Java Programming

Exception Handling & Introduction to Data Access with JDBC

Review of Lecture 7

- **JavaFX Basics:**
  - The main class for a JavaFX application extends the javafx.application.**Application** class
  - **Stage class is the top-level JavaFX container.**
  - **Scene class is the container for all content**
  - Layout panes:
    - Pane
    - GridPane
    - FlowPane
    - HBox
    - VBox
  - pane.getChildren().**addAll**(btOK, btCancel);

- Shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.
  - **Text, Rectangle, Circle, etc.**

- **Event Handling in JavaFX**

```
Button btOK = new Button("OK");
OKHandlerClass handler1 = new
OKHandlerClass();
btOK.setOnAction(handler1);
..................
class OKHandlerClass implements
EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
System.out.println("OK button clicked");
 }
}
```

Review of Lecture 7

- **Lambda Expressions**:
  - can be viewed as an anonymous method with a concise syntax.
  - btLeft.**setOnAction**(e -> text.setX(text.getX() - 10));
- **JavaFX GUI classes**
  - **Button**, **Label**, **CheckBox**, **RadioButton**, **TextField**, **TextArea**, **ComboBox**, **ListView**, **ScrollBar**

```
TextArea taDescription = new TextArea();
ScrollPane scrollPane = new
ScrollPane(taDescription);
setCenter(scrollPane);
```

- **Setting styles:**

```
lngBox.setStyle("-fx-padding: 10;" +
 "-fx-border-style: solid inside;" +
            "-fx-border-width: 2;" +
            "-fx-border-insets: 5;" +
            "-fx-border-radius: 5;" +
            "-fx-border-color: blue;");
```

Lesson 9 Objectives

- Understand **Exception Handling** mechanism in Java.
  - Use try and catch blocks to detect and handle exceptions.
  - Us throw statement to indicate a problem.
  - Use finally block to release the resources.
- Understand **JDBC API** to access databases.
- Create Java applications that establish a connection to a database and retrieve data from its tables.
  - **Connection**, **Statement** and **ResultSet** interfaces.

What is an exception in Java?

- Exception – an indication of a **problem that occurs during a program's execution**
- Exception handling – **resolving exceptions** that may occur so program can continue or terminate gracefully
- Exception handling enables programmers to create programs that are more **robust** and **fault-tolerant**
  - Exception handling helps improve a program's fault tolerance
- Exception examples:
  - **`ArrayIndexOutOfBoundsException`** – an attempt is made to access an element past the end of an array
  - **`ClassCastException`** – an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator
  - **`NullPointerException`** – when a `null` reference is used where an object is expected

Exception Handling

- **Intermixing** program logic with error-handling logic can make programs **difficult** to read, modify, maintain and debug
- Exception handling enables programmers to remove error-handling code from the “main line” of the program’s execution
  - Improves **clarity**
  - Enhances **modifiability**
- If the potential problems occur infrequently, **intermixing** program and error-handling logic can **degrade a program’s performance**, because the program must perform (potentially frequent) tests to determine whether the task executed correctly and the next task can be performed

Divide By Zero Without Exception Handling (Example)

- Thrown exception – an exception that has occurred
- Stack trace
  - Name of the exception in a **descriptive message** that indicates the problem
  - Complete method-call stack
- **`ArithmeticException`** – can arise from a number of different **problems in arithmetic operations**
- Throw point – initial **point at which the exception occurs**, top row of call chain
- **`InputMismatchException`** – occurs when `Scanner` method `nextInt` receives a **string that does not represent a valid integer**

Divide By Zero Without Exception Handling

```
import java.util.Scanner;
public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient
   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input

      System.out.print( "Please enter an integer numerator: " );
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer denominator: " );
      int denominator = scanner.nextInt();
      int result = quotient( numerator, denominator );
      System.out.printf(
         "\nResult: %d / %d = %d\n", numerator, denominator, result );
   } // end main
} // end class DivideByZeroNoExceptionHandling
```

Handling ArithmeticExceptions and InputMismatchExceptions

- With exception handling, the program **catches** and **handles** the exception
- `try` block – encloses **code that might throw an exception** and the code that should not execute if an exception occurs
  - Consists of keyword **`try`** followed by a block of code enclosed in curly braces
- Exceptions may surface through explicitly mentioned **code in a `try` block**, through **calls to other methods**, through **deeply nested method calls** initiated by code in a `try` block or **from the Java Virtual Machine** as it executes Java bytecodes.

Catching Exceptions

- **`catch`** block – catches and handles an exception:
  - Begins with keyword **`catch`**
  - **Exception** parameter in parentheses – exception parameter identifies the exception type and enables `catch` block to interact with caught exception object
  - **Block of code** in curly braces that executes when exception of proper type occurs
- Matching `catch` block – the type of the exception parameter **matches the thrown exception type** exactly or is a superclass of it
- **Uncaught exception** – an exception that occurs for which there are no matching `catch` blocks
  - Cause program to terminate if program has only one thread; Otherwise only current thread is terminated and there may be adverse effects to the rest of the program

Termination Model of Exception Handling

- When an exception occurs:
  - `try` block **terminates** immediately (expires)
  - Program control **transfers to first matching `catch` block**
- After exception is handled:
  - **Termination model** of exception handling – program control **does not return to the throw point** because the `try` block has expired; Flow of control proceeds to the first statement after the last `catch` block
  - **Resumption model** of exception handling – program control resumes just after throw point
- `try` statement – consists of `try` block and corresponding `catch` and/or `finally` blocks

Using the `throws` Clause

- `throws` clause – specifies the exceptions a method may throw:
  - Appears after method's parameter list and before the method's body
  - Contains a **comma-separated list of exceptions**
  - Exceptions can be thrown by statements in method's body of by methods called in method's body
  - Exceptions can be of types listed in `throws` clause or subclasses
- If you know that a method might throw an exception, include appropriate exception-handling code in your program to make it more robust.

Divide By Zero With Exception Handling

```
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient
```

Divide By Zero With Exception Handling

```
public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominator: " );
            int denominator = scanner.nextInt();

            int result = quotient( numerator, denominator );
            System.out.printf( "\nResult: %d / %d = %d\n", numerator,
               denominator, result );
            continueLoop = false; // input successful; end looping
         } // end try
```

Divide By Zero With Exception Handling

```
        catch ( InputMismatchException inputMismatchException )
        {
           System.err.printf( "\nException: %s\n",
              inputMismatchException );
           scanner.nextLine(); // discard input so user can try again
           System.out.println(
               "You must enter integers. Please try again.\n" );
        } // end catch
        catch ( ArithmeticException arithmeticException )
        {
          System.err.printf( "\nException: %s\n", arithmeticException );
           System.out.println(
               "Zero is an invalid denominator. Please try again.\n" );
        } // end catch
      } while ( continueLoop ); // end do...while
   } // end main
} // end class DivideByZeroWithExceptionHandling
```

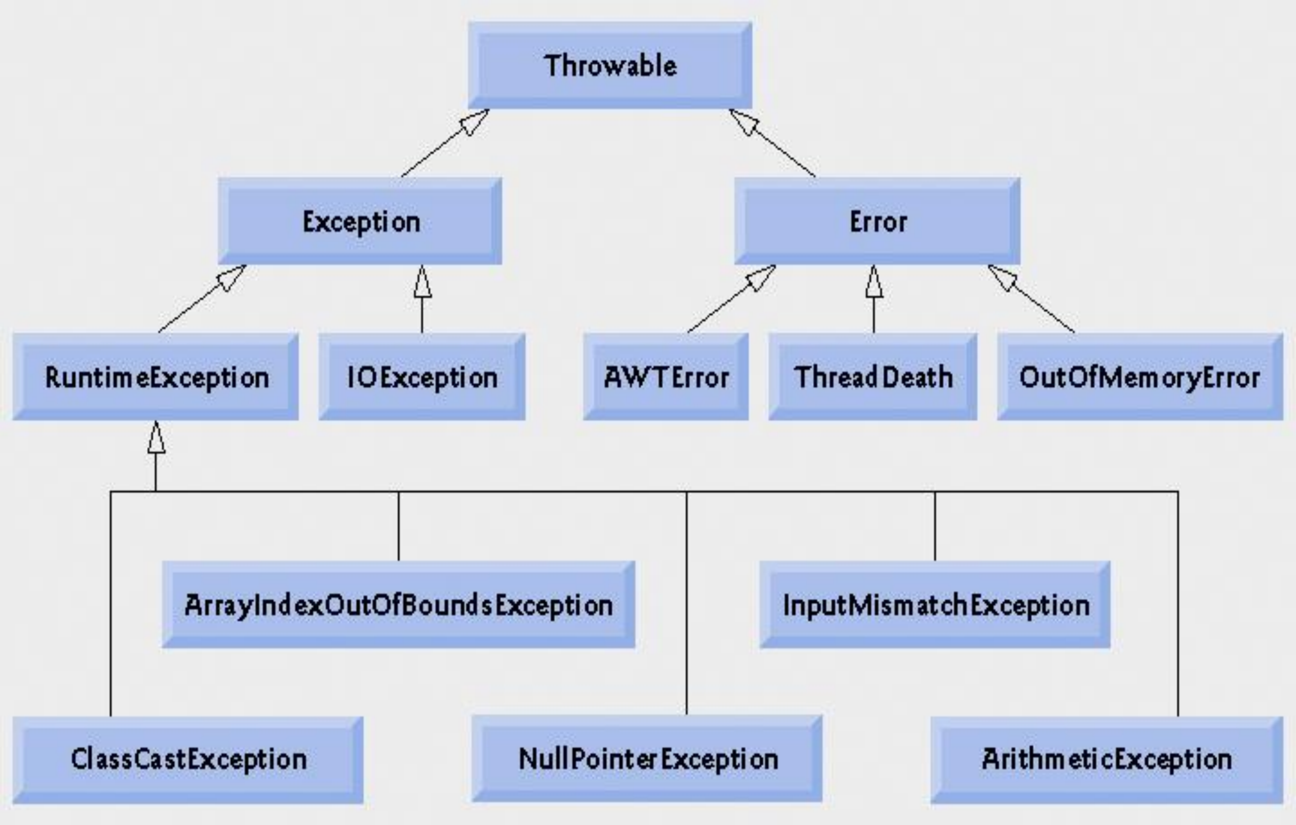
When to Use Exception Handling

- Exception handling designed **to process synchronous errors**
  - Synchronous errors – occur **when a statement executes**
  - Asynchronous errors – **occur in parallel with** and independent of the program’s flow of control
- Incorporate your exception-handling strategy into your system **from the design process’s inception**.
  - Including effective exception handling after a system has been implemented can be difficult
- Exception handling provides a **single, uniform technique** for processing problems
  - This helps programmers working on large projects understand each other’s error-processing code.

Java Exception Hierarchy

- All exceptions inherit either directly or indirectly from class `Exception`
- Exception classes form an inheritance hierarchy that can be extended
- Class **`Throwable`**, superclass of `Exception`
  - Only `Throwable` objects can be used with the exception-handling mechanism
  - Has two subclasses: `Exception` and `Error`
    - Class **`Exception`** and its subclasses **represent exception situations** that can occur in a Java program and that **can be caught** by the application
    - Class **`Error`** and its subclasses **represent abnormal situations** that could happen in the JVM – it is usually **not possible** for a program **to recover** from `Errors`

Java Exception Hierarchy



Java Exception Hierarchy

- Two categories of exceptions: checked and unchecked
  - **Checked** exceptions
    - Exceptions that inherit from class **`Exception`** but not from `RuntimeException`
    - **Compiler enforces** a **catch-or-declare** requirement
    - Compiler checks each method call and method declaration to determine whether the method `throws` checked exceptions.
      - If so, the compiler ensures that the checked exception **is caught** or is **declared in a `throws`** clause.
      - If not caught or declared, compiler error occurs.
  - **Unchecked** exceptions
    - Inherit from class **`RuntimeException`** or class **`Error`**
    - Compiler **does not check code** to see if exception is caught or declared
    - If an unchecked exception occurs and is not caught, the program terminates or runs with unexpected results
    - Can typically be **prevented by proper coding**

Java Exception Hierarchy

- Programmers are **forced to deal with checked exceptions**.
  - A compilation error occurs if a method explicitly attempts to throw a checked exception (or calls another method that throws a checked exception) and that exception is not listed in that method’s `throws` clause.
- If a subclass method overrides a superclass method, it is an **error for the subclass method to list more exceptions** in its `throws` clause than the overridden superclass method does.
  - However, a subclass’s `throws` clause can contain a subset of a superclass’s `throws` list.
- If your method calls other methods that explicitly throw checked exceptions, those exceptions **must be caught or declared** in your method
  - If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it

Java Exception Hierarchy

- Although the compiler does not enforce the catch-or-declare requirement for **unchecked exceptions**, **provide appropriate exception-handling code** when it is known that such exceptions might occur.
- For example, a program should process the `NumberFormatException` from `Integer` method `parseInt`, even though `NumberFormatException` (a subclass of `RuntimeException`) is an unchecked exception type.
  - This makes your programs more robust

Java Exception Hierarchy

- **`catch`** block catches **all exceptions** of its type **and subclasses** of its type
- If there are multiple `catch` blocks that match a particular exception type, **only the first matching `catch` block executes**
- It makes sense to use a `catch` block of a **superclass** when all the `catch` blocks for that class's subclasses will perform the same functionality

Java Exception Hierarchy

- Catching subclass types individually is subject to error if you forget to test for one or more of the subclass types explicitly
  - **catching the superclass** guarantees that objects of all subclasses will be caught
  - Positioning a `catch` block for the superclass type after all other subclass `catch` blocks for subclasses of that superclass ensures that all subclass exceptions are eventually caught
- Placing a `catch` block for a superclass exception type before other `catch` blocks that catch subclass exception types prevents those blocks from executing, so a **compilation error** occurs

`finally` block

- Programs that obtain certain resources must return them explicitly to avoid resource leaks
- finally block
  - Consists of finally keyword followed by a block of code enclosed in curly braces
  - Optional in a try statement
  - If present, is placed **after the last catch block**
  - **Executes whether or not an exception is thrown** in the corresponding try block or any of its corresponding catch blocks
  - Will not execute if the application exits early from a try block via method System.exit
  - Typically contains **resource-release code**.
    - This is also an effective way to eliminate resource leaks. For example, the finally block should **close any files** opened in the try block.

`finally` block

- If no exception occurs, `catch` blocks are skipped and control proceeds to `finally` block.
- After the `finally` block executes control proceeds to first statement after the `finally` block.
- If exception occurs in the `try` block, program skips rest of the `try` block.
  - First matching the `catch` block executes and control proceeds to the `finally` block.
  - If exception occurs and there are no matching `catch` blocks, control proceeds to the `finally` block.
  - After the `finally` block executes, the program passes the exception to the next outer the `try` block.
- **If `catch` block throws an exception**, the **`finally`** block **still executes**.

`finally` block

- Standard streams
  - `System.out` – standard output stream
  - `System.err` – standard error stream
- `System.err` can be used to **separate error output** from regular output
- `System.err.println` and `System.out.println` display data to the command prompt by default:

```
finally // executes regardless of what occurs in try...catch
{
  System.err.println(
             "Finally executed in doesNotThrowException" );
} // end finally
```

Throwing Exceptions Using the `throw` Statement

- `throw` statement – used to throw exceptions
  - Programmers can thrown exceptions themselves from a method **if something has gone wrong**
- `throw` statement consists of keyword **`throw` followed by the exception object**
- When `toString` is invoked on any `Throwable` object, its resulting string includes the descriptive string that was supplied to the constructor, or simply the class name if no string was supplied.
- An object can be thrown without containing information about the problem that occurred.
  - In this case, simple knowledge that an exception of a particular type occurred may provide sufficient information for the handler to process the problem correctly
- Exceptions can be thrown from constructors.
  - When an error is detected in a constructor, an exception should be thrown rather than creating an improperly formed object.

Rethrowing Exceptions

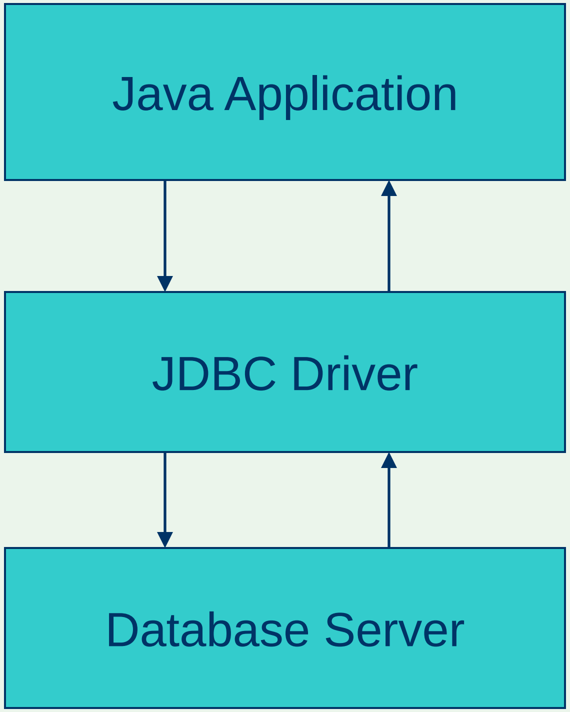
- Exceptions are rethrown when a `catch` block decides either that it **cannot process the exception** or that it can only partially process it
  - Exception is deferred to outer `try` statement
- Exception is rethrown by using keyword **`throw`** followed by a reference to the exception object
- If an exception has not been caught when control enters a `finally` block and the `finally` block throws an exception that is not caught in the `finally` block, the first exception will be lost and the exception from the `finally` block will be returned to the calling method.
- Avoid placing code that can `throw` an exception in a `finally` block.
  - If such code is required, enclose the code in a `try` statement within the `finally` block
- Assuming that an exception thrown from a `catch` block will be processed by that `catch` block or any other `catch` block associated with the same `try` statement can lead to logic errors

JDBC API

- Java programs interact with databases 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.

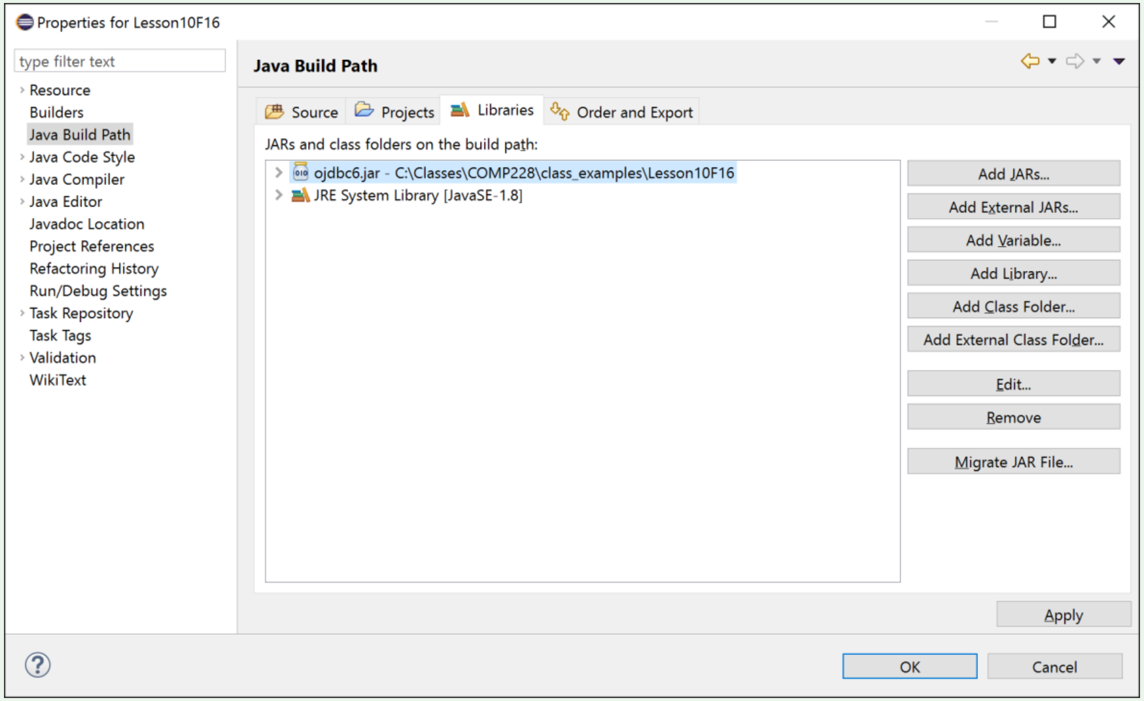
JDBC Drivers

❑ A JDBC driver is a software component that translates Java calls to a database language:



Oracle thin JDBC Driver

- The ojdbc6.jar can be downloaded from Oracle site or copied from C:\sqldeveloper**\jdbc\lib** folder of SQL Developer installation.
- Add the driver to the path of your application as an external jar file:



Properties for Lesson10F16

Java Build Path

JARs and class folders on the build path:

- ojdbc6.jar - C:\Classes\COMP228\class_examples\Lesson10F16
- JRE System Library [JavaSE-1.8]

Oracle thin JDBC Driver

- The Oracle JDBC driver class implements the java.sql.Driver interface.
- To access a database from a Java application, you must first use the **forName()** method of the java.lang.Class class to load the JDBC drivers directly. For example:

```
Class.forName ("oracle.jdbc.OracleDriver");
```

- Online documentation for OracleDriver class:

https://docs.oracle.com/cd/E11882_01/appdev.112/e13995/oracle/jdbc/OracleDriver.html

Accessing Databases with JDBC

- **JDBC is Java's Database API**.
- It is an object-oriented API that is consisted of a set of Java classes and interfaces that declare and/or define the necessary methods to:
  - Connect to a data source
  - Execute SQL statements
  - Process the results
- In order to use JDBC classes you need to import **java.sql** package in your programs

Accessing Databases with JDBC

- **Connection object represents a connection with a database**
- The simplest way to establish a connection to a data source is using the static method **getConnection**(String URL, String username, String password) of the class **DriverManager**.
- The **DriverManager** class locates the most appropriate driver that can connect a Java program with a particular database

Creating a connection

```
import java.sql.*;
public class TestConnection {
public static void main(String[] args) {
      try{
        //this loads the driver in memory
        Class.forName("oracle.jdbc.OracleDriver");
        Connection c =
             DriverManager.getConnection("jdbc:oracle:thin:@oracle1.centennialcollege.
             ca:1521:SQLD", user,password);
        }
        catch(ClassNotFoundException e) {
                JOptionPane.showMessageDialog(null, e.getMessage());
                                e.printStackTrace();
        }
        catch(SQLException e) {
                JOptionPane.showMessageDialog(null, e.getMessage());
                                e.printStackTrace();
        }
}
```

Creating a connection

- The syntax of url string is the following:

  jdbc:<subprotocol>:<subname>

  where:

  - **subprotocol** is **the name of the driver or the name of a database connectivity mechanism**, which may be supported by one or more drivers.
  - **subname** is the database path
- Note that all the JDBC code must be placed inside a **try block** and the exceptions must be handled in a **catch** block.

Connection Strings

RDBMS	Database URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
ORACLE	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`:`*databaseName*
DB2	`jdbc:db2:`*hostname*`:`*portNumber*`/`*databaseName*
Java DB/Apache Derby	`jdbc:derby:`*dataBaseName* (embedded) `jdbc:derby://`*hostname*`:`*portNumber*`/`*databaseName* (network)
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

Creating and executing a Statement

- To create a Statement object you should use the method **createStatement()** of Connection object:

  Statement st = c.**createStatement**();

- The Statement interface provides three different methods for executing SQL statements:
  - The method **executeQuery** is used for executing row-returning queries.
  - The method **executeUpdate** is used to execute INSERT, UPDATE, or DELETE statements and also SQL DDL (Data Definition Language) statements like CREATE TABLE and DROP TABLE.
  - The method **execute** is used to execute statements that return **more than one result set**, more than one update count, or a combination of the two

    ResultSet rs = st.**executeQuery**("SELECT * FROM EMP");

ResultSet object

- A ResultSet **contains all of the rows which satisfied the conditions in an SQL statement**, and it **provides access to the data in those rows through a set of get methods** that allow access to the various columns of the current row.

- ResultSet object uses a cursor to point to the current row.
  - Initially it is positioned before the first row.
  - The first call to **next()** method **sets the cursor to point to the first row**, **and so on**.

- The resultSet objects uses the **getXxx** methods to get column values for the current row:
  - getString
  - getInt
  - getFloat
  - getDouble
  - getLong
  - getObject
- You may use either **field name or field index to get its value**, as follows:

  String s = rs.**getString**(“name");

  String s = rs.**getString**(1);
- **The smallest index is 1**.

Connecting to and Querying a Database

- `OracleTest.java example:`
  - Drops table EMP if exists
  - Creates table EMP
  - Populates it with some rows
  - Retrieves the entire EMP table
  - Displays the data in text area
  - Example illustrates
    - Connect to the database
    - Query the database
    - Process the result

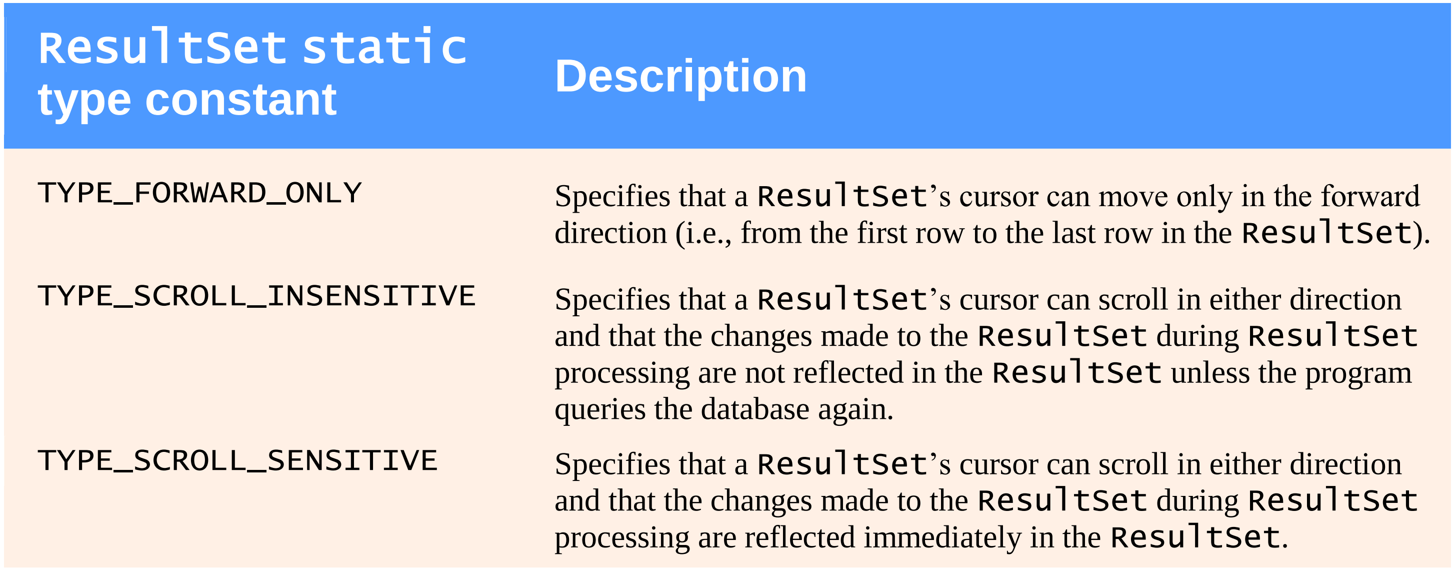
Connecting to and Querying a Database

- Information about the columns in a ResultSet is available by calling the method ResultSet.**getMetaData**.
- The **ResultSetMetaData** object returned gives the number, types, and properties of its ResultSet object's columns.
- The following code displays the rows of table EMP provided that all the fields are of String type:

```
ResultSetMetaData md = rs.getMetaData();;
while(rs.next())
{
        for( int i=1;i <= md.getColumnCount();i++)
      {
        System.out.println(rs.getString(i) + "|");
      }
}
```

Connecting to and Querying a Database

- Different types of Statement objects



ResultSet static type constant	Description
TYPE_FORWARD_ONLY	Specifies that a ResultSet's cursor can move only in the forward direction (i.e., from the first row to the last row in the ResultSet).
TYPE_SCROLL_INSENSITIVE	Specifies that a ResultSet's cursor can scroll in either direction and that the changes made to the ResultSet during ResultSet processing are not reflected in the ResultSet unless the program queries the database again.
TYPE_SCROLL_SENSITIVE	Specifies that a ResultSet's cursor can scroll in either direction and that the changes made to the ResultSet during ResultSet processing are reflected immediately in the ResultSet.

Connecting to and Querying a Database

- Different types of Statement objects

ResultSet static concurrency constant	Description
CONCUR_READ_ONLY	Specifies that a ResultSet cannot be updated (i.e., changes to the ResultSet contents cannot be reflected in the database with ResultSet's update methods).
CONCUR_UPDATABLE	Specifies that a ResultSet can be updated (i.e., changes to the ResultSet contents can be reflected in the database with ResultSet's update methods).

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT * FROM EMP");
```

References

- Textbook
- Java DB documentation:
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html
  - http://docs.oracle.com/javadb/index.html
- http://www.java2s.com/Code/Java/Database-SQL-JDBC/CatalogDatabase-SQL-JDBC.htm