

try {
    in = new FileReader("book.txt");
    out = new FileWriter("bookCopy.txt");

    int c;
    while ((c = in.read()) != -1) {
        out.write(c);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
FileReader in = null;
FileWriter out = null;

try {
    in = new FileReader("book.txt");
    out = new FileWriter("bookCopy.txt");

    int len;
    char[] buf = new char[5];
    while ((len = in.read(buf)) != -1) {
        out.write(buf, 0, len);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Use fast Reader as needed

- 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 streams. Instead, typically use `BufferedReader` directly.

- All bytes written to the BufferedWriter will first get buffered inside an internal byte array in the BufferedWriter. When the buffer is full, the buffer is flushed to the underlying OutputStreamWriter all at once.

(See code on the next slide.)

Reader | Writer Example

```java
BufferedReader in = null;
BufferedWriter out = null;

try {
    in = new BufferedReader(new FileReader("book.txt"));
    out = new BufferedWriter(new FileWriter("bookCopy.txt"));

    String line;

    while ((line = in.readLine()) != null) {
        out.write(line);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Alternative to Reader

```java
Scanner in = null;
PrintWriter out = null;
try {
    in = new Scanner(new File("book.txt"));
    out = new PrintWriter("bookCopy.txt");

    while (in.hasNextLine()) {
        String line = in.nextLine();
        out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Byte Streams

- All data that is processed by a computer is in the form of sequences of bytes.
  - Examples: Photoshop reads in and writes an image file as a byte stream; similarly for video and audio editors.
- Java makes it possible to work directly with bytes using subclasses of `InputStream` and `OutputStream`.
- Demo shows how to read a file from the hard drive as a byte stream and then output each byte in the file. Output could be in the form of a sequence of base-10 ints; a sequence of length-8 0-1 sequences; or a sequence of hexadecimal pairs.

`lesson13.byte_streams\WorkWithBytes.java`

Character/Bytes Stream Bridge

- You can convert between Unicode character streams and byte streams of non-Unicode text. With the InputStreamReader class, you can convert byte streams to character streams. You use the OutputStreamWriter class to translate character streams into byte streams

- When you create InputStreamReader and OutputStreamWriter objects, you specify the byte encoding that you want to convert. For example, to translate a text file in the UTF-8 encoding into Unicode, you create an InputStreamReader as follows:

```
FileInputStream fis = new FileInputStream("test.txt");
InputStreamReader isr = new InputStreamReader(fis, "UTF8");
```

- If you omit the encoding identifier, InputStreamReader and OutputStreamWriter rely on the default encoding. You can determine which encoding an InputStreamReader or OutputStreamWriter uses by invoking the getEncoding method, as follows:

```
InputStreamReader defaultReader = new InputStreamReader(fis);
String defaultEncoding = defaultReader.getEncoding();
```

Reading Characters from a Byte Stream

- Java's encoding scheme is able to translate ASCII codes to the correct characters, so it is possible to read a text file or read user input from standard input by directly reading the bytes of the stream and converting each byte to a character -- as long as only ASCII characters are used.
  `Demo:lesson13.readWriteEncodings.Main.justAscii`

- However, if any of the bytes in the input stream are non-ASCII, bytes will be rendered as chars using the default encoding, and the resulting chars may not match the original characters
  `Demo: lesson13.readWriteEncodings.Main`

- Not every character (in any encoding) can be represented by single bytes. Example: Chinese characters usually require 2 bytes (in unicode).
  `Demo: lesson13.chars_from_byte_streams.CharsFromBytes`

- Useful Conversions
  (see `lesson13.readWriteEncodings.Main.simple`)

```
//to get the utf-8 bytes (in binary) for 好, use getBytes
printArrayAsBytes("好".getBytes());
//to reassemble the bytes to obtain '好',
//create new String from the byte array (uses utf-8 by default)
System.out.println(new String("好".getBytes()));
//to see the precise unicode value of '好', use getCodePoint
System.out.println("好".codePointAt(0));
//to assemble '好' from the exact unicode value, cast to a char
System.out.println((char)("好".codePointAt(0)));
```

The try-with-resources Statement

- The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement.

- Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

The try-with-resources Statement

```
try (Scanner in = new Scanner(new File("book.txt"));
        PrintWriter out = new PrintWriter("bookCopy.txt");
) {
        while (in.hasNextLine()) {
                String line = in.nextLine();
                out.println(line);
        }
} catch (IOException e) {
        e.printStackTrace();
}
```

The try-with-resources Statement

```
OutputStream output = new FileOutputStream("data.dat");

try(OutputStreamWriter  outputStreamWriter =
    new OutputStreamWriter(output)){

    Person person = new Person();
    person.name = "Jakob Jenkov";
    person.age  = 40;

    outputStreamWriter.writeObject(person);
}
```

Exercise 13.1

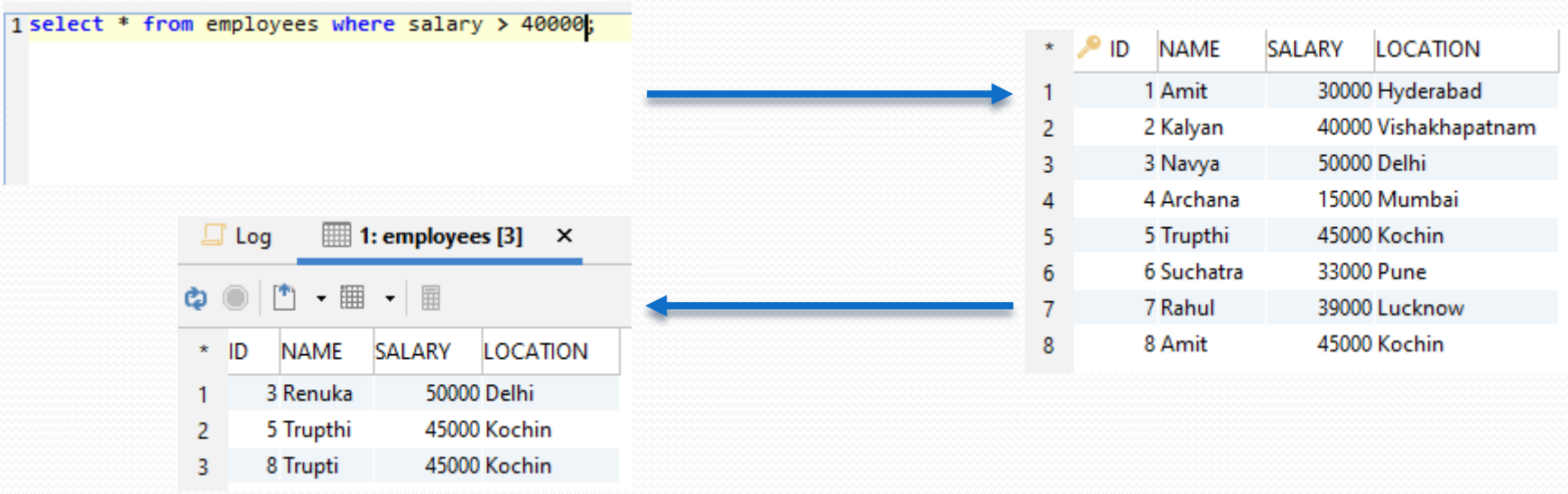
In your `InClassExercises` package, the comments in the `main` method in the `Main` class ask you to write a file to the file system using a PrintWriter and then read the file back in using a BufferedReader.

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.

Quick Relational DB Review

- A database is a means of storing information in such a way that information can be retrieved from it.
- A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database.



SQL Commands

- Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands.

- DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

DDL

- CREATE TABLE — creates a table with the column names the user provides. The user also needs to specify a type for the data in each column.

```
CREATE TABLE Employees(
    Id INT NOT NULL GENERATED ALWAYS AS IDENTITY,
    Name VARCHAR(255),
    Salary INT NOT NULL,
    Location VARCHAR(255),
    PRIMARY KEY (Id))
```

- DROP TABLE — deletes all rows and removes the table definition from the database

```
DROP TABLE Employees
```

- ALTER TABLE — adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

```
ALTER TABLE Employees ADD COLUMN Age INT
```

DML

- SELECT — used to query and display data from a database. The SELECT statement specifies which columns to include in the result set.

```
SELECT * FROM Employees WHERE Salary > 30000;
```

- INSERT — adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.

```
INSERT INTO Employees(Name, Salary, Location) VALUES
('Amit', 30000, 'Hyderabad'),('Trupthi', 45000, 'Kochin’);
```

- DELETE — removes a specified row or set of rows from a table

```
DELETE FROM Employees WHERE Name = 'Trupti';
```

- UPDATE — changes an existing value in a column or group of columns in a table

```
UPDATE Employees SET Location='Chennai’, Salary=43000 WHERE Name = 'Kalyan';
```

Steps for Working with a Database

- Setup DB server(eg. Apache Derby)

1. Download from https://db.apache.org/derby/releases/release-10_15_2_0.cgi ,
2. Unzip to C:\Derby_10

Optional: set DERBY_HOME=C:\Derby_10 and set PATH=%DERBY_HOME%\bin;%PATH%

3. Startup DB server: java -jar %DERBY_HOME%\lib\derbyrun.jar server start | shutdown

Or java –jar C:\Derby_10\lib\derbyrun.jar server start | shutdown

- Obtain your DB driver. A DB driver is provided by the DB vendor and often takes the form of a jar file that is added as an external library. For Apache derby DB, the JDBC driver can be found at C:\Derby_10\lib\derbyclient.jar

- Run your JDBC client program, requires DB driver in classpath

```
java -cp %DERBY_HOME%\lib\derbyclient.jar;. DerbyJDBCDemo.class
```

Or

```
set CLASSPATH=.;%DERBY_HOME%\lib\derbyclient.jar
java DerbyJDBCDemo.class
```

DB Client

- Using DB build-in command line client Eg. In Derby, you can use tool - ij

  java -jar %DERBY_HOME%\lib\derbyrun.jar ij

  Create and open a connection to the database using the client driver.

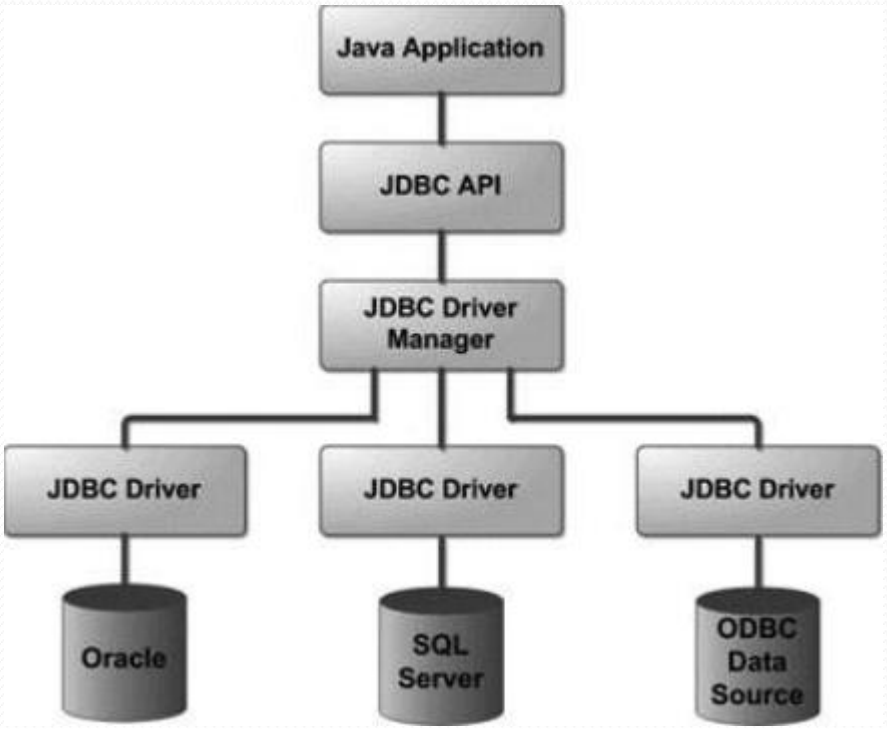
  CONNECT 'jdbc:derby://localhost:1527/testdb;create=true';

- Using third party GUI client Eg. DBVisualizer, Toad, SQuirreL SQL Client

- Write your own DB application with JDBC API.

Interacting with a Database Using JDBC

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.



JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database

2. Send queries and update statements to the database

3. Retrieve and process the results received from the database in answer to your query

JDBC Architecture

- In the two-tier model, a Java applet or application talks directly to the data source.
- In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source.

Figure 1: Two-tier Architecture for Data Access.

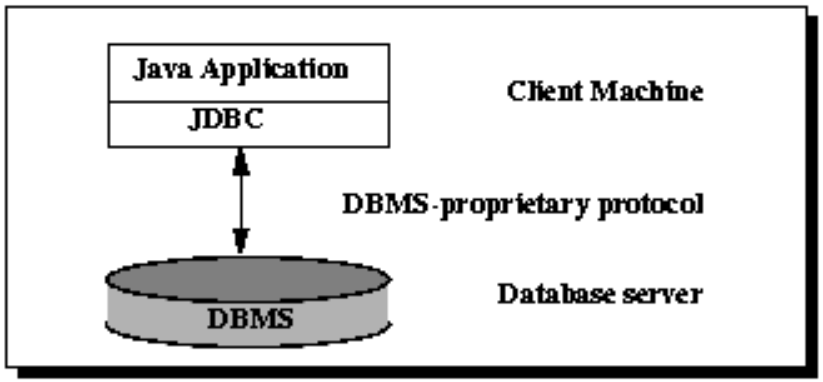
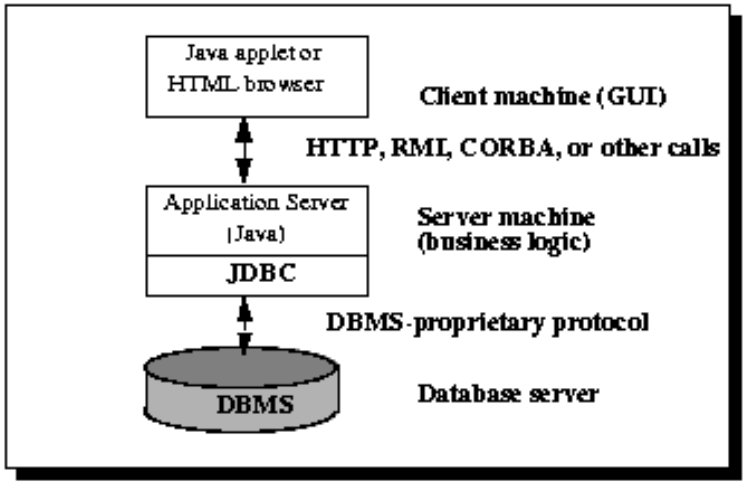


Figure 2: Three-tier Architecture for Data Access.

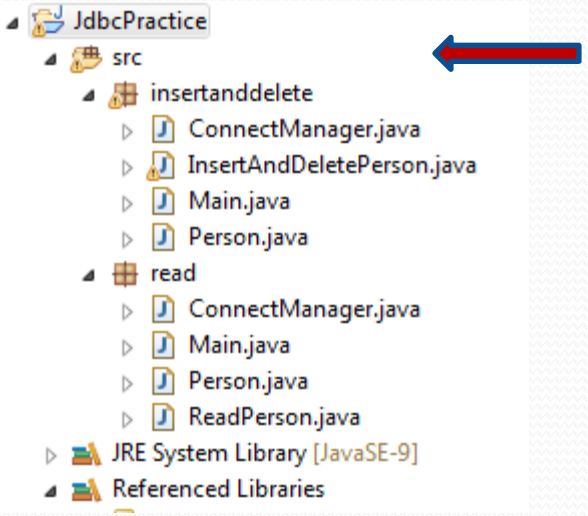


Create a JDBC application

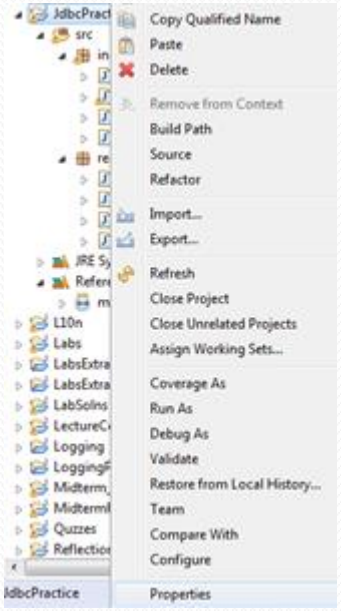
1. Establishing a connection.
2. Create a statement.
3. Execute the query.
4. Process the ResultSet object.
5. Close the connection.

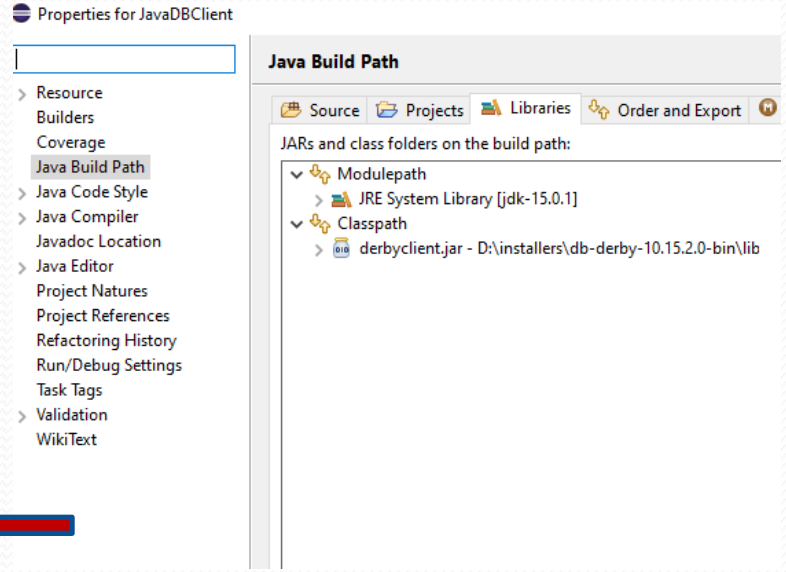
Adding Driver to IDE

- In this course we use the Derby DB
- Add the driver as an external jar to your project



Right click





1.Establishing Connections

- The method DriverManager.getConnection() establishes a database connection. This method requires a database URL, which varies depending on your DBMS. The following are some examples of database URLs:
  - Derby:

  DriverManager.getConnection("jdbc:derby://localhost:1527/testDB;create=true", props);

- Pre JDBC 4.0, to obtain a connection, you had to initialize JDBC Driver by calling Class.forName() to load a java.sql.Driver derived class manually.
  - MySQL: **Class**.forName("com.mysql.cj.jdbc.Driver");
  - Derby: **Class**.forName("org.apache.derby.jdbc.ClientDriver");

2. Creating Statements

- A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.

```
Statement stmt = conn.createStatement();
```

- There are three different kinds of statements:
  - Statement
  - PreparedStatement
  - CallableStatement

Creating a PreparedStatement

```
conn = ConnectManager.getConnection();
String query = "SELECT * FROM Person WHERE firstName = ?";
PreparedStatement stat = conn.prepareStatement(query);
stat.setString(1, firstName);
```

1. Begin by getting the `Connection` object `conn`
2. Be ready with your SQL command
3. The `prepareStatement` method of `Connection` puts your SQL in compiled form. `PreparedStatements` may accept parameters, whose values must be filled in later. Pre-compilation of SQL is a security measure (prevents SQL-Injection attacks)
4. Use the `setString` (and other similar methods) to set parameter values in the `PreparedStatement`
5. The statement is now ready to be executed.

3.Executing Queries

```
stat.executeUpdate()  //for inserts, updates, and deletes
ResultSet rs = stat.executeQuery()    //for reads
```

When a read is done, a `ResultSet` is returned. The client class then unpacks the `ResultSet` to obtain the desired data.

4.Process ResultSet

```
private List<Person> populatePersonList(ResultSet rs) throws SQLException {
    List<Person> list = new ArrayList<>();
    String id = null;
    String firstName = null;
    String lastName = null;
    String ssn = null;
    while(rs.next()) {
        id = rs.getString("id").trim();
        firstName = rs.getString("firstname").trim();
        lastName = rs.getString("lastname").trim();
        ssn = rs.getString("ssn").trim();
        list.add(new Person(id, firstName, lastName, ssn));
    }
    return list;
}
```

5. Closing Connections

- When you are finished using a Connection, Statement, or ResultSet object, call its close method to immediately release the resources it's using.

```
…
if (conn != null) {
        try {
                conn.close();
        } catch (SQLException e) {
                e.printStackTrace();
        }
}
…
```

- Alternatively, use a try-with-resources statement to automatically close Connection, Statement, and ResultSet objects, regardless of whether an SQLException has been thrown

```
try (Statement stmt = conn.createStatement()) {
    // …
  }
```

See Demos

See Java project `JdbcPractice` in your workspace.

Exercise 13.2

The files in the `read` package from the `JdbcProject` have been copied into the package `lesson13.exercise_2`. Add a method call `findStreet()` in the `main` method of `Main` that reads from the `fppdb` database all street names of addresses belonging to persons having `ssn = 535811101`.

Implement by making a call to the `ReadPerson` class; in that class, assign a sql statement to `query4` and implement the (unimplemented) method `getStreetNames(ssn)`, which will execute your `query4` to return the required street names in a `List`.

Hint: Create your query first and try it out on the mysql client. Once your query is correct, write the Java code.

Important: Make sure you have added the mysql driver jar to the `InClassExercises` project and that your mysql server is running.

Solution

```
mysql> select street from address a, person p where p.ssn='535811101' and p.id=a.id;
+--------------+
| street       |
+--------------+
| 10 Adams St. |
+--------------+
1 row in set (0.00 sec)
```

//Snippet from class `ReadPerson`, inside `getStreetNames()`:

```
conn = ConnectManager.getConnection();
PreparedStatement stat = conn.prepareStatement(query4);
stat.setString(1, ssn);
ResultSet rs = stat.executeQuery();
return populateStreetList(rs);
    ...
```

//Method from class `ReadPerson`:

```
private List<String> populateStreetList(ResultSet rs) throws SQLException {
    List<String> streetNames = new ArrayList<>();
    while(rs.next()) {
        streetNames.add(rs.getString("street"));
    }
    return streetNames;
}
```

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." -- Upanishads
4. **Wholeness moving within itself**: In Unity Consciousness, the final truth about life is realized in a single stroke of knowledge.