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).

Overview

- An **InputStreamReader** is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.
- **FileReader**: Reads text from character files. It Inherits from InputStreamReader. Need to give the name of the file to read. FileNotFoundException if the named file does not exist.
- An **OutputStreamWriter** is a bridge from character streams to byte streams. Characters written to it are encoded into bytes using charset. Inherits from Writer class. It's an Abstract class.
- **PrintWriter** Prints formatted representations of objects to a text-output stream.
- To read character streams, use a subclass of **Reader**. To write character streams, use a subclass of **Writer**

Java I/O: Character Streams

- 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, which represent all characters using 20-bit codes).
- A text file (created by Notepad for example)
  - Characters entered into *standard input* (the keyboard)
- <u>Overview</u>: In practice, to read character streams, use a subclass of `Reader`. To write character streams, use a subclass of `Writer` (rather than an `OutputStream`).
- All sample code for `Readers` and `Writers` can be found at

  `lesson13.readersandwriters.Main`

Readers

- `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 (recall characters are 16 bit in Java), using, by default, UTF-8 encoding (as of Java 9).

- `BufferedReader` organizes data stored in a `Reader` object to be read in convenient ways and reads character streams very efficiently.

  (See code on the next slide.)

```java
try {
    InputStreamReader is = new InputStreamReader(System.in);
    BufferedReader reader = new BufferedReader(is);
    System.out.print("Type something: ");
    System.out.println(reader.readLine());
    is.close();
    reader.close();
}
catch(IOException e) {
    System.out.println(e.getMessage());
}
```

//output

```
Type something: hi
hi
```

- 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.

Example. We have a file `text.txt` containing the line of text

"This is a Chinese character: 你."

Next slide shows a couple of ways of reading this file.

Example:

```
//uses a FileReader
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: (alternative to Readers)

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

Output in each case:

```
 This is a Chinese character: 你
```

Writers

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

Example: (using an OutputStreamWriter)

```
try {
     OutputStreamWriter os = new OutputStreamWriter(System.out));
     PrintWriter writer = new PrintWriter(os);
      writer.println("output to console with chinese: 你");
     os.close();
     writer.close();
} catch(IOException e) {
     System.out.println(e.getMessage());
}
```

Example: (using a FileWriter)

```
try {
     FileWriter fw = new FileWriter("text2.txt");
     PrintWriter pw = new  PrintWriter(fw);
     pw.println("output to file with chinese: 你");
     fw.close();
     pw.close();
} catch(IOException e) {
     System.out.println(e.getMessage());


}
```

Introduction to Data Streams in Java

- **Definition**: Data streams in Java support binary I/O of primitive data types (boolean, char, byte, short, int, long, float, double) and String values.
- This contrasts with character-based streams, which handle text data. Data streams are particularly useful when you need to store or transmit data efficiently, as binary representation is generally more compact than text.
- **Key Classes**:
  - DataInputStream: Allows an application to read primitive Java data types from an underlying input stream.
  - DataOutputStream: Enables an application to write primitive Java data types to an output stream in a portable manner.
- Refer: lesson13\datastream

Introduction to Data Streams in Java

Reading Data: Use DataInputStream to read data types in the same order they were written.

Methods include readInt(), readDouble(), readUTF(), etc.

```
DataInputStream dis  = new DataInputStream(new FileInputStream("data.dat"))
                int i = dis.readInt();
                double d = dis.readDouble();
                String s = dis.readUTF(); // For Strings
```

Writing Data: Use DataOutputStream to write data types in a consistent format.

Methods include writeInt(), writeDouble(), writeUTF(), etc.

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.dat"))
                  dos.writeInt(123);
                  dos.writeDouble(3.14);
                  dos.writeUTF("Hello"); // For Strings
```

Error Handling

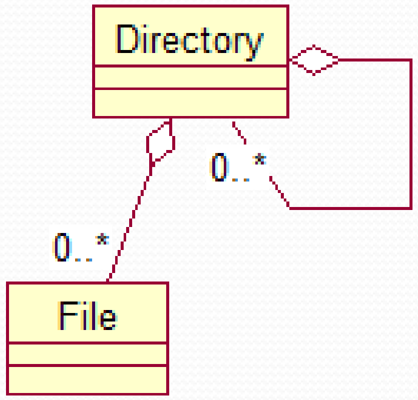
- Always use try-catch blocks to handle potential IOExceptions that can occur during file operations.
- Using try-with-resources is highly recommended for automatic resource management.
- When reading from a DataInputStream, you'll eventually reach the end of the file.
- The read...() methods will throw an EOFException (End Of File Exception) when this happens.
- Your code *must* handle this exception to gracefully terminate the reading process.

The File Class

- The `File` class is an abstraction that can be used to represents both a file and 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`

Example: Searching for a File

- Suppose we want to write a Java method that searches for a particular file. This problem is naturally solved by recursion. To see what is involved, we represent the structure of a directory in the following class diagram:



Strategy

To search for a given file *file* in a given directory *dir,* the recursive strategy is:

- Get all the files and other directories that lie in the given directory *dir*
- For each of these files, compare with the given file *file* – if the same, return true
- For each directory *d* among the directories found in *dir*, recursively search for *file*
- Return false

Pseudo-code for File Search

```
//this is not Java code
boolean searchForFile(Object file, Object startDir) {

    Object[] fileSystemObjects = startDir.getContents();

    for(Object o: fileSystemObjects) {
        //base case
        if(isFile(o) && isSameFile(o,file)) {
            return true;
        }

        if(isDirectory(o)) {
            searchForFile(file, o);
        }
    }
    //file not found in startDir
    return false;
}
```

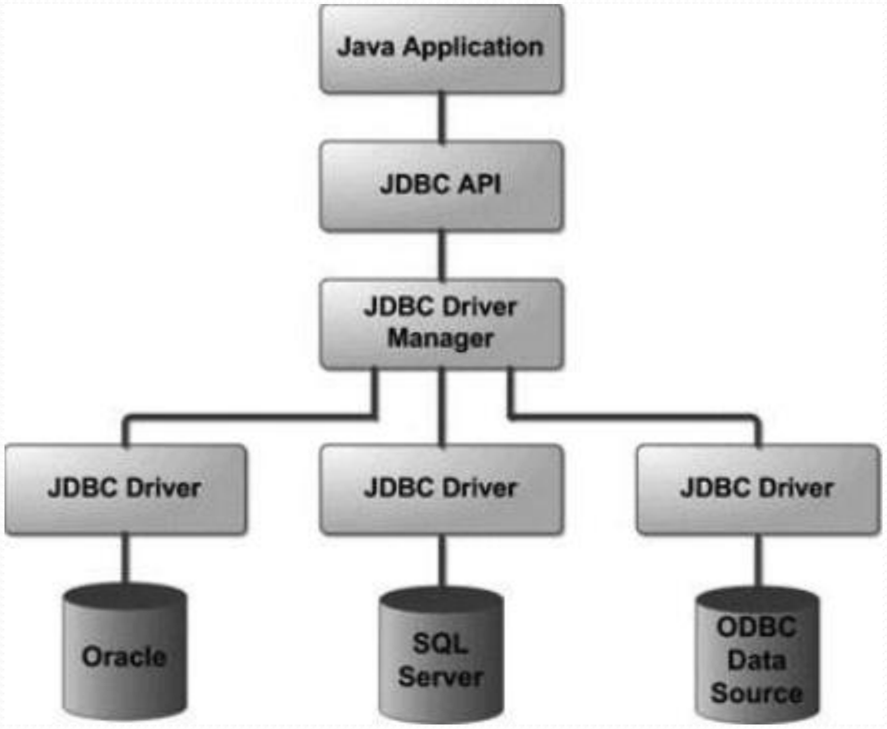
You will turn this pseudo-code into actual Java code in the lab for this lesson.

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 Using JDBC

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



1. To access a database, client code uses the JDBC API to get a connection, create and execute a statement, and get results
2. To communicate with a particular database, a vendor-specific *JDBC driver* is provided and registered with Java's `DriverManager` class
3. Commands made via the JDBC API involve interaction with the `DriverManager` . Ultimately, an SQL statement is sent to the DBMS for execution and the results are returned to the caller.

Steps for Working with a Database

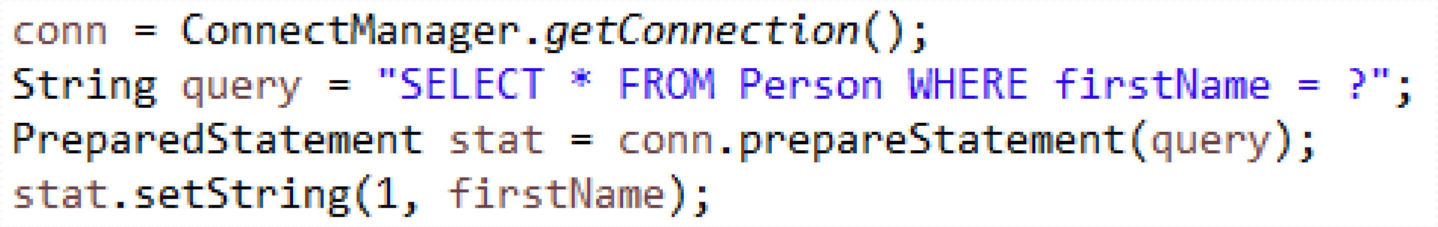
- *Set up the Database and Start the Server.* (See the folder setup/mysql folder. Work through the steps o f Lesson-13-PostgreSQL JDBC Set UP.pdf .

- *Code*
  - Use Java's `DriverManager` to get a `Connection`; this step automatically registers the driver in the `DriverManager`. You must tell JDBC the database, username, and password, along with the driver information in the form of a *db url.*
  - Use the `Connection` object to create a `PreparedStatement`, which is a wrapper for a SQL command
  - Execute the `PreparedStatement` with either `executeQuery` or `executeUpdate`
  - Reads (using `executeQuery`) will return a `ResultSet` which you use to transform the data you requested into a usable form.

Registering the Driver and Getting the Connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class ConnectManager {
   private static final String DB_URL =
                    "jdbc:postgresql://localhost:5432/FPPDB";
   private static final String USERNAME = "postgres";
   private static final String PASSWORD = "1234";
   public static Connection getConnection() throws SQLException {
      Connection conn = DriverManager.getConnection(DB_URL,
                              USERNAME,
                              PASSWORD);
      return conn;
   }
}
```

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.

PreparedStatement (Prevents SQL-Injection Attacks)

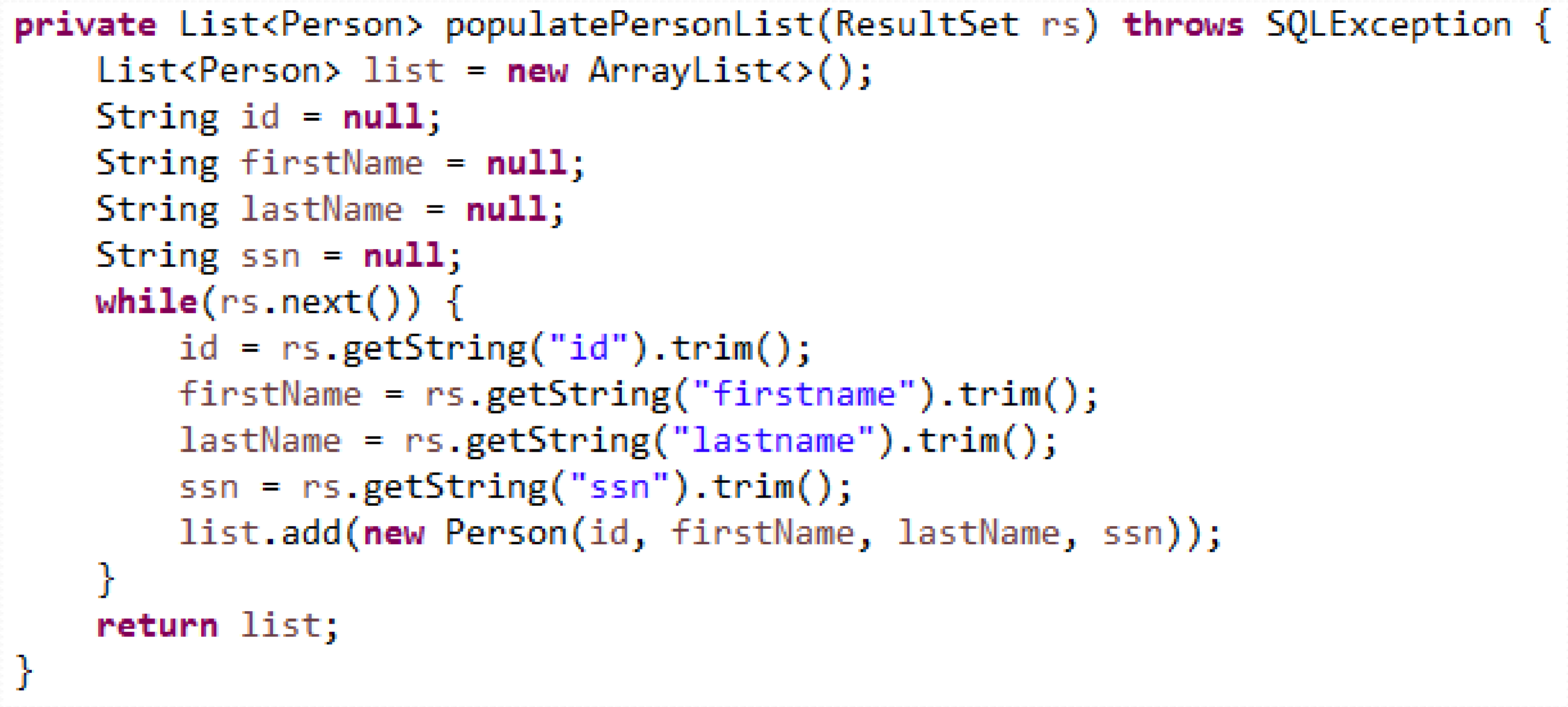
- A `PreparedStatement` is a precompiled SQL statement that allows for parameterized queries.
- By using placeholders (?) for parameters, it separates SQL logic from data input. This separation ensures that user inputs are treated as data rather than executable code, effectively preventing SQL injection attacks.
- SQL Injection occurs when an attacker injects malicious SQL code into input fields to manipulate the database.
- Java's PreparedStatement prevents SQL Injection by binding parameters instead of concatenating strings.
- See Demo: jdbcpractice package

Execute the Statement

```
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.

Process ResultSet (for reads)



```
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;
}
```

See Demos

See Java project `JdbcDemo` in your workspace.

The **Java Swing application** directly communicates with the **PostgreSQL database** using **JDBC (Java Database Connectivity)**.

Book Demo program as an example of **2-tier architecture (Client-Server Architecture)** because it consists of:

1. **Client Tier (Front-End)** → Developed using **Java Swing**, responsible for user interaction (GUI).
2. **Server Tier (Back-End)** → Uses **PostgreSQL** (or MySQL) as the database for storing and managing book records.

GUI and Database: bookdemo

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.

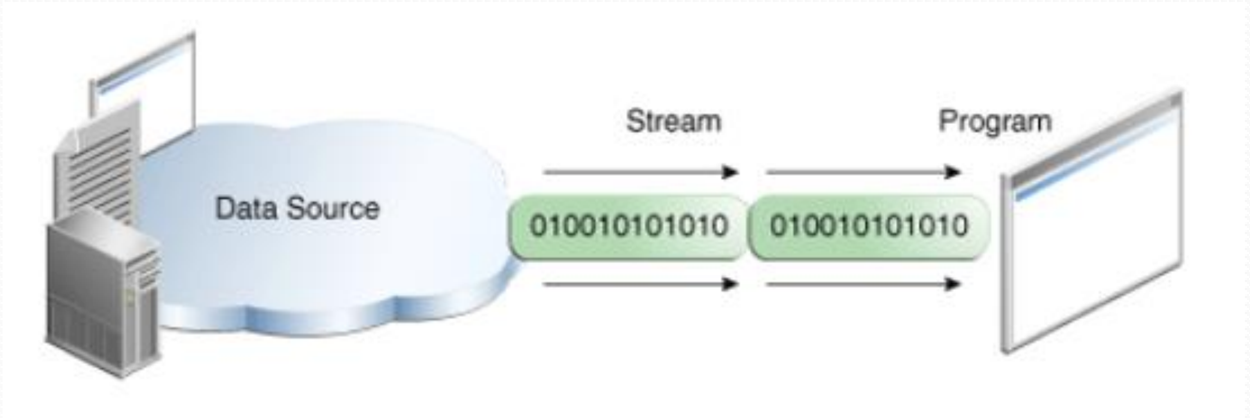
---

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.

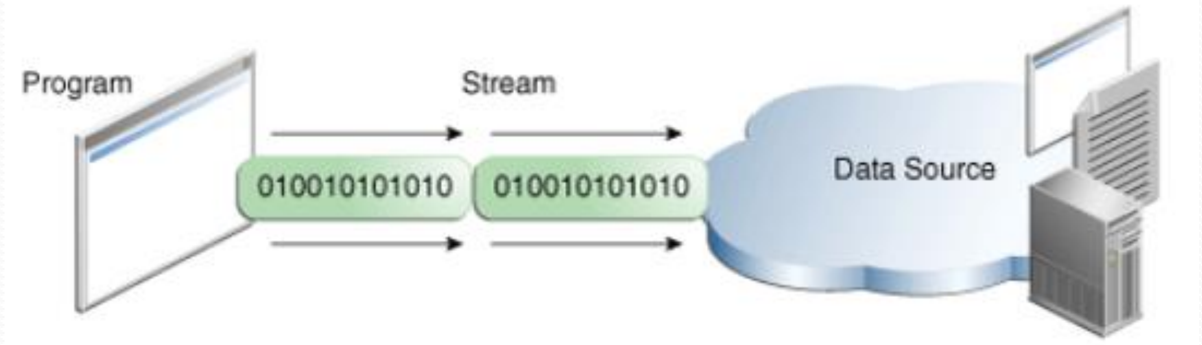
Optiona: I/O Streams in Java

- Communication between a Java program and an external device or program is often accomplished using *streams*. A stream is a sequence of bytes.
- 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.

Optional: I/O Streams in Java



Reading information into a program.



Writing information from a program.

Optional: 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`

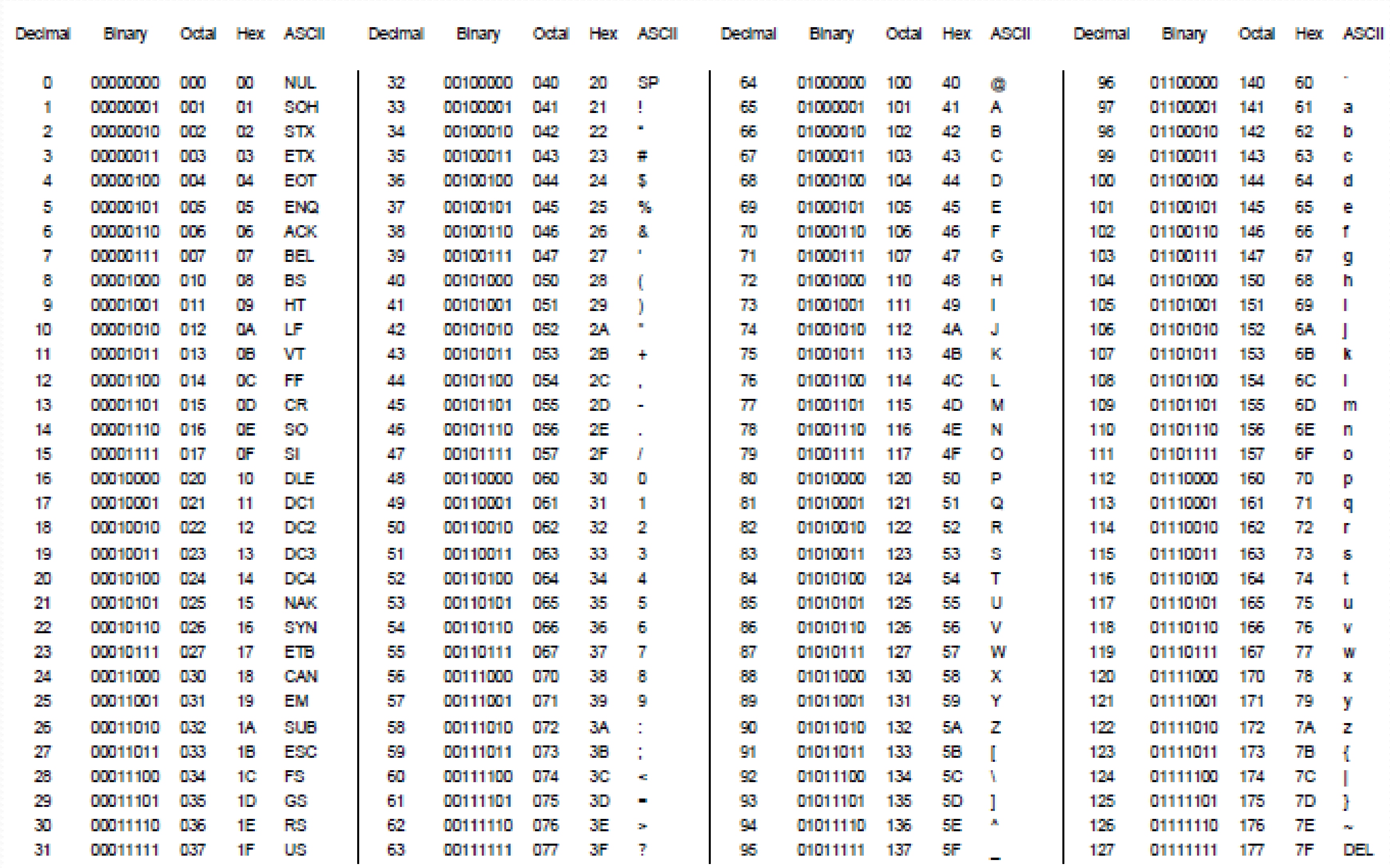
Optional: Reading/Writing Character Data

- 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)

Optional: ASCII Encoding



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

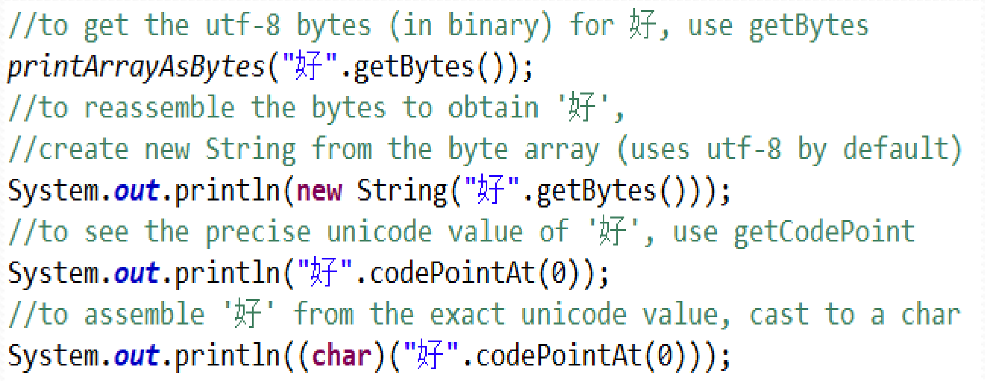
Optional: Reading Characters from a Byte Stream

- 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)));
```