

# JDBC



Dr. Jason Barron

South East Technological University

November 17, 2023

- A database is an organised collection of data
- A database management system (DBMS) provides mechanisms for storing, organising, retrieving and modifying data for many users
- SQL is the international standard language used with relational databases to perform queries and to manipulate data
- Popular relational database management systems (RDBMSs)
  - Microsoft SQL Server
  - Oracle
  - Sybase
  - IBM DB2
  - Informix
  - PostgreSQL
  - MySQL

- Java programs communicate with databases and manipulate their data using the Java Database Connectivity (JDBC) API
- A JDBC driver enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API

- A relational database is a logical representation of data that allows the data to be accessed without consideration of its physical structure
- A relational database stores data in tables
- Tables are composed of rows, and rows are composed of columns in which values are stored
- **Primary key** - a column (or group of columns) with a unique value that cannot be duplicated in other rows

# Example Database Table

	Number	Name	Department	Salary	Location
	23603	Jones	413	1100	New Jersey
	24568	Kerwin	413	2000	New Jersey
Row {	34589	Larson	642	1800	Los Angeles
	35761	Myers	611	1400	Orlando
	47132	Neumann	413	9000	New Jersey
	78321	Stephens	611	8500	Orlando
	Primary key		Column		

# Table Relationships



- In order to work with a relational database, a standard database language known as SQL (Standard Query Language) is used
- Once you learn SQL, you will be able to write queries for any relational database
- SQL is powerful yet not very difficult to learn

SQL Keyword	Description
SELECT	Retrieve data
FROM	Which tables involved
WHERE	For Criteria
GROUP BY	For grouping rows
ORDER BY	For sorting results
INNER JOIN	Select from multiple tables
INSERT	Insert new row
UPDATE	Update row(s)
DELETE	Delete row(s)



- A SQL query "selects" rows and columns from one or more tables in a database
- The basic form of a **SELECT** query is

---

```
SELECT * FROM tableName
```

---

- The asterisk (\*) wildcard character indicates that all columns from the tableName table should be retrieved
- To retrieve all the data in the Authors table, use

---

```
SELECT * FROM Authors
```

---

- To retrieve only specific columns, replace the asterisk (\*) with a comma-separated list of the column names, e.g.,

---

```
SELECT AuthorID, LastName FROM Authors
```

- In most cases, only rows that satisfy selection criteria are selected
- SQL uses the optional **WHERE** clause in a query to specify the selection criteria for the query
- The basic form of a query with selection criteria is

---

```
SELECT columnName1, columnName2, ... FROM tableName WHERE criteria
```

---

- To select the **Title**, **EditionNumber** and **Copyright** columns from table **Titles** for which the **Copyright** date is greater than 2010, use the query

---

```
SELECT Title, EditionNumber, Copyright FROM Titles WHERE Copyright > '2010'
```

---

- Strings in SQL are delimited by single (') rather than double (") quotes
- The **WHERE** clause criteria can contain the operators <, >, <=, >=, =, <> and **LIKE**

- Operator **LIKE** is used for pattern matching with wildcard characters percent (%) and underscore (\_)
- A pattern that contains a percent character (%) searches for strings that have zero or more characters at the percent character's position in the pattern
- For example, the next query locates the rows of all the authors whose last name starts with the letter D:

---

```
SELECT AuthorID, FirstName, LastName FROM Authors WHERE  
      LastName LIKE 'D%'
```

---

- An underscore (\_) in the **LIKE** pattern string indicates a single wildcard character at that position in the pattern
- The following query locates the rows of all the authors whose last names start with any character (specified by \_), followed by the letter o, followed by any number of additional characters (specified by %):

---

```
SELECT AuthorID, FirstName, LastName FROM Authors WHERE  
    LastName LIKE '_o%'
```

---

- The rows in the result of a query can be sorted into ascending or descending order by using the optional **ORDER BY** clause
- The basic form of a query with an **ORDER BY** clause is

---

```
SELECT columnName1, columnName2, ... FROM tableName ORDER  
      BY column ASC
```

```
SELECT columnName1, columnName2, ... FROM tableName ORDER  
      BY column DESC
```

---

- ASC specifies ascending order (lowest to highest)
- DESC specifies descending order (highest to lowest)
- Column specifies the column on which the sort is based

- To obtain the list of authors in ascending order by last name (), use the query

---

```
SELECT AuthorID, FirstName, LastName FROM Authors ORDER BY LastName ASC
```

---

- To obtain the same list of authors in descending order by last name (), use the query

---

```
SELECT AuthorID, FirstName, LastName FROM Authors ORDER BY LastName DESC
```

---

- Multiple columns can be used for sorting with an **ORDER BY** clause of the form
  - sortOrder is either ASC or DESC

---

```
ORDER BY column1 sortOrder, column2 sortOrder, ...
```

---

- Sort all the rows in ascending order by last name, then by first name

---

```
SELECT AuthorID, FirstName, LastName FROM Authors ORDER BY LastName, FirstName
```

---

- The **WHERE** and **ORDER BY** clauses can be combined in one query, as in

---

```
SELECT ISBN, Title, EditionNumber, Copyright FROM Titles  
WHERE Title LIKE '%How to Program' ORDER BY Title ASC
```

---

- This returns the **ISBN**, **Title**, **EditionNumber** and **Copyright** of each book in the **Titles** table that has a **Title** ending with "How to Program" and sorts them in ascending order by **Title**

- Database designers often split related data into separate tables to ensure that a database does not store data redundantly
- Often, it is necessary to merge data from multiple tables into a single result
  - Referred to as joining the tables
- An **INNER JOIN** merges rows from two tables by matching values in columns that are common to the tables

---

```
SELECT columnName1, columnName2, ... FROM table1 INNER  
JOIN table2 ON table1.columnName = table2.columnName
```

---

- The **ON** clause specifies the columns from each table that are compared to determine which rows are merged



- The following query produces a list of authors accompanied by the ISBNs for books written by each author:

---

```
SELECT FirstName, LastName, ISBN FROM Authors INNER JOIN  
  AuthorISBN ON Authors.AuthorID = AuthorISBN.AuthorID  
ORDER BY LastName, FirstName
```

---

- The syntax **tableName.columnName** in the **ON** clause, called a qualified name, specifies the columns from each table that should be compared to join the tables

- The **INSERT** statement inserts a row into a table

---

```
INSERT INTO tableName ( columnName1, columnName2, ...,  
                        columnNameN ) VALUES ( value1, value2, ..., valueN )
```

---

- Where **tableName** is the table in which to insert the row
  - **tableName** is followed by a comma-separated list of column names in parentheses
  - Not required if the **INSERT** operation specifies a value for every column of the table in the correct order
- The list of column names is followed by the SQL keyword **VALUES** and a comma-separated list of values in parentheses
  - The values specified here must match the columns specified after the table name in both order and type

- The **INSERT** statement

---

```
INSERT INTO Authors ( FirstName, LastName ) VALUES (  
    'Sue', Red' )
```

---

- Indicates that values are provided for the **FirstName** and **LastName** columns
- The corresponding values are 'Sue' and Red'
- We do not specify an **AuthorID** in this example because **AuthorID** is an autoincremented column in the **Authors** table
  - Not every database management system supports autoincremented columns

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class InsertAuthor {

    public static void main(String [] args) {

        // database URL
        final String DATABASE_URL = "jdbc:mysql://localhost/books";
        Connection connection = null ;
        PreparedStatement pstat = null;
        String firstname = "Mark";
        String lastname = "Power";
        int i=0;

        try {
```

---

```
// establish connection to database
connection = DriverManager.getConnection(DATABASE_URL, "root", "password" );

// create Prepared Statement for inserting data into table
pstat = connection.prepareStatement("INSERT INTO Authors (FirstName, LastName)
VALUES (?,?)");
pstat.setString(1, firstname);
pstat.setString(2, lastname);

// insert data into table
i = pstat .executeUpdate();
System.out.println (i + " record successfully added to the table.");
}
```

---

---

```
catch(SQLException sqlException){
    sqlException.printStackTrace();
}
finally {
    try {
        pstat.close();
        connection.close();
    }
    catch (Exception exception){
        exception.printStackTrace();
    }
}
} // end main
} // end class
```

---

- An **UPDATE** statement modifies data in a table

---

```
UPDATE tableName SET columnName1 = value1, columnName2 = value2, ..., columnNameN = valueN WHERE criteria
```

---

- Where **tableName** is the table to update
  - **tableName** is followed by keyword **SET** and a comma-separated list of column name/value pairs in the format **columnName = value**
  - Optional **WHERE** clause provides criteria that determine which rows to update
- The **UPDATE** statement

---

```
UPDATE Authors SET LastName = 'Black' WHERE LastName = 'Red' AND FirstName = 'Sue'
```

---

- Indicates that **LastName** will be assigned the value Black for the row in which **LastName** is equal to Red and **FirstName** is equal to Sue

---

```
//Update an Author in the Authors table .
```

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.SQLException;
```

```
public class UpdateAuthor {
```

```
    public static void main(String[] args) {
```

```
        // database URL
```

```
        final String DATABASE_URL = "jdbc:mysql://localhost/books";
```

```
        String firstname="Lisa";
```

```
        String lastname="Brennan";
```

```
        Connection connection = null;
```

```
        PreparedStatement pstat = null;
```

```
        int i=0;
```

```
        try{
```



---

```
// establish connection to database
connection = DriverManager.getConnection(DATABASE_URL, "root", "password" );

// create Prepared Statement for updating data in the table
pstat = connection.prepareStatement("Update Authors SET LastName=? Where
    FirstName=?");
pstat.setString (1, lastname);
pstat.setString (2, firstname);

//Update data in the table
i = pstat.executeUpdate();
System.out. println (i + " record successfully updated in the dtable.");

}
```

---

---

```
catch(SQLException sqlException) {  
    sqlException.printStackTrace();  
}  
finally {  
    try {  
        pstat.close();  
        connection.close();  
    }  
    catch (Exception exception) {  
        exception.printStackTrace();  
    }  
}  
} // end main  
} // end class
```

---

- A **DELETE** statement removes rows from a table

---

**DELETE FROM** tableName **WHERE** criteria

---

- Where **tableName** is the table from which to delete
  - Optional **WHERE** clause specifies the criteria used to determine which rows to delete
  - If this clause is omitted, all the table's rows are deleted
- The **DELETE** statement

---

**DELETE FROM** Authors **WHERE** LastName = 'Black' **AND** FirstName = 'Sue'

---

- Deletes the row for Sue Black in the **Authors** table

# Delete Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class DeleteAuthor {

    public static void main(String[] args) {

        // database URL
        final String DATABASE_URL = "jdbc:mysql://localhost/books";

        Connection connection = null;
        PreparedStatement pstat = null;
        int i=0;
        int authorID=6;

        try{
```

---

```
// establish connection to database
connection = DriverManager.getConnection(
    DATABASE_URL, "root", "password" );

// create Prepared Statement for deleting data from the table
pstat = connection.prepareStatement("Delete From Authors Where
    AuthorID=");
pstat.setInt (1, authorID);

// delete data from the table
i = pstat.executeUpdate();
System.out.println (i + " record successfully removed from the table.");
}
```

---

---

```
catch(SQLException sqlException) {  
    sqlException.printStackTrace();  
}  
finally {  
    try {  
        pstat.close();  
        connection.close();  
    }  
    catch (Exception exception) {  
        exception.printStackTrace();  
    }  
}  
} // end main  
} // end class
```

---

---

```
//Displaying the contents of the Authors table.  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.ResultSetMetaData;  
import java.sql.SQLException;  
  
public class DisplayAuthors{  
  
    public static void main( String args[] ){  
  
        // database URL  
        final String DATABASE_URL = "jdbc:mysql://localhost/books";  
  
        Connection connection = null;  
        PreparedStatement pstat= null;  
        ResultSet resultSet = null;  
        try{
```

---

---

```
// establish connection to database
connection = DriverManager.getConnection(DATABASE_URL, "root", "password" );

// create Prepared Statement for querying data in the table
pstat = connection.prepareStatement("SELECT AuthorID, FirstName, LastName FROM
    Authors");

// query data in the table
resultSet = pstat.executeQuery();

// process query results
ResultSetMetaData metaData = resultSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();
System.out.println ( "Authors Table of Books Database:\n" );

for ( int i = 1; i <= numberOfColumns; i++ )
    System.out.print (metaData.getColumnName( i ) + "\t");
System.out.println ( );
```

---



```
while( resultSet .next() ){
    for ( int i = 1; i <= numberOfColumns; i++ )
        System.out.print( resultSet .getObject( i ) + "\t\t");
    System.out.println ();
}
}

catch(SQLException sqlException ) {
    sqlException .printStackTrace();
}
}

finally {
    try{
        resultSet .close ();
        pstat .close ();
        connection .close ();
    }
    catch (Exception exception){
        exception .printStackTrace();
    }
}
}

} // end main
} // end class
```

Authors Table of Books Database:

authorID	firstName	lastName
1	John	Power

- The database URL identifies the name of the database to connect to, as well as information about the protocol used by the JDBC driver
- JDBC 4.0 and higher support automatic driver discovery
  - No longer required to load the database driver in advance
  - To ensure that the program can locate the database driver class, you must include the class's location in the program's classpath when you execute the program

- An object that implements interface **Connection** manages the connection between the Java program and the database
- **Connection** objects enable programs to create SQL statements that manipulate databases
- **DriverManager**'s static method **getConnection** attempts to connect to the database specified by its URL
- Three arguments
  - A **String** that specifies the database URL
  - A **String** that specifies the username
  - A **String** that specifies the password

- The URL **jdbc:mysql://localhost/books** specifies
  - the protocol for communication (**jdbc**)
  - the subprotocol for communication (**mysql**)
  - the location of the database (**//localhost/books**, where **localhost** is the host running the MySQL server and **books** is the database name)
- The subprotocol **mysql** indicates that the program uses a MySQL - specific subprotocol to connect to the MySQL database

- **Connection**'s method **createStatement** obtains an object that implements interface **Statement** (package **java.sql**)
  - Used to submit SQL statements to the database
- The **Statement** object's **executeQuery** method submits a query to the database
  - Returns an object that implements interface **ResultSet** and contains the query results
  - The **ResultSet** method enables the program to manipulate the query result
- A **ResultSet**'s **ResultSetMetaData** describes the **ResultSet**'s contents
  - Can be used programmatically to obtain information about the **ResultSet**'s column names and types
- **ResultSetMetaData**'s method **getColumnCount** retrieves the number of columns in the **ResultSet**

- The first call to **ResultSet**'s method **next** positions the **ResultSet** cursor to the first row
  - Returns **boolean** value **true** if it is able to position to the next row; otherwise, the method returns **false**
- **ResultSetMetaData**'s method **getColumnType** returns a constant integer from class **Types** (package **java.sql**) indicating the type of a specified column
- **ResultSet**'s method **getInt** returns a column value as an **int**
- **ResultSet**'s get methods typically receive as an argument either a column number (as an **int**) or a column name (as a **String**) indicating which column's value to obtain
- **ResultSet**'s method **getObject** returns a column value as an **Object**

- The next example allows the user to enter any query into the program
- Displays the result of a query in a **JTable**, using a **TableModel** object to provide the **ResultSet** data to the **JTable**
- **JTable** is a swing GUI component that can be bound to a database to display the results of a query
- Class **ResultSetTableModel** performs the connection to the database via a **TableModel** and maintains the **ResultSet**
- Class **DisplayQueryResults** creates the GUI and specifies an instance of class **ResultSetTableModel** to provide data for the **JTable**



- **ResultSetTableModel** overrides **TableModel**'s methods **getColumnClass**, **getColumnCount**, **getColumnName**, **getRowCount** and **getValueAt** (inherited from **AbstractTableModel**)

---

//A TableModel that supplies ResultSet data to a JTable.

```
import java . sql . Connection ;
import java . sql . Statement ;
import java . sql . DriverManager ;
import java . sql . ResultSet ;
import java . sql . ResultSetMetaData ;
import java . sql . SQLException ;
import javax . swing . table . AbstractTableModel ;

public class ResultSetTableModel extends AbstractTableModel {

    private Connection connection ;
    private Statement statement ;
    private ResultSet resultSet ;
    private ResultSetMetaData metaData ;
    private int numberOfRows ;
    private boolean connectedToDatabase = false ;
```

---

```
public ResultSetTableModel( String url, String username, String password, String
    query ) throws SQLException{
    connection = DriverManager.getConnection( url, username, password );
    statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY );
    connectedToDatabase = true;
    setQuery( query );
}
public Class getColumnClass( int column ) throws IllegalStateException {
    if ( !connectedToDatabase ) throw new IllegalStateException( "Not Connected to
        Database" );
    try{
        String className = metaData.getColumnClassName( column + 1 );
        // return Class object that represents className
        return Class.forName( className );
    }
    catch ( Exception exception ) {
        exception.printStackTrace();
    }
    return Object.class; // if problems occur above, assume type Object
}
```

# Querying the books Database

```
public int getColumnCount() throws IllegalStateException {
    if ( !connectedToDatabase ) throw new IllegalStateException( "Not Connected to
        Database" );
    try {
        return metaData.getColumnCount();
    }
    catch ( SQLException sqlException ){
        sqlException.printStackTrace();
    }
    return 0; // if problems occur above, return 0 for number of columns
}

public String getColumnName( int column ) throws IllegalStateException {
    if ( !connectedToDatabase ) throw new IllegalStateException( "Not Connected to
        Database" );
    try {
        return metaData.getColumnName( column + 1 );
    }
    catch ( SQLException sqlException ){
        sqlException.printStackTrace();
    }
    return ""; // if problems, return empty string for column name
}
```

```
public int getRowCount() throws IllegalStateException {
    if ( !connectedToDatabase ) throw new IllegalStateException( "Not Connected to
        Database" );
    return numberOfRows;
}

public Object getValueAt( int row, int column )throws IllegalStateException {
    if ( !connectedToDatabase ) throw new IllegalStateException( "Not Connected to
        Database" );
    try {
        resultSet.absolute( row + 1 );
        return resultSet.getObject( column + 1 );
    }
    catch ( SQLException sqlException ){
        sqlException.printStackTrace();
    }
    return ""; // if problems, return empty string object
}
```

# Querying the books Database

```
public void setQuery( String query ) throws SQLException, IllegalStateException {
    if ( !connectedToDatabase ) throw new IllegalStateException( "Not Connected to Database" );
    resultSet = statement.executeQuery( query );
    metaData = resultSet.getMetaData();
    resultSet . last ();
    numberOfRows = resultSet.getRow();
    fireTableStructureChanged ();
}

public void disconnectFromDatabase(){
    if ( connectedToDatabase ) {
        try{
            resultSet . close ();
            statement . close ();
            connection . close ();
        }
        catch ( SQLException sqlException ) {
            sqlException . printStackTrace ();
        }
        finally {
            connectedToDatabase = false;
        }
    }
}
} //end class
```

- **Connection**'s method **createStatement** with two arguments receives the result set type and the result set concurrency
- The **ResultSet.TYPE** specifies whether the **ResultSet**'s cursor is able to scroll in both directions or forward only and whether the **ResultSet** is sensitive to changes made to the underlying data
  - **ResultSet**s that are sensitive to changes reflect those changes immediately after they are made with methods of interface **ResultSet**
  - If a **ResultSet** is insensitive to changes, the query that produced the **ResultSet** must be executed again to reflect any changes made
- The **ResultSet.CONCUR** specifies whether the **ResultSet** can be updated with **ResultSet**'s update methods

- **ResultSetMetaData**'s method **getColumnClassName** obtains the fully qualified class name for the specified column
- **ResultSetMetaData**'s method **getColumnCount** obtains the number of columns in the **ResultSet**
- **ResultSetMetaData**'s method **getColumnName** obtains the column name from the **ResultSet**
- **ResultSet**'s method **absolute** positions the **ResultSet** cursor at a specific row
- **ResultSet**'s method **last** positions the **ResultSet** cursor at the last row in the **ResultSet**
- **ResultSet**'s method **getRow** obtains the row number for the current row in the **ResultSet**
- Method **fireTableStructureChanged** (inherited from class **AbstractTableModel**) notifies any **JTable** using this **ResultSetTableModel** object as its model that the structure of the model has changed
  - Causes the **JTable** to repopulate its rows and columns with the new **ResultSet** data



---

//Display the contents of the Authors table in the books database.

```
import java .awt.BorderLayout;  
import java .awt.event . ActionListener ;  
import java .awt.event . ActionEvent;  
import java .awt.event . WindowAdapter;  
import java .awt.event . WindowEvent;  
import java . sql . SQLException;  
import java . util . regex . PatternSyntaxException;  
import javax . swing . JFrame;  
import javax . swing . JTextArea;  
import javax . swing . JScrollPane ;  
import javax . swing . ScrollPaneConstants;  
import javax . swing . JTable;  
import javax . swing . JOptionPane;  
import javax . swing . JButton;  
import javax . swing . Box;  
import javax . swing . JLabel;  
import javax . swing . JTextField ;  
import javax . swing . RowFilter;  
import javax . swing . table . TableRowSorter;  
import javax . swing . table .TableModel;
```

```
public class DisplayQueryResults extends JFrame {

    static final String DATABASE_URL = "jdbc:mysql://localhost/books";
    static final String USERNAME = "root";
    static final String PASSWORD = "password";
    static final String DEFAULT_QUERY = "SELECT * FROM Authors";
    private ResultSetTableModel tableModel;
    private JTextArea queryArea;

    public DisplayQueryResults(){
        super( "Displaying Query Results" );
        try{
            tableModel = new ResultSetTableModel( DATABASE_URL,
            USERNAME, PASSWORD, DEFAULT_QUERY );
            queryArea = new JTextArea( DEFAULT_QUERY, 3, 100 );
            queryArea.setWrapStyleWord( true );
            queryArea.setLineWrap( true );
            JScrollPane scrollPane = new JScrollPane( queryArea,
            JScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
            JScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
            JButton submitButton = new JButton( "Submit Query" );
```

---

```
Box boxNorth = Box.createHorizontalBox();
boxNorth.add( scrollPane );
boxNorth.add( submitButton );
JTable resultTable = new JTable( tableModel );
JLabel filterLabel = new JLabel( " Filter:" );
final JTextField filterText = new JTextField();
JButton filterButton = new JButton( "Apply Filter" );
Box boxSouth = Box.createHorizontalBox();
boxSouth.add( filterLabel );
boxSouth.add( filterText );
boxSouth.add( filterButton );
add( boxNorth, BorderLayout.NORTH );
add( new JScrollPane( resultTable ), BorderLayout.CENTER );
add( boxSouth, BorderLayout.SOUTH );
```

---

```
submitButton.addActionListener(new ActionListener() {  
    public void actionPerformed( ActionEvent event ) {  
        try {  
            tableModel.setQuery( queryArea.getText() );  
        }  
        catch ( SQLException sqlException ) {  
            JOptionPane.showMessageDialog( null, sqlException.getMessage(),  
                "Database error", JOptionPane.ERROR_MESSAGE );  
            try {  
                tableModel.setQuery( DEFAULT_QUERY );  
                queryArea.setText( DEFAULT_QUERY );  
            }  
            catch ( SQLException sqlException2 ) {  
                JOptionPane.showMessageDialog( null, sqlException2.getMessage(),  
                    "Database error", JOptionPane.ERROR_MESSAGE );  
                tableModel.disconnectFromDatabase();  
                System.exit( 1 );  
            }  
        }  
    }  
} //end Submit Button Action Listener class  
};
```

```
final TableRowSorter< TableModel > sorter =new TableRowSorter< TableModel
    >( tableModel );
resultTable .setRowSorter( sorter );
setSize( 500, 250 );
setVisible ( true );
filterButton .addActionListener(new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
        String text = filterText .getText();
        if ( text .length() == 0 )
            sorter .setRowFilter( null );
        else {
            try {
                sorter .setRowFilter( RowFilter. regexFilter ( text ) );
            }
            catch ( PatternSyntaxException pse ) {
                JOptionPane.showMessageDialog( null, "Bad regex pattern", "Bad regex
                    pattern", JOptionPane.ERROR_MESSAGE );
            }
        }
    }
} //end Filter Button Action Listener class
);;
```

```
catch ( SQLException sqlException ){
    JOptionPane.showMessageDialog( null, sqlException.getMessage(), "Database
        error", JOptionPane.ERROR_MESSAGE );
    tableModel.disconnectFromDatabase();
    System.exit( 1 );
}
// dispose of window when user quits application (this overrides the
// default of HIDE_ON_CLOSE)
setDefaultCloseOperation( DISPOSE_ON_CLOSE );
addWindowListener(new WindowAdapter(){
    public void windowClosed( WindowEvent event ){
        tableModel.disconnectFromDatabase();
        System.exit( 0 );
    }
}
); //end Window Listener class
}

public static void main( String args[] ) {
    new DisplayQueryResults();
} // end main
} // end class
```

# Querying the books Database

Displaying Query Results

```
SELECT * FROM Authors
```

Submit Query

authorID	firstName	lastName
1	John	Power

Filter:

Apply Filter

- Any local variable that will be used in an anonymous inner class must be declared final; otherwise, a compilation error occurs
- Class **TableRowSorter** (from package **javax.swing.table**) can be used to sort rows in a **JTable**
  - When the user clicks the title of a particular **JTable** column, the **TableRowSorter** interacts with the underlying **TableModel** to reorder the rows based on the data in that column
- **JTable**'s method **setRowSorter** specifies the **TableRowSorter** for the **JTable**



- **JTable**s can now show subsets of the data from the underlying **TableModel**
  - This is known as filtering the data
- **JTable**'s method **setRowFilter** specifies a **RowFilter** (from package **javax.swing**) for a **JTable**
- **RowFilter**'s static method **regexFilter** receives a **String** containing a regular expression pattern as its argument and an optional set of indices that specify which columns to filter
  - If no indices are specified, then all the columns are searched