

# **UNIT -6. Streams, Files:**

# Streams

- Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –
- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.
- Different Operations On Streams-  
**Intermediate Operations:**
- **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
List number = Arrays.asList(2,3,4,5);  
List square = number.stream().map(x->x\*x).collect(Collectors.toList());
- **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
- **sorted:** The sorted method is used to sort the stream.  
List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().sorted().collect(Collectors.toList());

- **Terminal Operations:**

- **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.

```
List number = Arrays.asList(2,3,4,5,3);
```

```
Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

- **forEach:** The forEach method is used to iterate through every element of the stream.

```
List number = Arrays.asList(2,3,4,5);
```

```
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

- **reduce:** The reduce method is used to reduce the elements of a stream to a single value.

The reduce method takes a BinaryOperator as a parameter.

```
List number = Arrays.asList(2,3,4,5);
```

```
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)->ans+i);
```

- Here ans variable is assigned 0 as the initial value and i is added to it

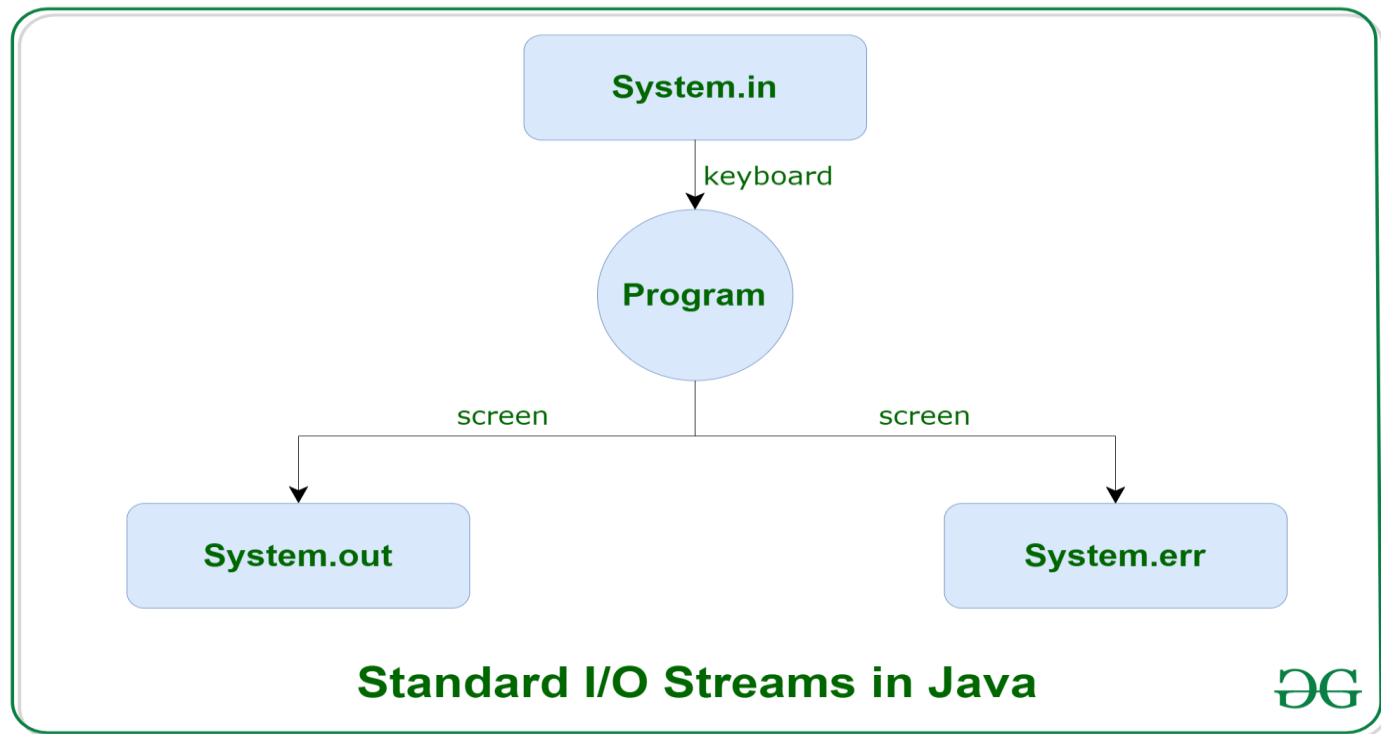
.

# Text Input and Output

- [Java](#) brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.



- Before exploring various input and output streams lets look at **3 standard or default streams** that Java has to provide which are also most common in use:



- **System.in**: This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.
- **System.out**: This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen. Here is a list of the various print functions that we use to output statements:
  - **print()**: This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.  
**Syntax:** `System.out.print(parameter);`

```
// Java code to illustrate print()
import java.io.*;

class Demo_print {
    public static void main(String[] args)
    {

        // using print()
        // all are printed in the
        // same line
        System.out.print("GfG! ");
        System.out.print("GfG! ");
        System.out.print("GfG! ");

    }
}
```

**Output:**

GfG! GfG! GfG!

- **println():** This method in Java is also used to display a text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

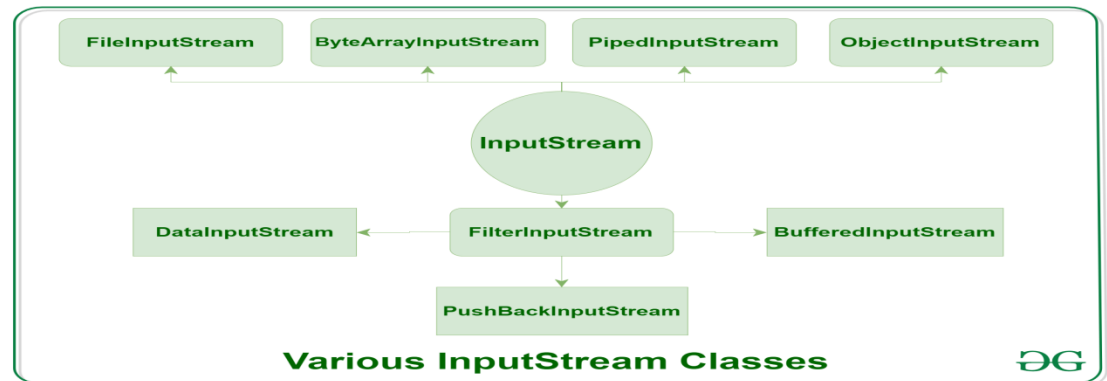
**Syntax:** `System.out.println(parameter);`

- `// Java code to illustrate println()`
- 
- `import java.io.*;`
- 
- `class Demo_print {`
- `public static void main(String[] args)`
- `{`
- 
- `// using println()`
- `// all are printed in the`
- `// different line`
- `System.out.println("GfG! ");`
- `System.out.println("GfG! ");`
- `System.out.println("GfG! ");`
- `}`
- `}`
- **Output:**
- GfG!
- GfG!
- GfG!

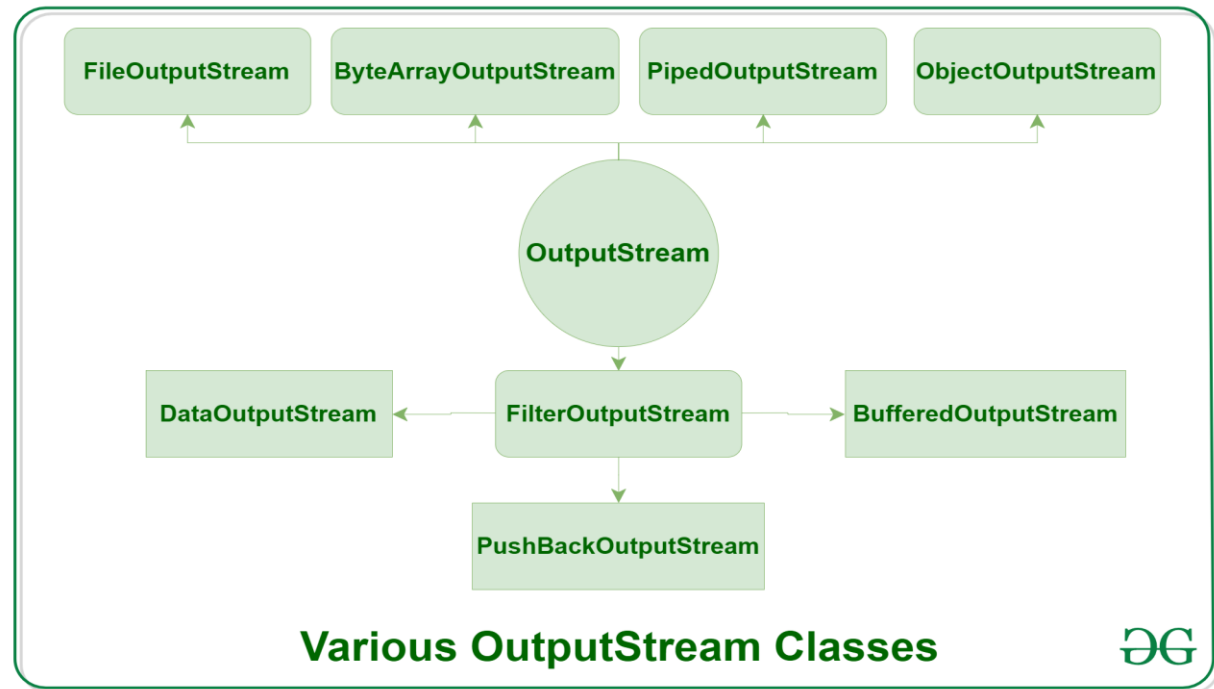


- **printf()**: This is the easiest of all methods as this is similar to printf in C. Note that System.out.print() and System.out.println() take a single argument, but printf() may take multiple arguments. This is used to format the output in Java.
- **System.err**: This is the **standard error stream** that is used to output all the error data that a program might throw, on a computer screen or any standard output device. This stream also uses all the 3 above-mentioned functions to output the error data:
  - print()
  - println()
  - printf()

- **Types of Streams:**
- **Depending on the type of operations,** streams can be divided into two primary classes:
- **Input Stream:** These streams are used to read data that must be taken as an input from a source array or file or any peripheral device. For eg., FileInputStream, BufferedInputStream, ByteArrayInputStream etc.



- **Output Stream:** These streams are used to write data as outputs into an array or file or any output peripheral device. For eg., `FileOutputStream`, `BufferedOutputStream`, `ByteArrayOutputStream` etc.



- **ByteStream:** This is used to process data byte by byte (8 bits). Though it has many classes, the `FileInputStream` and the `FileOutputStream` are the most popular ones. The `FileInputStream` is used to read from the source and `FileOutputStream` is used to write to the destination. Here is the list of various `ByteStream` Classes:

STREAM CLASS	DESCRIPTION
<a href="#"><u>BufferedInputStream</u></a>	It is used for Buffered Input Stream.
<a href="#"><u>DataInputStream</u></a>	It contains method for reading java standard datatypes.
<a href="#"><u>FileInputStream</u></a>	This is used to reads from a file
<a href="#"><u>InputStream</u></a>	This is an abstract class that describes stream input.
<a href="#"><u>PrintStream</u></a>	This contains the most used print() and println() method
<a href="#"><u>BufferedOutputStream</u></a>	This is used for Buffered Output Stream.
<a href="#"><u>DataOutputStream</u></a>	This contains method for writing java standard data types.
<a href="#"><u>FileOutputStream</u></a>	This is used to write to a file.
<a href="#"><u>OutputStream</u></a>	This is an abstract class that describe stream output.

```

// Java Program illustrating the
// Byte Stream to copy
// contents of one file to another file.
import java.io.*;
public class BStream {
    public static void main(
        String[] args) throws IOException
    {

        FileInputStream sourceStream = null;
        FileOutputStream targetStream = null;

        try {
            sourceStream
                = new FileInputStream("sourcefile.txt");
            targetStream
                = new FileOutputStream("targetfile.txt");

            // Reading source file and writing
            // content to target file byte by byte
            int temp;
            while ((
                temp = sourceStream.read())
                != -1)
                targetStream.write((byte)temp);
        }
        finally {
            if (sourceStream != null)
                sourceStream.close();
            if (targetStream != null)
                targetStream.close();
        }
    }
}

```

### Output:

Shows contents of file test.txt

- **CharacterStream:** In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character. Though it has many classes, the `FileReader` and the `FileWriter` are the most popular ones. `FileReader` and `FileWriter` are character streams used to read from the source and write to the destination respectively. Here is the list of various `CharacterStream` Classes:

STREAM CLASS	DESCRIPTION
<a href="#">BufferedReader</a>	It is used to handle buffered input stream.
<a href="#">FileReader</a>	This is an input stream that reads from file.
<a href="#">InputStreamReader</a>	This input stream is used to translate byte to character.
OutputStreamReader	This output stream is used to translate character to byte.
<a href="#">Reader</a>	This is an abstract class that define character stream input.
<a href="#">PrintWriter</a>	This contains the most used print() and println() method
<a href="#">Writer</a>	This is an abstract class that define character stream output.
<a href="#">BufferedWriter</a>	This is used to handle buffered output stream.
<a href="#">FileWriter</a>	This is used to output stream that writes to file.



```

// Java Program illustrating that
// we can read a file in a human-readable
// format using FileReader

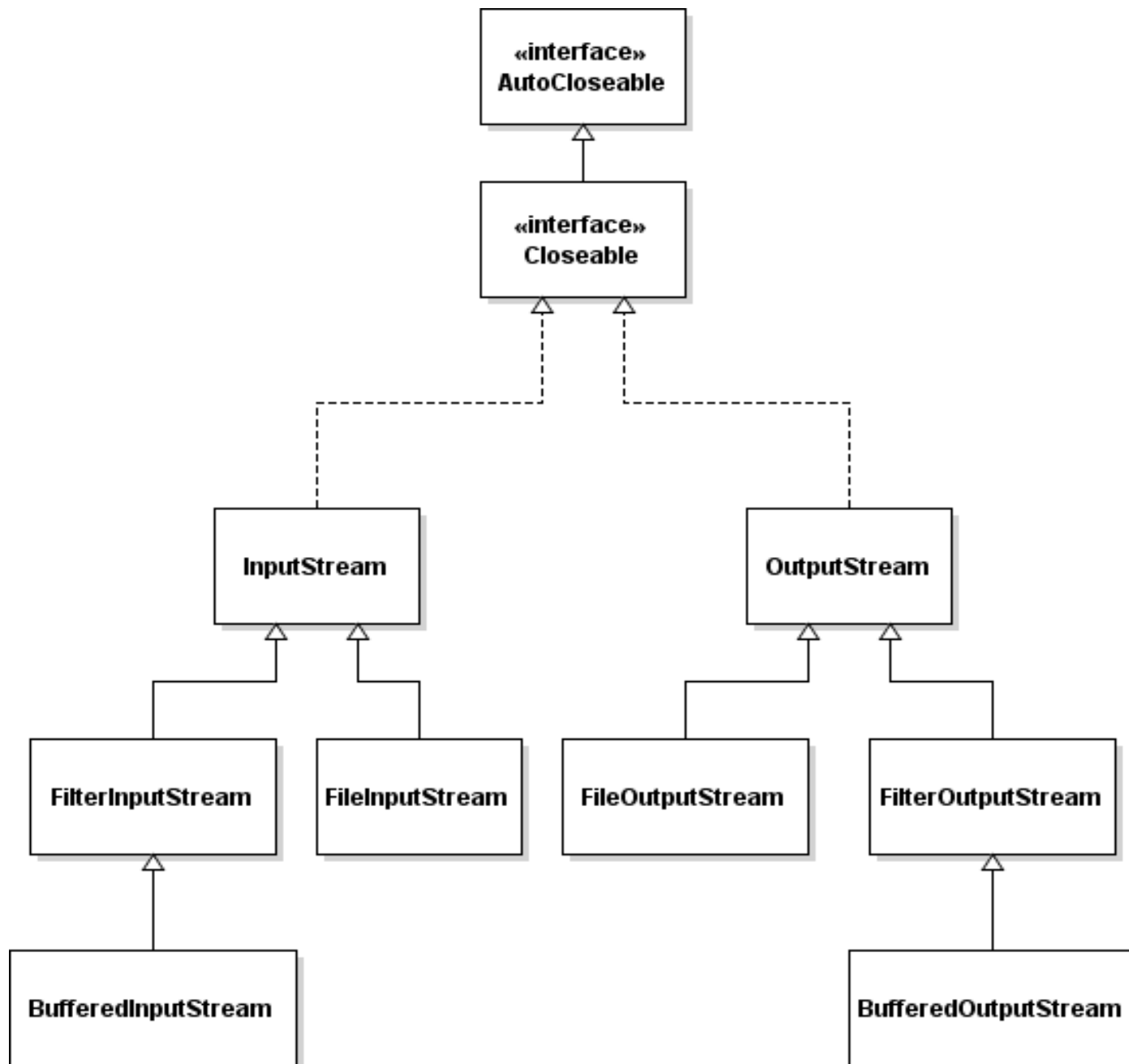
// Accessing FileReader, FileWriter,
// and IOException
import java.io.*;
public class GfG {
    public static void main(
        String[] args) throws IOException
    {
        FileReader sourceStream = null;
        try {
            sourceStream
                = new FileReader("test.txt");

            // Reading sourcefile and
            // writing content to target file
            // character by character.
            int temp;
            while ((
                temp = sourceStream.read())
                != -1)
                System.out.println((char)temp);
        }
        finally {
            // Closing stream as no longer in use
            if (sourceStream != null)
                sourceStream.close();
        }
    }
}

```

# Reading and Writing Binary Data

- **1. Understanding Byte Streams**
- We use byte streams to read and write data in binary format, exactly 8-bit bytes. All byte stream classes are descended from the abstract classes **InputStream** and **OutputStream**. The following class diagram depicts the main classes in the legacy File I/O API that are designed for working with binary files:



- You can notice that these classes implement the **AutoCloseable** interface, which means that we can use the [try-with-resources](#) structure to close these streams automatically.
- At the top of the hierarchy, the abstract class **InputStream** defines two primary methods for reading bytes from an input stream:
- **read()**: reads one byte of data, returns the byte as an integer value. Return -1 if the end of the file is reached.
- **read(byte[])**: reads a chunk of bytes to the specified byte array, up to the size of the array. This method returns -1 if there's no more data or the end of the file is reached.
- 
- Similarly, the abstract class **OutputStream** defines two primary methods for writing bytes to an output stream:
- **write(int)**: writes the specified byte to the output stream.
- **write(byte[])**: writes the specified array of bytes to the output stream.
- Moving down, the implementation classes **FileInputStream** and **FileOutputStream** are for reading and writing streams of raw bytes, one or multiple bytes at a time. Whereas the **BufferedInputStream** and **BufferedOutputStream** are more efficient by buffering the input stream and output stream to reduce the number of calls to the native API.

- **2. Reading and Writing Binary Files Using FileInputStream and FileOutputStream**
- The following examples use the **FileInputStream** and **FileOutputStream** classes to perform low level binary I/O.
- Following program runs faster because it reads the whole input file into an array of bytes and then write the whole array of bytes to the output file:

```

• import java.io.*;
•
• /**
•  * Copy one file to another using low level byte streams,
•  * read and write a whole at once.
•  * @author www.codejava.net
•  */
• public class CopyFilesOne {
•
•     public static void main(String[] args) {
•         if (args.length < 2) {
•             System.out.println("Please provide input and output files");
•             System.exit(0);
•         }
•
•         String inputFile = args[0];
•         String outputFile = args[1];
•
•
•         try (
•             InputStream inputStream = new FileInputStream(inputFile);
•             OutputStream outputStream = new FileOutputStream(outputFile);
•         ) {
•
•             long fileSize = new File(inputFile).length();
•
•             byte[] allBytes = new byte[(int) fileSize];
•
•             inputStream.read(allBytes);
•
•             outputStream.write(allBytes);
•
•         } catch (IOException ex) {
•             ex.printStackTrace();
•         }
•     }
• }

```

# File Management(File Class)

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion, etc.

The File object represents the actual file/directory on the disk. Following is the list of constructors to create a File object.

Sr.No.	Method & Description
1	<b>File(File parent, String child)</b> This constructor creates a new File instance from a parent abstract pathname and a child pathname string.
2	<b>File(String pathname)</b> This constructor creates a new File instance by converting the given pathname string into an abstract pathname.
3	<b>File(String parent, String child)</b> This constructor creates a new File instance from a parent pathname string and a child pathname string.
4	<b>File(URI uri)</b> This constructor creates a new File instance by converting the given file: URI into an abstract pathname.

Once you have *File* object in hand, then there is a list of helper methods which can be used to manipulate the files.

Sr.No.	Method & Description
1	<b>public String getName()</b> Returns the name of the file or directory denoted by this abstract pathname.
2	<b>public String getParent()</b> Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
3	<b>public File getParentFile()</b> Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
4	<b>public String getPath()</b> Converts this abstract pathname into a pathname string.
5	<b>public boolean isAbsolute()</b> Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise.
6	<b>public String getAbsolutePath()</b> Returns the absolute pathname string of this abstract pathname.



7	<b>public boolean canRead()</b>  Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
8	<b>public boolean canWrite()</b>  Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
9	<b>public boolean exists()</b>  Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise.
10	<b>public boolean isDirectory()</b>  Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
11	<b>public boolean isFile()</b>  Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise.

12	<b>public long lastModified()</b> Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.
13	<b>public long length()</b> Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
14	<b>public boolean createNewFile() throws IOException</b> Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.
15	<b>public boolean delete()</b> Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.
16	<b>public void deleteOnExit()</b> Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
17	<b>public String[] list()</b> Returns an array of strings naming the files and directories in the directory denoted by

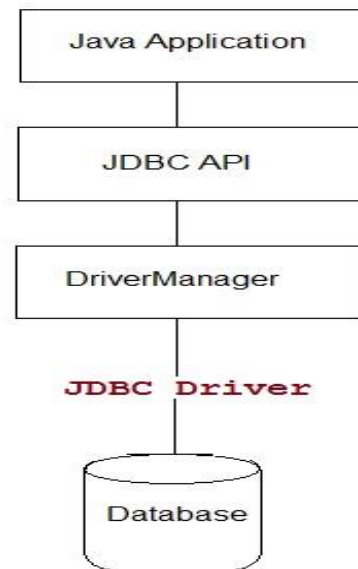
18	<p><b>public String[] list(FilenameFilter filter)</b></p> <p>Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.</p>
20	<p><b>public File[] listFiles()</b></p> <p>Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.</p>
21	<p><b>public File[] listFiles(FileFilter filter)</b></p> <p>Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.</p>
22	<p><b>public boolean mkdir()</b></p> <p>Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise.</p>
23	<p><b>public boolean mkdirs()</b></p> <p>Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.</p>
24	<p><b>public boolean renameTo(File dest)</b></p> <p>Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise.</p>

25	<p><b>public boolean setLastModified(long time)</b></p> <p>Sets the last-modified time of the file or directory named by this abstract pathname. Returns true if and only if the operation succeeded; false otherwise.</p>
26	<p><b>public boolean setReadOnly()</b></p> <p>Marks the file or directory named by this abstract pathname so that only read operations are allowed. Returns true if and only if the operation succeeded; false otherwise.</p>
27	<p><b>public static File createTempFile(String prefix, String suffix, File directory) throws IOException</b></p> <p>Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. Returns an abstract pathname denoting a newly-created empty file.</p>
28	<p><b>public static File createTempFile(String prefix, String suffix) throws IOException</b></p> <p>Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. Invoking this method is equivalent to invoking createTempFile(prefix, suffix, null). Returns abstract pathname denoting a newly-created empty file.</p>
29	<p><b>public int compareTo(File pathname)</b></p> <p>Compares two abstract pathnames lexicographically. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.</p>
30	<p><b>public int compareTo(Object o)</b></p> <p>Compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.</p>
31	<p><b>public boolean equals(Object obj)</b></p> <p>Tests this abstract pathname for equality with the given object. Returns true if and only if the argument is not null and is an abstract pathname that denotes the same file as</p>

# **UNIT -6. JDBC:**

# The Design of JDBC,

- **Java Database Connectivity(JDBC)** is an **Application Programming Interface(API)** used to connect Java application with Database. JDBC is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server. JDBC can also be defined as the platform-independent interface between a relational database and Java programming. It allows java program to execute SQL statement and retrieve result from database.



## What's new in JDBC 4.0

- **JDBC 4.0** is new and advance specification of JDBC. It provides the following advance features
- Connection Management
- Auto loading of Driver Interface.
- Better exception handling
- Support for large object
- Annotation in SQL query.

## JDBC Driver

- JDBC Driver is required to process SQL requests and generate result. The following are the different types of driver available in JDBC.
- **Type-1 Driver** or **JDBC-ODBC bridge**
- **Type-2 Driver** or **Native API Partly Java Driver**
- **Type-3 Driver** or **Network Protocol Driver**
- **Type-4 Driver** or **Thin Driver**

- **java.sql package**
- This package include classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.
- Important classes and interface of java.sql package

classes/interface	Description
java.sql.BLOB	Provide support for BLOB(Binary Large Object) SQL type.
java.sql.Connection	creates a connection with specific database
java.sql.CallableStatement	Execute stored procedures
java.sql.CLOB	Provide support for CLOB(Character Large Object) SQL type.
java.sql.Date	Provide support for Date SQL type.
java.sql.Driver	create an instance of a driver with the DriverManager.
java.sql.DriverManager	This class manages database drivers.
java.sql.PreparedStatement	Used to create and execute parameterized query.
java.sql.ResultSet	It is an interface that provide methods to access the result row-by-row.
java.sql.Savepoint	Specify savepoint in transaction.
java.sql.SQLException	Encapsulate all JDBC related exception.
java.sql.Statement	This interface is used to execute SQL statements.



- `javax.sql` package
- This package is also known as JDBC extension API. It provides classes and interface to access server-side data.
- Important classes and interface of `javax.sql` package

classes/interface	Description
<code>javax.sql.ConnectionEvent</code>	Provide information about occurrence of event.
<code>javax.sql.ConnectionEventListener</code>	Used to register event generated by <b>PooledConnection</b> object.
<code>javax.sql.DataSource</code>	Represent the <b>DataSource</b> interface used in an application.
<code>javax.sql.PooledConnection</code>	provide object to manage connection pools.

# Executing SQL Statements

- Once a connection is obtained we can interact with the database. The *JDBC Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.
- They also define methods that help bridge data type differences between Java and SQL data types used in a database.
- The following table provides a summary of each interface's purpose to decide on the interface to use.

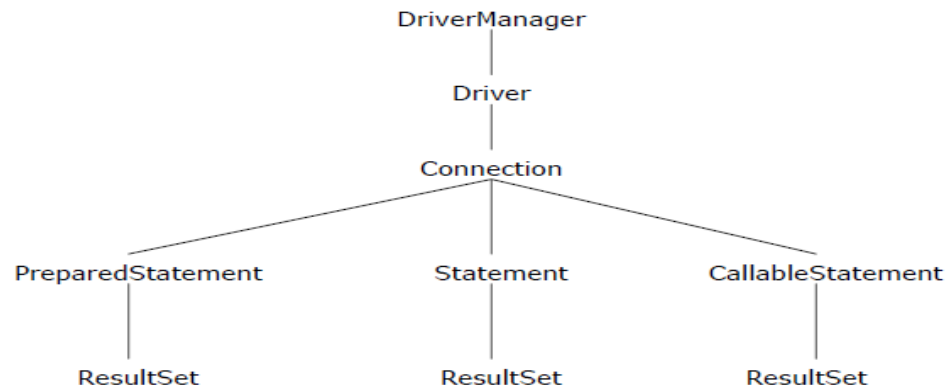
Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

- **The Statement Objects**
- **Creating Statement Object**
- Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's `createStatement( )` method, as in the following example –
- `Statement stmt = null;`
- `try`
- `{`
- `stmt = conn.createStatement( );`
- `...`
- `}`
- `catch (SQLException e)`
- `{ ... }`
- `Finally`
- `{ ... }`

- ***The JDBC API***

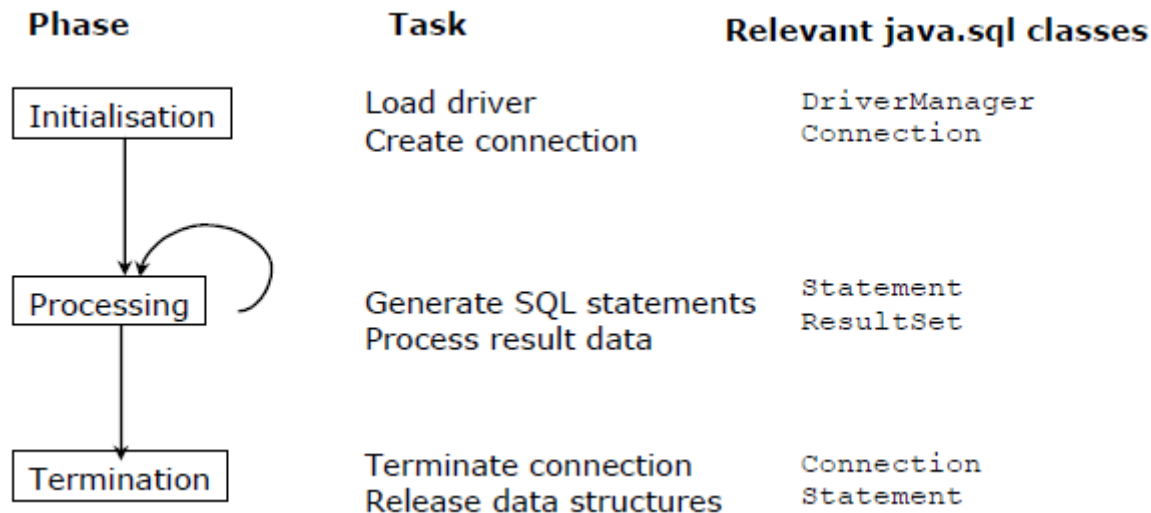
- The JDBC API is contained in two packages named java.sql and javax.sql. The java.sql package contains core Java objects of JDBC API. There are two distinct layers within the JDBC API: the application layer, which database application developers use and driver layer which the drivers vendors implement. The connection between application and driver layers is illustrated in figure below:

- ***The JDBC API***



- There are four main interfaces that every driver layer must implement and one class that bridges the Application and driver layers. The four interfaces are Driver, Connection, Statement and ResultSet.
- The Driver interface implementation is where the connection to the database is made.
- In most applications, Driver is accessed through DriverManager class.

- ***The JDBC process***
- ***Accessing JDBC / ODBC Bridge with the database***
- Before actual performing the Java database application, we associate the connection of database source using JDBC – ODBC Bridge. The steps are as follows:
  1. Go to Control Panel -> Administrative Tools -> Data Sources.
  2. Open Data Sources ODBC icon.
  3. Select the tab with heading “User DSN”.
  4. Click on ‘Add’ button.
  5. Select the appropriate driver as per the database to be used. (e.g. Microsoft ODBC driver for Oracle to access Oracle Database
  6. Click finish button and the corresponding ODBC database setup window will appear.
  7. Type DSN name and provide the required information such as user name and password for the database (.mdb files) of Microsoft Access Database etc. and click on OK button.
  8. Our DSN name will get appeared in user data sources.
- There are six different steps to use JDBC in our Java application program. These can be shown diagrammatically as below:



1. Load the driver
2. Define and establish the Connection
3. Create a Statement object
4. Execute a query
5. Process the results
6. Close the connection

### ***Loading the JDBC driver***

The JDBC drivers must be loaded before the Java application connects to the DBMS. The

### **Class.forName()**

is used to load the JDBC driver. The developer must write routine that loads the JDBC / ODBC Bridge. The bridge driver called sun.jdbc.odbc.JdbcOdbc Driver. It is done in following way:

**Class.forName("sun.jdbc.odbc.JdbcOdbc Driver");**

- **Connect to the DBMS** After loading the driver the application must get connected to DBMS. For this we use DriverManager.getConnection() method. The Driver Manager is highest class in Java.sql hierarchy and is responsible for managing driver related information.
- The DriverManager.getConnection() method is passed the URL of the database and user ID and password required by the database. The URL is the string object that contains the driver name that is being accessed by the Java program.
- The DriverManager.getConnection() method returns Connection interface that is used throughout the process to reference the database. The signature of this method is:
- Connection DriverManager.getConnection(String url, String userID, String password);

Here, the URL format is specified as follows:

<protocol>:<subprotocol>:<dsn-name>

- The 'protocol' is a JDBC protocol that is used to read the URL. The 'subprotocol' is JDBC driver name and 'dsn-name' is the name of the database that we provided while creating JDBC Bridge through control panel. We use the following URL for our application:
- jdbc:odbc:customer



- here, 'customer' is an example of DSN name given to our database. The user name and password are also provided at the time of creating DSN. It is not compulsory to provide the username and password. For example:
- Connction con;
- con = DriverManager.getConnection("jdbc:odbc:customer","micro",  
"pitch");
- **Create Statement object**
- The createStatement( ) method of Connection interface is used to create the Statement object which is then used to execute the query. For example:

Statement st = con.createStatement();

- **Execute the query**
- The executeQuery() method of Statement object is used execute and process the query which returns the ResultSet object. ResultSet is the object which actually contains the result returned by the query. For example:

**ResultSet rs = st.executeQuery("select \* from customer");**

Here, the 'customer' is neither database name nor DSN name but it is a table name.

- **Process the results**
- The ResultSet object is assigned the results received from the DBMS after the query is processed. The ResultSet object consists of methods used to interact with data that is returned by the DBMS to Java application program. For example, the next() method is used to proceed throughout the result set. It returns true, if the data is available in result set to read.
- The ResultSet also contains several getXxx( ) methods to read the value from particular column of current row. For example, getString("name") will read the value from column 'name' in the form of string. Instead of passing column name as parameter, we can pass column as parameter also. Such as,
  - getString(1). For example:
  - String name;
  - int age;
  - do
  - {
  - name = rs.getString("name");
  - age = rs.getInt("age");
  - System.out.println(name+"="+age);
  - } while(rs.next());

### ***Terminate the Connection***

- The Connection to the DBMS is terminated by using the close() method of the Connection object once Java program has finished accessing the DBMS. The close( ) method throws an exception if a problem is encountered when disengaging the DBMS. For example:
- **con.close();**
- The close() method of Statement object is used to close the statement object to stop the further processing.

- **Statement Objects**
- Once the connection to the database is opened, the Java application creates and sends a query to access data contained in the database. One of three type of statement objects is used to execute the query immediately. A Prepared Statement is used to execute the compiled query and Callable Statement is used to execute the stored procedure.
- **Statement object**
- The Statement object is used whenever a Java program needs to immediately execute a query without first having query compiled. The Statement contains three different methods depending upon the type of query these will be used

**1. *executeQuery()***

This method returns the ResultSet object that contains rows, columns and metadata that represent data requested by the query. Its signature is:

**ResultSet executeQuery(String query);**

Generally, this method is used to execute only the 'SELECT' query of the SQL.

**2. *executeUpdate()***

This method is used to execute the queries that contain INSERT, DELETE and UPDATE statements. This method returns integer indicating the number of rows that were updated by the query. Its signature is:

**int executeUpdate(String query);**

**For Example**

```
int rows = st.executeUpdate("DELETE FROM EMPLOYEES WHERE STATUS=0");
```

### 3. ***execute()***

It executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts we must then use the method `getResultSet()` or `getUpdateCount()` to retrieve the result, and `getMoreResults()` to move to any subsequent result(s).

Signature is as follows:

```
public boolean execute(String sql)
```

For example:

```
if(st.execute())
```

```
rs = st.getResultSet();
```

Signatures of other methods:

```
public ResultSet getResultSet() public int getUpdateCount()
```

```
public boolean getMoreResults()
```

- ***PreparedStatement object***
- A SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object's execution method is called. Compiling a query is an overhead that is acceptable if the query is called once. However, compiling process can become an expensive overhead if the query is executed several times by the same program during the same session.
- A SQL query can be precompiled and executed by using the Prepared Statement object. In such cases a query is created similar to other queries. However, a question mark is given on the place for the value that is inserted into the query after it is compiled. It is the value that changes each time the query is executed.
- For doing this process, we need to construct the query with question marks such as,
- “select \* from nation where population > ?”

- Such type of the query is passed as the parameter to the `prepareStatement( )` method of the `Connection` object which then returns the `PreparedStatement` object. For example:
- `String query = "select * from nation where population > ?";`
- `PreparedStatement ps = preparedStatement(query);`
- Once the `PreparedStatement` object is obtained, the `setXxx( )` methods of it can be used to replace question mark with the value passed to `setXxx()` method. There are a number of `setXxx()` methods available in `PreparedStatement` object, each of which specifies the data type of value that is being passed to `setXxx()` method. For example, considering the above query again,
- `ps.setInt(1, 100000);`
- This method requires two parameters. First parameter is an integer that identifies position of the question mark placeholder and second is the value that replaces the question mark. If the query contains two question marks we have to pass second value also using `setXxx()` method.
- Now, we need to use appropriate execute method depending upon type of the query without any parameters. Such as,  
`ResultSet rs = ps.executeQuery();`

- This will generate the ResultSet object as the execution of the query. The Prepared Statement contain all three execute methods but without any parameters as given below:
  1. `ResultSet executeQuery( )`
  2. `int executeUpdate( )`
  3. `boolean execute( )`
  4. The `setXxx( )` methods:
    5. `void setBoolean(int index, boolean value);`
    6. `void setByte(int index, byte value);`
    7. `void setDate(int index, Date value);`
    8. `void setDouble(int index, double value);`
    9. `void setFloat(int index, float value);`
    10. `void setInt(int index, int value);`
    11. `void setLong(int index, long value);`
    12. `void setObject(int index, Object value);`
    13. `void setShort(int index, short value);`
    14. `void setString(int index, String value);`

- ***Example:***
- Consider the following database:

Consider the following database:

	Roll	Name	Marks
	1	Rakhee	75
	2	Amit	49
	3	Ajita	63
	4	Rahul	78
	5	Minal	67
	6	Karthik	71



```

import java.sql.*;
class StudentData
{
public static void main(String args[])
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con =
DriverManager.getConnection("jdbc:odbc:stud");
PreparedStatement ps = con.prepareStatement("select * from Student where Marks >
?");
ps.setInt(1,70); //set question marks place holder
ResultSet rs = ps.executeQuery(); //execute
System.out.println("Students having marks > 70 are:");
while(rs.next())
System.out.println(rs.getString(2));
con.close();
}
catch(Exception e){ }
}
}

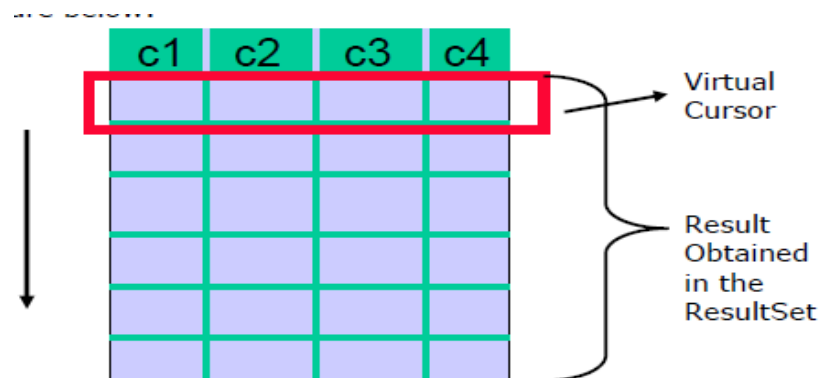
```

### ***Output***

Students having marks > 70 are:  
Rakhee  
Rahul  
Karthik

# ***ResultSet***

- The ResultSet class provides methods to access data generated by a table query. This includes a series of get methods which retrieve data in any one of the JDBC SQL type formats, either by column number or by column name.
- The ResultSet object contains methods that are used to copy data from the ResultSet into Java collection object or variable for further processing.
- Data in Result Set object is logically organized into virtual table consisting of rows and columns. In addition to the data, the ResultSet object also contains metadata such as column names, column size, and column data type.
- The Result Set uses a virtual cursor to point to the row of virtual table as shown in figure below:




# ***ResultSet operations***

- A JDBC application program must move the virtual cursor to each row and then move the virtual cursor to each row and then use the methods of the ResultSet object to interact with the data stored in the column of that row.
- The virtual cursor is positioned above the first row of data when the ResultSet is returned by the execute Query() method. This means that the virtual cursor must be moved to the first row using next() method.
- The next() method returns a boolean value if the row contains data otherwise false is returned indicating that no more rows exists in the ResultSet.
- Once a virtual cursor points to a row the getXxx() method is used to copy the data from the row to a collection, object or a variable. The selection of particular get method depends upon the type of value that column contains. Following get methods are used in ResultSet:

1. boolean getBoolean(int columnIndex)
2. boolean getBoolean(String columnName)
3. byte getByte(int columnIndex)
4. byte getByte(String columnName)
5. Date getDate(int columnIndex)
6. Date getDate(String columnName)
7. double getDouble(int columnIndex)
8. double getDouble(String columnName)
9. float getFloat(int columnIndex)
10. float getFloat(String columnName)
11. int getInt(int columnIndex)
12. int getInt(String columnName)
13. long getLong(int columnIndex)
14. long getLong(String columnName)
15. double getDouble(int columnIndex)
16. double getDouble(String columnName)
17. Object getObject(int columnIndex)
18. Object getObject(String columnName)
19. short getShort(int columnIndex)
20. short getShort(String columnName)
21. String getString(int columnIndex)
22. String getString(String columnName)
23. Time getTime(int columnIndex)
24. Time getTime(String columnName)

Now, consider the following database,

	Roll	Name	Marks	Passed	Birthdate
	1	Rakhee	75	<input checked="" type="checkbox"/>	12/14/1990
	2	Amit	38	<input type="checkbox"/>	10/2/1990
	3	Ajita	63	<input checked="" type="checkbox"/>	1/24/1989
	4	Rahul	78	<input checked="" type="checkbox"/>	1/1/1990
	5	Minal	67	<input checked="" type="checkbox"/>	2/12/1991
	6	Karthik	71	<input checked="" type="checkbox"/>	7/6/1988

The program given below reads the database and displays its contents in a tabular format:

```

import java.sql.*;
class StudentData
{
public static void main(String args[])
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con =
DriverManager.getConnection("jdbc:odbc:stud");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select * from Student");
System.out.println("The Database is:-");
System.out.println("RollNameMarks Pass Birth-Date");
System.out.println("=====");
while(rs.next())
{
int roll = rs.getInt(1);
String name = rs.getString(2);
int marks = rs.getInt("Marks");
boolean pass = rs.getBoolean(4);
Date d = rs.getDate(5);
System.out.printf("%-5d",roll);
System.out.printf("%-10s",name);
System.out.printf("%-6d",marks);
if(pass)
System.out.printf("Yes ");
else
System.out.printf("No ");
System.out.printf("%-15sn",d.toString());
}
con.close();
}
catch(Exception e){ }
}
}

```

## ***Output***

The Database is:

Roll Name Marks Pass Birth-Date

```

===== 1
Rakhee 75 Yes 1990-12-14 2 Amit 38 No 1990-10-02 3
Ajita 63 Yes 1989-01-24 4 Rahul 78 Yes 1990-01-01 5
Minal 67 Yes 1991-02-12 6 Karthik 71 Yes 1988-07-06

```

- The addition to the next() method, the ResultSet can also be moved backward or any other position inside it. For doing these scrolling, we can use the following methods:

1. `public boolean first()`
2. `public boolean last()`
3. `public boolean next()`
4. `public boolean previous()`
5. `public boolean absolute(int position)`
6. `public boolean relative(int rows)`

- The first() method moves the virtual cursor to the first row of the ResultSet. Likewise, last() method moves the virtual cursor to the last row of the ResultSet.

- The previous() method moves the virtual cursor to the previous row of the Result Set from current position.

- The absolute() method positions virtual cursor at the row number specified by integer passed as parameter to this method.

- The relative() method moves the virtual cursor the specified number of rows virtual cursor.

- If this parameter is positive, virtual cursor moves forward by that number of rows. Similarly, negative number moves the cursor backward direction by that number of rows.

- The Statement object that is created using create Statement() of Connection object must be set up to handle a scrollable ResultSet by passing create Statement method one of three constants given below:
  1. TYPE\_FORWARD\_ONLY
  2. TYPE\_SCROLL\_INSENSITIVE
  3. TYPE\_SCROLL\_SENSITIVE
- The TYPE\_FORWARD\_ONLY constant restricts the virtual cursor to downward movement, which is a default setting.
- The TYPE\_SCROLL\_INSENSITIVE and TYPE\_SCROLL\_SENSITIVE constants permit virtual cursor to move in both directions.
- The TYPE\_SCROLL\_INSENSITIVE constant makes the ResultSet insensitive to changes made by another JDBC application to data in the table whose rows are reflected in the ResultSet.
- The TYPE\_SCROLL\_SENSITIVE constant makes the ResultSet sensitive to those changes.



# ***Updatable ResultSet***

- The rows contained in the ResultSet can be updatable similar to how rows in the current table can be updated. This is made possible by passing the create Statement() method of the Connection object the CONCUR \_UPDATABLE. Alternatively, the CONCUR \_READ \_ONLY constant can be passed to the create Statement() method to prevent the ResultSet from being updated.
- There are three ways in which the ResultSet can be changed. These are updating values in a row and inserting a new row. All these changes are accomplished by using methods of Statement object.
- ***Updating ResultSet***
- Once the execute Query() method of the Statement object returns the ResultSet, the updateXxx() method is used to change the value of column in the current row of the ResultSet. The xxx is replaced by any data type of the column that is to be updated.
- The updateXxx() method requires two parameters. The first is either the number or the name of the ResultSet that is being updated and the second parameter is the value in the column of the ResultSet. Such as,
  - void updateInt(int columnIndex, int x)
  - void updateInt(String columnName, int x)
  - void updateLong(int columnIndex, long x)
  - void updateLong(String columnName, long x)
  - and so on....

- For example:
- `s.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,`
- `ResultSet.CONCUR_UPDATABLE);`
- `r.updateString("Name", "Alisha");`
- `r.updateRow();`
- In above code, the current row's column 'Name's value will be replaced by 'Alisha'.
- ***Deleting a Row***
- The `deleteRow()` method is used to remove a row from `ResultSet`. This method directly deletes the contents on the current row and the virtual cursor will be moved to the next row automatically. For example:

```
rs.deleteRow();
```

- ***Inserting a Row***
- Inserting a row in the ResultSet is accomplished using basically the same technique as is used to update the ResultSet.
- That is, the updateXxx() method is used to specify the column and the value that will be placed into the column of the ResultSet.
- The updateXxx() requires two parameters. The first is either name of the column or number of the column of the ResultSet. The second parameter is new value that will be placed in the column of the ResultSet.
- The insertRow() method is called after the updateXxx() methods, which causes a new row to be inserted into the ResultSet having values that reflect the parameters in the updateXxx() methods.
- This also updates underlying database.
- **For example:**
- s.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,
- ResultSet.CONCUR\_UPDATABLE );
- r.updateString("Name", "Aneeta");
- r.updateInt("Roll", 12);
- r.insertRow();

# ***Metadata***

- Metadata is nothing but data about data. Database name, table name, column name, column attributes, and other information that describes database components are known as Metadata. For Example, the size of client's first name column describes the data contained within the column and therefore is referred to as metadata.
- Metadata is used by JDBC application to identify database components without needing to know details of a column, the table or the database. For example, a J2EE component can request from the DBMS the data type of a specific column. The column type is used by a JDBC application to copy data retrieved from the DBMS into a Java collection. JDBC provides two meta -data interfaces:

`java.sql.ResultSetMetaData`

`java.sql.DatabaseMetaData.`

- The meta -data described by these classes was included in the original JDBC Result Set and Connection classes. The extra functionality could be served by creating meta -data classes to provide the often esoteric information required by a minority of developers.
- The JDBC application program retrieves the object of Database MetaData by using the `get MetaData` of the Connection object. Such as,
- `Database MetaData dm = con.getMetaData();`

- Then we can use the following methods of Database MetaData interface to retrieve the metadata from the database.
- ***getDatabaseProductName()*** It returns product name of database
- ***getDatabaseProductVersion()*** It returns product version of database
- ***getUserName()*** It returns the user name
- ***getURL()*** It returns URL of database
- ***getSchemas()*** It returns schema names available in database
- ***getPrimaryKeys()*** It returns primary keys
- ***getProcedures()*** It returns stored procedure names
- ***getTables()*** It returns names of tables in database
- Once the Result Set MetaData is retrieved, the JDBC application can call methods of ResultSet MetaData object to retrieve specific kind of metadata. The more commonly called methods are as follows:
- ***getColumnCount()*** It returns no. of columns contained in ResultSet.
- ***getColumnName(int number)*** It returns the name of column specified by column number.
- ***getColumnType(int no)*** It returns the data type of column specified by column number.

- For example:
- `ResultSetMetaData md = rs.getMetaData();`
- `// get number of columns`
- `int nCols = md.getColumnCount();`
- `// print column names`
- `for(int i=1; i < nCols; ++i)`
- `System.out.print(md.getColumnName(i)+",");`
- `// output resultset`
- `while (rs.next())`
- `{ for(int i=1; i < nCols; ++i)`
- `System.out.print(rs.getString(i)+",");`
- `System.out.println(rs.getString(nCols));`
- `}`

- ***SQLException***
- It commonly reflects the SQL syntax error in the query and is thrown many of the methods contained in java.sql package. This exception is most commonly caused by connectivity issues with the database. It can also be caused by subtle coding errors like trying to access an object that has been closed. Following SQL Exception class specific methods are listed below:
  - ***getMessage()*** It gives the description of the error.
  - ***getSQLState()*** It returns SQLState (Open Group SQL specification) identifying the exception.
  - ***getErrorCode()*** It returns a vendor-specific integer error code.
  - ***getNextException()*** If more than one error occurs, they are chained together. This method is used to chain to the next exception.

- For example:

```
try
{
... // JDBC statement
} catch (SQLException sqle) {
while (sqle != null) {
System.out.println("Message: " + sqle.getMessage());
System.out.println("SQLState: " + sqle.getSQLState());
System.out.println("Vendor Error: " +
sqle.getErrorCode());
sqle.printStackTrace(System.out);
sqle = sqle.getNextException();
}
}
```



# Transactions

## Transaction Management in JDBC

Transaction represents a single unit of work.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all successful or none.

Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Advantage of Transaction Management- fast performance It makes the performance fast because database is hit at the time of commit.

In JDBC, Connection interface provides methods to manage transaction.

Method	Description
<code>void setAutoCommit(boolean status)</code>	It is true bydefault means each transaction is committed bydefault.
<code>void commit()</code>	commits the transaction.
<code>void rollback()</code>	cancels the transaction.

- Simple example of transaction management in jdbc using Statement
- Let's see the simple example of transaction management using Statement.
- **import** java.sql.\*;
- **class** FetchRecords{
- **public static void** main(String args[])**throws** Exception{
- Class.forName("oracle.jdbc.driver.OracleDriver");
- Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "oracle");
- con.setAutoCommit(**false**);
- Statement stmt=con.createStatement();
- stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
- stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
- con.commit();
- con.close();
- }}

# Unit-6 Java packages

## Java Packages; JAR Archives

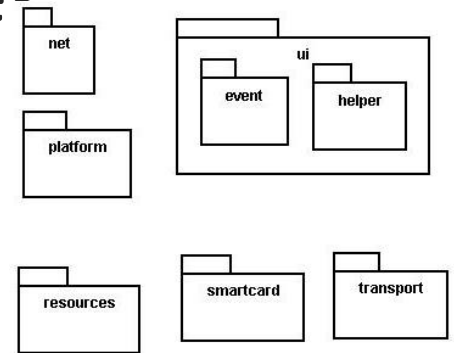
slides created by Marty Stepp

based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia

<http://www.cs.washington.edu/331/>

# Java packages

- **package:** A collection of related classes
  - Can also "contain" sub-packages.
  - *Sub-packages* can have similar names, but are not actually contained inside.
    - `java.awt` does not contain `java.awt.event`

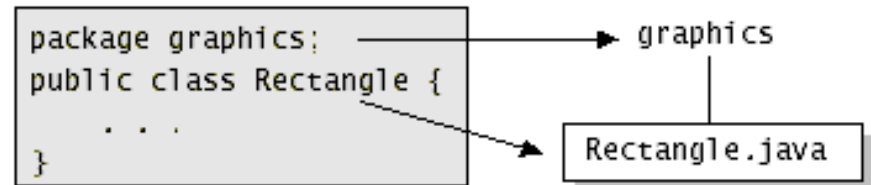


- Uses of Java packages:
  - group related classes together
  - as a *namespace* to avoid name collisions
  - provide a layer of access / protection
  - keep pieces of a project down to a manageable size

# Packages and directories

- package  $\longleftrightarrow$  directory (folder)
- class  $\longleftrightarrow$  file
- A class named `D` in package `a.b.c` should reside in this file:

`a/b/c/D.class`



– (relative to the root of your project)

- The "root" directory of the package hierarchy is determined by your *class path* or the directory from which `java` was run.

# Classpath

- **class path:** The location(s) in which Java looks for class files.
- Can include:
  - the current "working directory" from which you ran `javac / java`
  - other folders
  - JAR archives
  - URLs
  - ...
- Can set class path manually when running java at command line:
  - `java -cp /home/stepp/libs:/foo/bar/jbl MyClass`

# A package declaration

```
package name;  
public class name { ...
```

## Example:

```
package pacman.model;  
public class Ghost extends Sprite {  
    ...  
}
```

- **File** `Sprite.java` should go in folder `pacman/model`.

# Importing a package

```
import packageName.*;           // all classes
```

Example:

```
package pacman.gui;  
import pacman.model.*;  
  
public class PacManGui {  
    ...  
    Ghost blinky = new Ghost();  
}
```

- PacManGui must import the model package in order to use it.



# Importing a class

```
import packageName.className;    // one  
class
```

Example:

```
package pacman.gui;  
import pacman.model.Sprite;  
  
public class PacManGui {  
    Ghost blinky = new Ghost();  
}
```

- Importing single classes has high precedence:
  - if you import `. *`, a same-named class in the current dir will override
  - if you import `. className`, it will not

# Static import

```
import static packageName.className.*;
```

Example:

```
import static java.lang.Math.*;  
  
...  
double angle = sin(PI / 2) + ln(E * E);
```

- Static import allows you to refer to the members of another class without writing that class's name.
- Should be used rarely and only with classes whose contents are entirely static "utility" code.

# Referring to packages

**packageName . className**

## Example:

```
java.util.Scanner console =  
    new java.util.Scanner(java.lang.System.in);
```

- You can use a type from any package without importing it if you write its full name.
- Sometimes this is useful to disambiguate similar names.
  - Example: `java.awt.List` and `java.util.List`
  - Or, explicitly import one of the classes.

# The default package

- Compilation units (files) that do not declare a package are put into a default, unnamed, package.
- Classes in the default package:
  - Cannot be imported
  - Cannot be used by classes in other packages
- Many editors discourage the use of the default package.
- Package `java.lang` is implicitly imported in all programs by default.
  - `import java.lang.*;`

# Package access

- Java provides the following access modifiers:
  - `public` : Visible to all other classes.
  - `private` : Visible only to the current class (and any nested types).
  - `protected` : Visible to the current class, any of its subclasses, and any other types within the same package.
  - default (package): Visible to the current class and any other types within the same package.
- To give a member default scope, do not write a modifier:

```
package pacman.model;
public class Sprite {
    int points;           // visible to pacman.model.*
    String name;          // visible to pacman.model.*
```

# Package exercise

- Add packages to the Rock-Paper-Scissors game.
  - Create a package for core "model" data.
  - Create a package for graphical "view" classes.
  - Any general utility code can go into a default package or into another named utility (util) package.
  - Add appropriate package and import statements so that the types can use each other properly.