Database APIs

Database Programming Techniques

ACS 575: Database Systems

Instructor: Dr. Jin Soung Yoo, Professor
Department of Computer Science
Purdue University Fort Wayne

References

- W. Lemanhieu, et al., Principles of Database Management Ch. 15
- R. Elmasri et al., Fundamentals of Database Systems, Ch 10, 11

Outline

- Introduction
- Database API Types
- Embedded API
- Call-level API

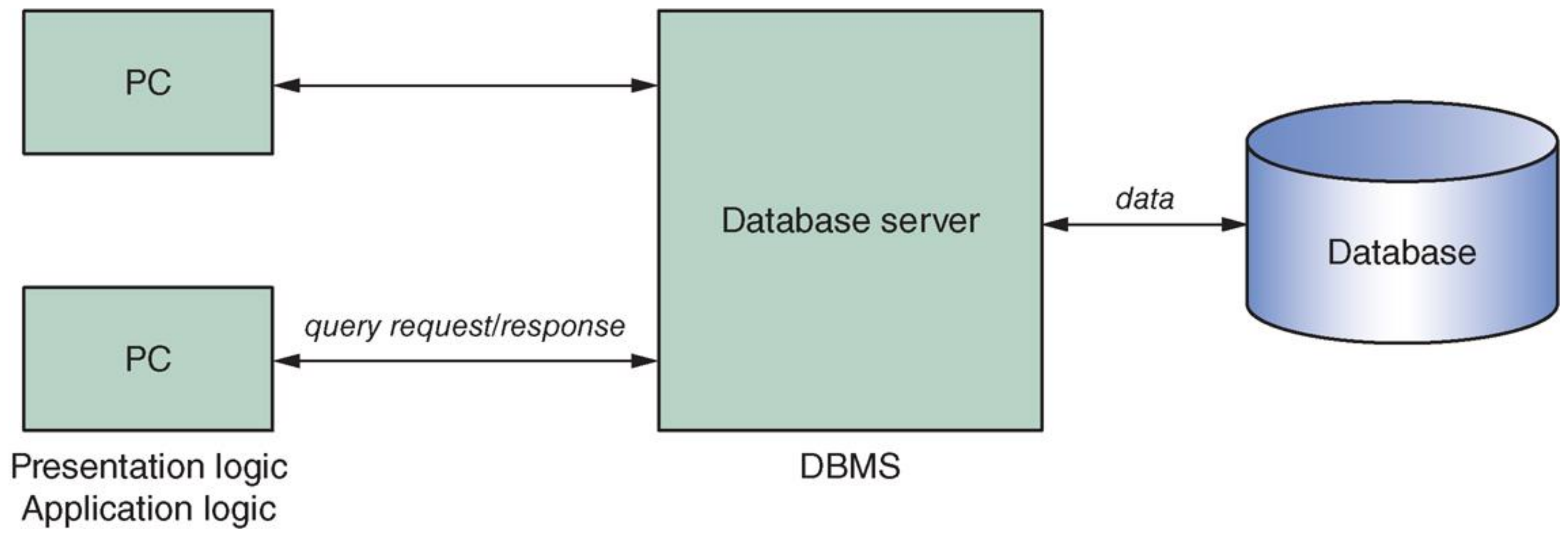
Database Programming

- Objective:
  - To access a database from an application program (as opposed to interactive interfaces)
- Why?
  - An interactive interface (like MySQL Command and SQL Developer tools) is convenient <u>but not sufficient</u>
  - A majority of database operations are made thru application programs (increasingly thru web applications)
  - We can implement, via general programming languages,
    - Complex computational processing of the data.
    - Specialized user interfaces.

Tiered System Architectures

- Tiered system architectures operate in networked environments
- Tiered system architectures aim to decouple the centralized server setup by combining the computing capabilities of powerful central computers with the flexibility of PCs
- Multiple variants of this architecture: **two-, three-, or *n*-tier architectures**, depending on the placement of the three types of application logic.

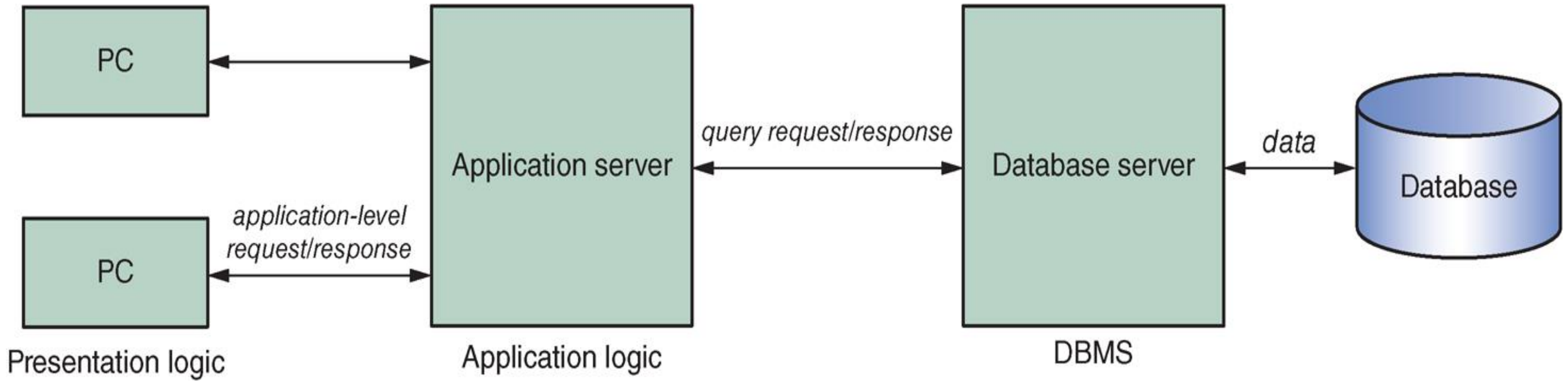
Two-tier Client and Server Architecture



- Processing of an application distributed between font-end clients and back-end servers
  - **Client**–Workstation (usually a PC) that requests and uses a service
  - **Server**–Computer (PC/mini/mainframe) that provides a service
- For DBMS, server is a database server

Three-tier Architecture

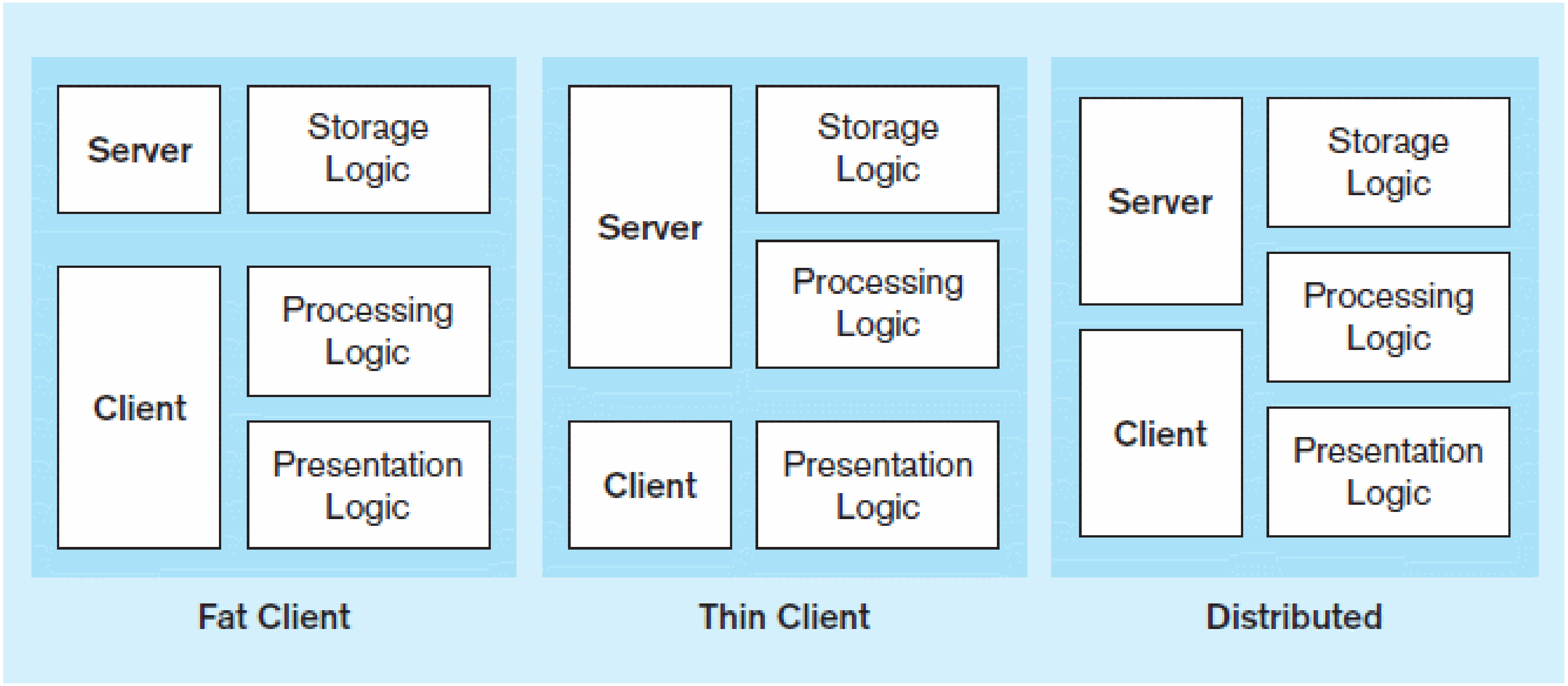
- Three-tier architecture decouples application logic from the application program and DBMS, and puts it in a separate layer (i.e., application server).
- Note: “application server” or “database server” may consist of multiple physical, distributed machines



Application Logic Components

Presentation Logic Input–keyboard/mouse Output–monitor/printer	- For formatting and presenting data on the user’s screen or other device, and for managing user input from a keyboard or other input device. - Graphical user Interface (GUI)
Processing (Application) Logic Data processing logic Business rules logic Data management logic	- Procedures, functions, programs for • Data validation and identification of processing errors • Business rules that have not been coded at the DBMS level • Identifying the data necessary for processing the transaction or query
Storage Logic Data storage/retrieval	- Responsible for data storage and retrieval from the physical storage devices associated with the application - DBMS activities

Common Logic Distributions in Two-tier Client Server Architecture

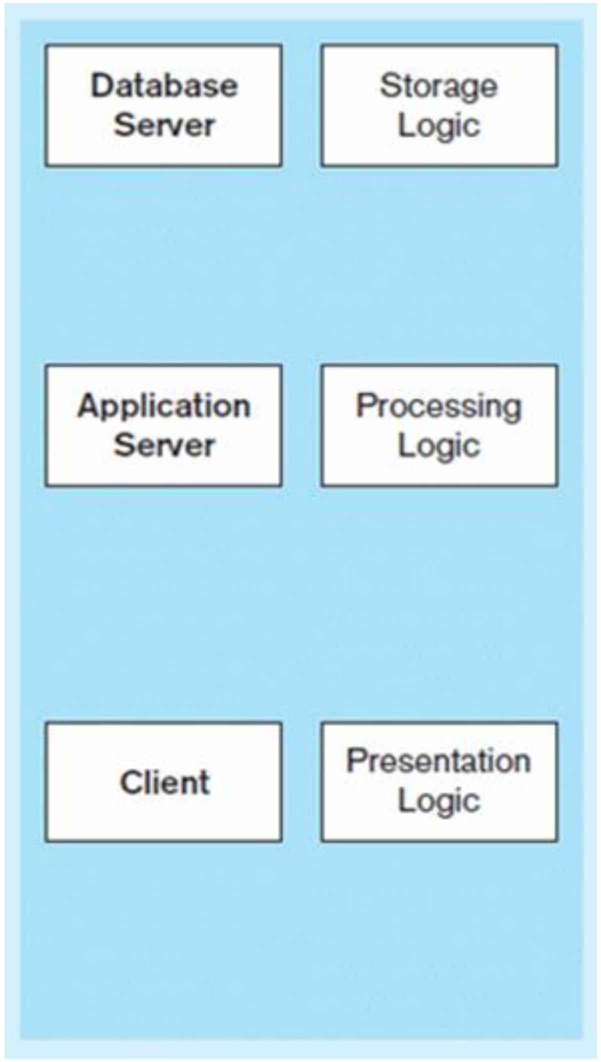


(a) A typical two-tier client server

(b) A variant of two-tier client server

(c) A variant of two-tier client server

Application Logic Distribution in Three-tier Architecture



Database API

- In a tiered DBMS system architecture,
  - client applications are able to query database servers and receive the results, and
  - database server receives calls made by clients and executes the relevant operations before returning the results
- Client applications that wish to utilize the services provided by a DBMS are commonly programed to use a specific **application program interface** (**API**) provided the DBMS.
- The **database API** exposes an interface through which client applications can access and query a DBMS.

Database API (cont.)

- In a two-tiered client-server architecture with a fat client, database API interface is present on <u>the client's side</u>, together with the applications logic
- In an *n*-tiered environment, the database API interface is present on <u>the application server's side</u>.

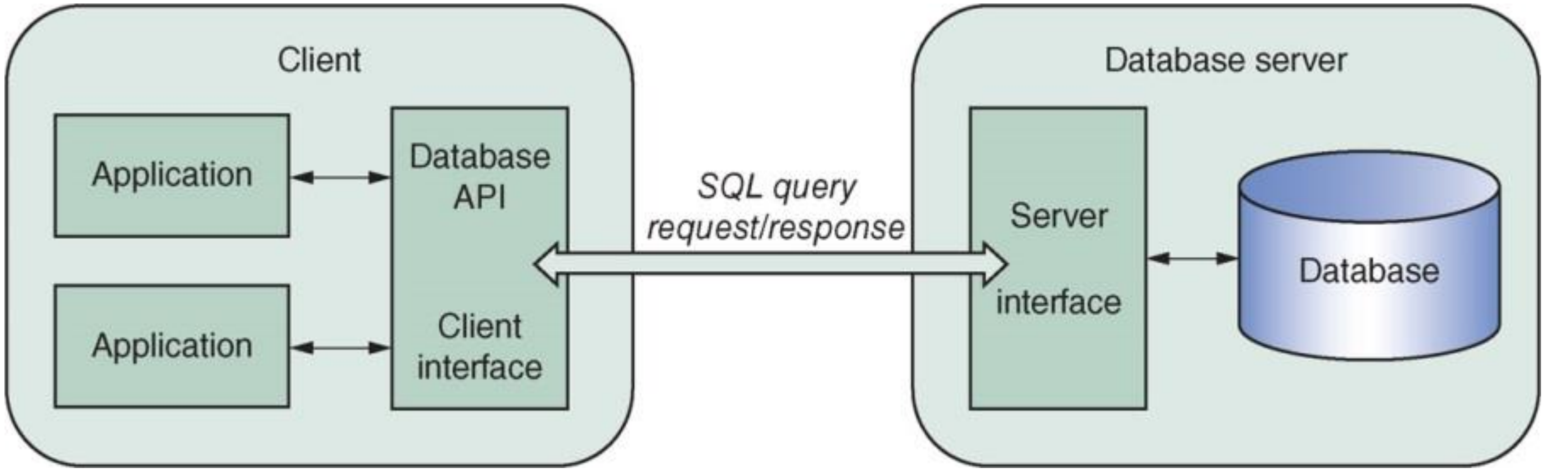


Figure. Position of the database API in a two-tiered client-server architecture.

Steps for Using Databases via Database API

No matter which API or language is used, the basic steps for accessing a database from an application are similar.

1. Open a connection to a database.
2. Send a database operation (query, update, insert, and delete) to the database.
3. Process the results of the query if the operation is for query.
4. Repeat steps 2–3 as necessary.
5. Close the connection to the database.

Benefits of Database API

- When the database API exposes an interface through which other parties can utilize the services provided by the DMBS, it hides network-related aspects (e.g., by enabling clients to access a DBMS as if it were running locally)
- Application programmers do not have to concern themselves with implementing a full communication protocol, but can focus on creating their applications and talking to the DBMS through a common langue, SQL.
- <u>The Database API is then responsible to implement and handle the underlying protocols and formats</u>

Outline

- ☐ Introduction
- ☞ Database API Types
- ☐ Embedded API
- ☐ Call-level API

Database API Types

- Proprietary vs. Universal
  - Proprietary, DBMS-specific API
  - Universal API
- Embedded vs. Call-level
  - Embedded API
  - Call-level API
- Early Binding vs. Late Binding

Proprietary API

- Most DBMS vendors provide a proprietary, **DBMS-specific API** together with the DBMS software
- Disadvantages
  - Client applications must be aware of the DBMS that will be utilized on the server-side
  - If a different DBMS is to be used, the client application needs to be modified to interact with the new database API.
  - This creates an often <u>undesirable dependency between applications and DBMS.</u>

Universal API

- To overcome the issues of DMBS-specific API, many generic, vendor-agnostic **universal APIs** have been proposed.
- Universal APIs differ in terms of
  - embedded or call-level
  - programming language(s)
  - functionalities
- Applications can be easily ported to multiple DBMSs
- However, differences in support regarding different versions of the SQL standard and vendor-specific extensions or interpretative details can still lead to issues regarding portability.
- Some vendor-specific optimization and performance tweaks cannot be utilized with the universal API.

Embedded APIs vs. Call-Level APIs

- **Embedded APIs**
  - An embedded API embeds SQL statements in the host programming language
  - Before the program is compiled, an "SQL pre compiler" parse the SQL-related instructions and replaces these with host language codes. Then the actual compiler constructs a runnable program.
- **Call-level APIs**
  - SQL instructions are passed to the DBMS by means of direct calls to a series of procedures, functions, or methods as provided by the API to perform the necessary actions

Early Binding vs Late Binding

- **SQL binding** refers to the translation of SQL code to a lower-level representation that can be executed by the DBMS, after performing tasks such as validation of table and column names, user access right and a query plan.
- Early versus late binding then refers to the actual moment when this binding step is performed.
- **Early Binding**
  - The binding step happens before the program is executed
  - Typically occurs if a pre-compiler (embedded API) is used.
- **Late Binding**
  - The binding of SQL statements are performed at runtime.
  - Often used for call-level API

Characteristics of Early Binding

- Early binning typically occurs if embedded API is used. However, it can be also combined with call-level APIs which call a stored procedure, which is early-bound by the DBMS.
- Binding only needs to be performed once, which can result in a significant performance benefit if the same query has to be executed many times afterwards.
- Because early binding typically occurs if a pre-compiler is used, the pre-compiler can perform specific syntax check and immediately warn the program if badly formatted SQL statements are present or table names are misspelled.
- Errors are detected before the actual execution of the code, rather than causing the program to crash or malfunction because the errors occur during the execution

Characteristics of Late Binding

- Late binding gives additional flexibility – SQL statements can be generated at runtime, so this is also called “dynamic SQL” rather than “static SQL” as employed by early binding.
- A drawback of this approach is that syntax errors or authorization issues will remain hidden until the program is executed
- The SQL statements will typically look like textual “strings” placed in the source code of the program, and are hence treated as such during compilation without additional checks taking place. This can make testing the application harder.

Characteristics of Late Binding (cont.)

- Late binding <u>is less efficient</u>, <u>especially for queries that must be executed multiple times,</u> because the binding is repeated with every execution for the query.
- <u>However, most APIs provide support for parametrized prepared queries</u>, so the same SQL statement can be executed using different input parameters, e.g., subsequently retrieving different customers according to different values of the customer ID.
- The DBMS will still try to optimize toward the most efficient query plan, even though the actual values for the input parameters are not yet given during the preparation of the statement

Summary

	Embedded APIs	Call-level APIs
Early binding (“static” SQL)	**Possible as a pre-compiler is used** • Performance benefit, especially when the same query must be executed many times • Pre-compiler detects errors before the actual execution of the code • SQL queries must be known upfront	**Only possible through stored procedures** which is early-bound by the DBMS.
Late binding (“dynamic” SQL)	**Not used with embedded APIs**	**Necessary as no pre-compiler is used** • Flexibility benefit: SQL statements can be dynamically generated and used during execution • Errors are only detected during program execution • Possibility to use prepared SQL statements to perform binding once during execution

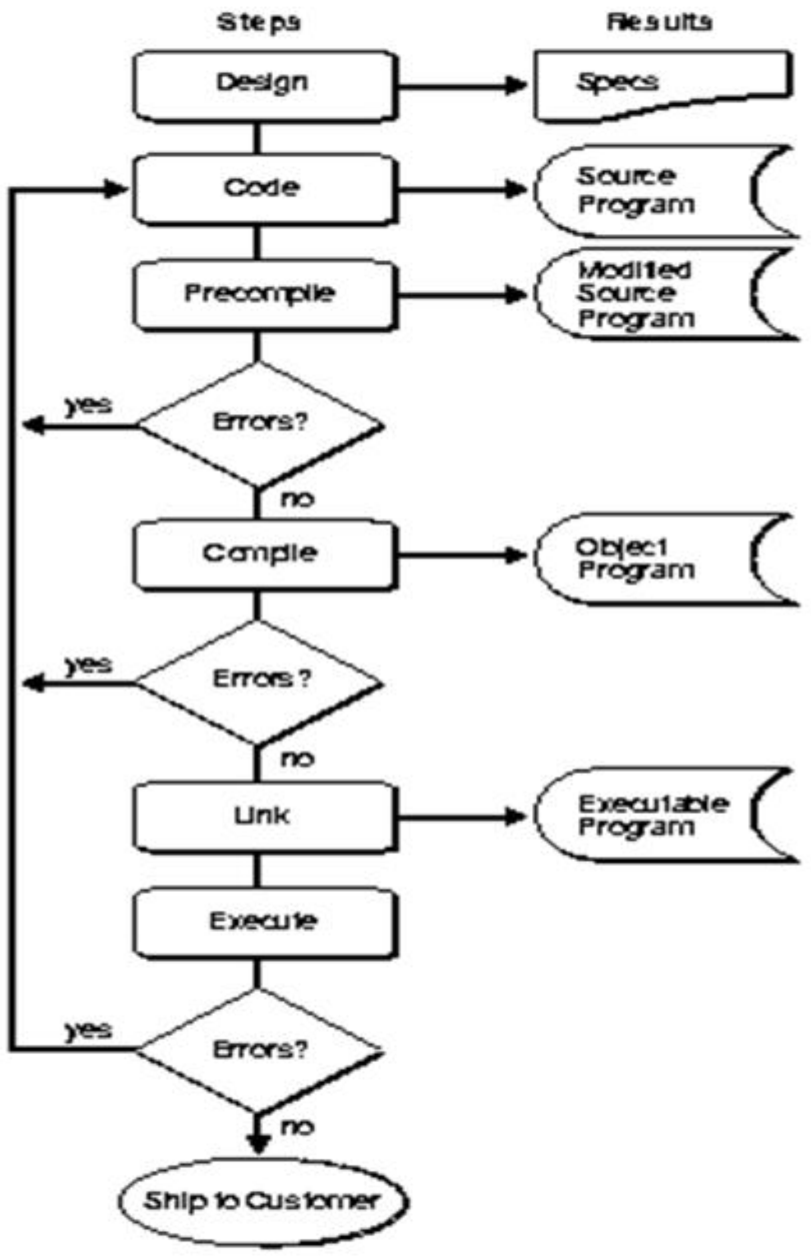
Outline

- Introduction
- Database API Types
- ☞ **Embedded API**
  - Embedding SQL
  - SQLJ
- Call-level API

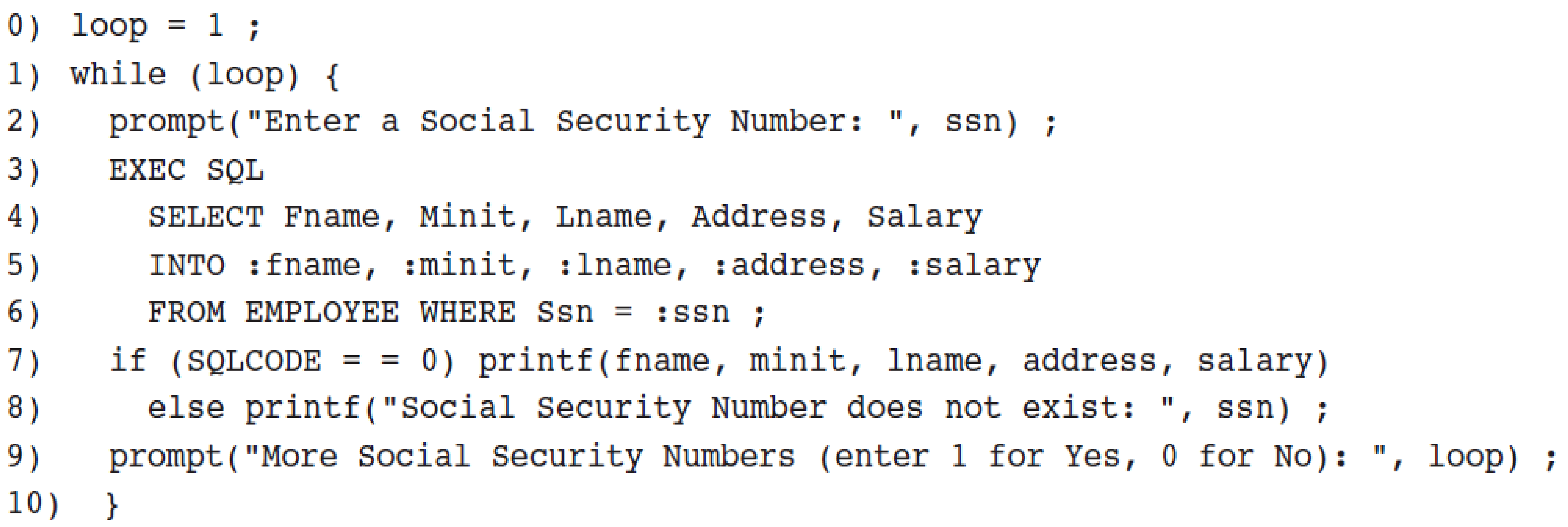
Embedded API

- **Embedded API** embeds SQL statements in the host programming language, meaning that SQL statement(s) will end up being an integral part of the source code of a program.
- Before the program is compiled, a "SQL pre-compiler" parses the SQL-related instructions and replaces these with source code instructions native to the host programming language used, invoking a separate code library.
- Converted source code is then sent to the actual compiler to construct a runnable program.
- Example: SQLJ, Oracle Pro*C

Embedded SQL Application Development Process



Example of Embedded SQL Programming



```
loop = 1 ;
while (loop) {
  prompt("Enter a Social Security Number: ", ssn) ;
  EXEC SQL
    SELECT Fname, Minit, Lname, Address, Salary
    INTO :fname, :minit, :lname, :address, :salary
    FROM EMPLOYEE WHERE Ssn = :ssn ;
  if (SQLCODE = = 0) printf(fname, minit, lname, address, salary)
    else printf("Social Security Number does not exist: ", ssn) ;
  prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
  }
```

Figure: A C program segment with embedded SQL – A single row query

Example of Embedded SQL Programming

```
prompt("Enter the Department Name: ", dname) ;
EXEC SQL
  SELECT Dnumber INTO :dnumber
  FROM DEPARTMENT WHERE Dname = :dname ;
EXEC SQL DECLARE EMP CURSOR FOR
  SELECT Ssn, Fname, Minit, Lname, Salary
  FROM EMPLOYEE WHERE Dno = :dnumber
  FOR UPDATE OF Salary ;
EXEC SQL OPEN EMP ;
EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
while (SQLCODE = = 0) {
  printf("Employee name is:", Fname, Minit, Lname) ;
  prompt("Enter the raise amount: ", raise) ;
  EXEC SQL
    UPDATE EMPLOYEE
    SET Salary = Salary + :raise
    WHERE CURRENT OF EMP ;
  EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
  }
EXEC SQL CLOSE EMP ;
```

Figure: A C program segment with embedded SQL – A multiple row query using CURSOR

Cursor in SQL

- A **cursor** is a database query tool that allows iteration over query results row by row.
  - Cursors are especially useful for processing data one record at a time in a controlled, sequential manner within a host language program.
- Usage:
  - **Declaration**: Declared simultaneously with the SQL query.
  - **Open**: The `OPEN CURSOR` command fetches the query result and positions the cursor before the first result row.
  - **Fetch**: The `FETCH` command advances the cursor to the next row, making it the current row, and transfers the data to program variables.
  - **Close**: The `CLOSE CURSOR` command is used once query processing is complete, freeing up resources.

Example of Oracle Pro*C Embedded SQL

```
#include <stdio.h>
#include <sqlca.h>

char emp_name[10];
int emp_number;    int dept_number;
char temp[32];

main() {
    /* declare a cursor */
    EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT ename FROM emp WHERE deptno = :dept_number;

     printf("Department number? "); gets(temp); dept_number = atoi(temp);

    /* open the cursor and identify the active set */
     EXEC SQL OPEN emp_cursor;
    /* fetch and process data in a loop exit when no more data */
    EXEC SQL WHENEVER NOT FOUND DO break;
    while (1) {
      EXEC SQL FETCH emp_cursor INTO :emp_name;
            printf("%s\n", emp_name);
      }
     EXEC SQL CLOSE emp_cursor;
     exit(0);
}
```

The complete of this example is at https://docs.oracle.com/en/database/oracle/oracle-database/12.2/lnpcc/embedded-SQL.html

Example (cont.)

- To build an Oracle Pro*C/C++ application
  - **Precompile** the Pro*C/C++ source file by using the proc command from your system prompt.

    For example:

    ```
    % proc iname=sample.pc
    ```

    The proc utility takes a .pc source file as input and produces a .c file.
  - **Compile** the resulting C code file.

    ```
    % gcc -c sample.c
    % gcc -o sample sample.o
    ```

SQLJ

- **SQLJ**, Java's embedded, static SQL API developed after JDBC, allows embedding SQL statements directly into Java programs
- Queries directly embedded in Java source code
  - Arguments for embedded SQL statements are passed through host variables (using "**:**" prefix)
  - Pre-compiler converts these statements into native Java code
  - Pre-compiler also performs additional checks
- However SQLJ are not popular due to
  - lack of support for dynamic SQL
  - extra overhead for programmer

SQLJ Example I

```
...
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
Connection conn = DriverManager.getConnection(dbUrl);
DefaultContext.setDefaultContext(new DefaultContext(conn));
// Define an SQLJ iterator
#sql iterator BookIterator(String, String, int);
// Perform query and fetch results
BookIterator books;
int min_pages = 100;
#sql books = {select title, author, num_pages
             from books where num_pages >= :min_pages };

String title; String author; int num_pages;
#sql {fetch :books into :title, :author, :num_pages};
while (!books.endFetch()) {
  System.out.println(title + ‘ by ‘ + author + ‘: ‘ + num_pages);
  #sql {fetch :books into :title, :author, :num_pages};
}
conn.close();
...
```

SQLJ Example II

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
#sql iterator MyIter (String ename, int empno, float sal);
public class MyExample
{
  public static void main (String args[]) throws SQLException
   {
     Oracle.connect
        ("jdbc:oracle:thin:@oow11:5521:sol2", "scott", "tiger");

     #sql { insert into emp (ename, empno, sal)
        values ('SALMAN', 32, 20000) };
     MyIter iter;
     #sql iter={ select ename, empno, sal from emp };
     while (iter.next()) {
        System.out.println
           (iter.ename()+" "+iter.empno()+" "+iter.sal());
     }
  }
}
```

Pros and Cons of Embedded API

- Advantages
  - The pre-compiler can also perform specific syntax checks to make sure the embedded SQL is correct.
- Disadvantages
  - The mixture between host language code and SQL statements can lead to harder-to-maintain code
- Overall, embedded database APIs become rare in contemporary DBMS implementations.

Outline

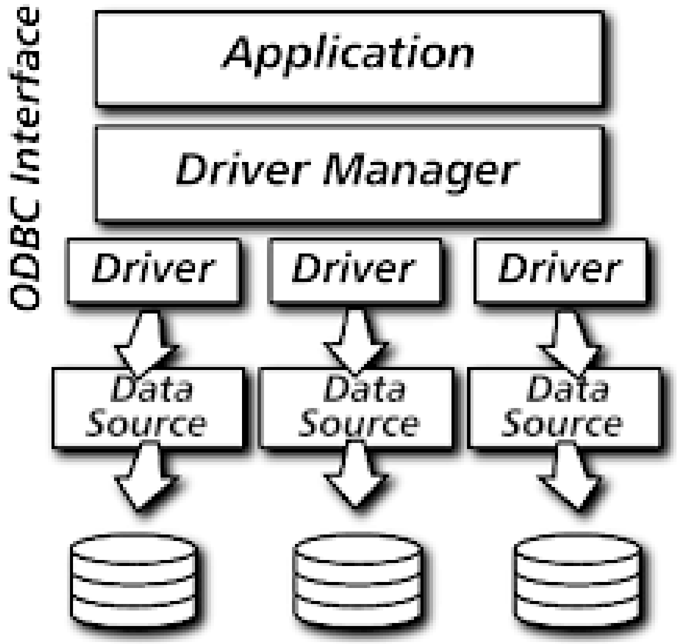
- Introduction
- Database API Types
- Embedded API
- ☞ **Call-level API (and Universal API)**
  - ODBC
  - OLE DB
  - ADO.NET
  - JDBC
  - Language-integrated Querying
  - Object Persistence and ORMS

Call-level API

- **Call-level API** works by passing SQL instructions to the DBMS by means of direct calls to a series of procedures, functions or methods as provided by the API to perform the necessary actions, such as setting up a database connection, sending queries, and iterating over the query result.
- Call-level APIs were developed in the early 1990s and standardized by the International Organization of Standardization (ISO) and International Electrotechnical Commission (IEC)
- The most widespread implementation of the standard is found in the ODBC

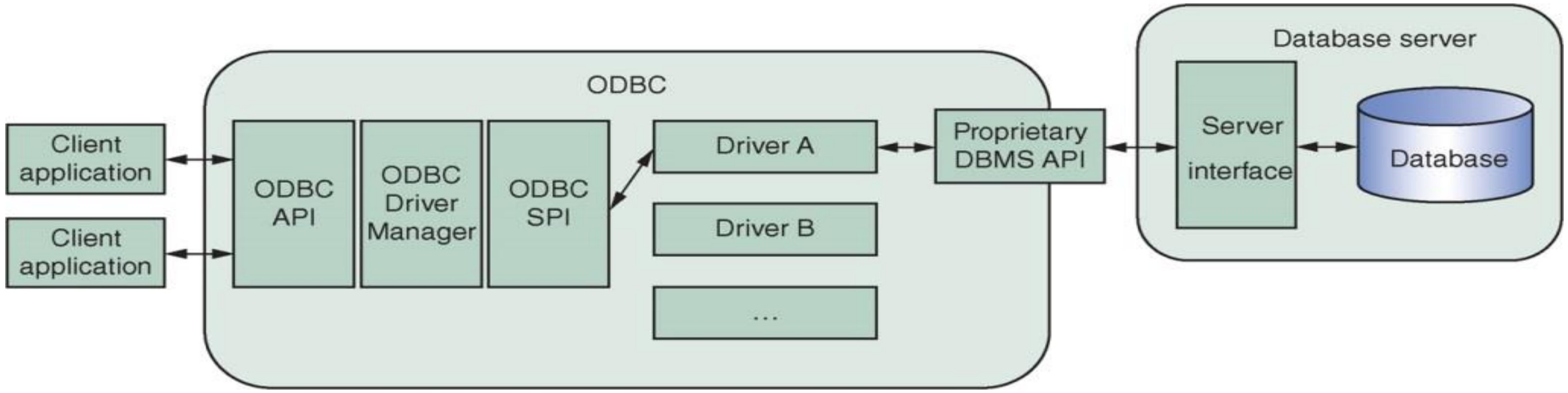
ODBC

- **Open DataBase Connectivity (ODBC)** is open standard, developed by Microsoft, with the aim to offer applications a common, uniform interface to various DBMSs

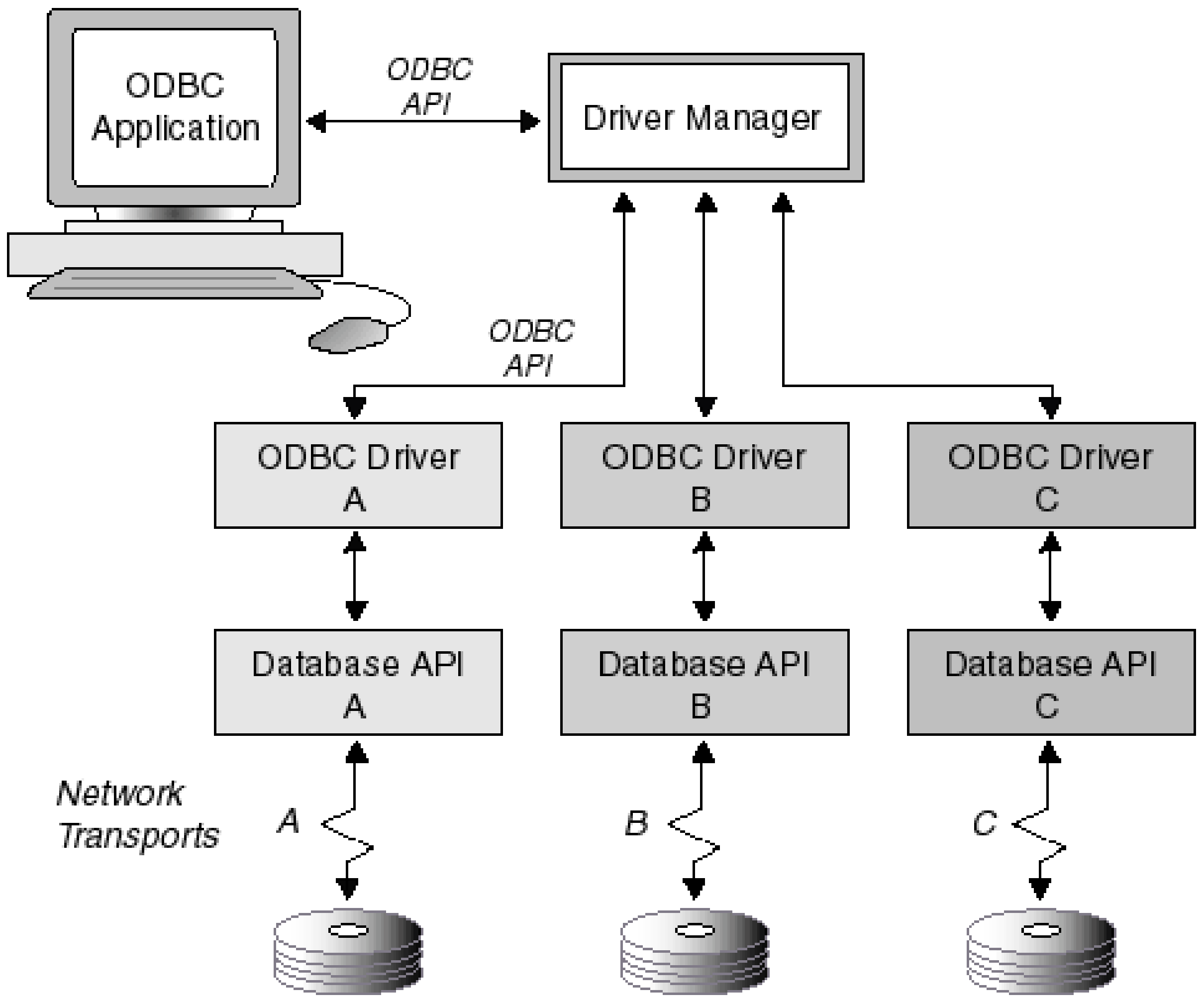


ODBC

- ODBC consists of 4 main components
  - **ODBC API**: universal interface through which client applications will interact with a DBMS (a call-level API)
  - **ODBC Driver Manager**: responsible for selecting the correct Database Driver to communicate with a DBMS
  - **Database Driver**: collection of routines that contain the actual code to communicate with a DBMS
  - **Service Provider Interface**: separate interface implemented by the DBMS vendor by which the Driver Manager interacts with various drivers



ODBC Model with Different DBMSs



ODBC Program Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>

#include "util.c"
...
int main () {
...
// Set ODBC Version
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3,
                            0);
...
// Connect to DSN
    retcode = SQLConnect(hdbc, (SQLCHAR*) "DATASOURCE", SQL_NTS,(SQLCHAR*) NULL, 0,
                       NULL, 0);
...
// Bind Parameters to all fields
    retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                 FIRSTNAME_LEN, 0, strFirstName, FIRSTNAME_LEN, &lenFirstName);
....
// Prepare a SQL statement
retcode = SQLPrepare(hstmt, (SQLCHAR*)"INSERT INTO TestTBL1"
             "(FirstName, LastName, Address, City) VALUES (?, ?, ?, ?)", SQL_NTS);
...
retcode = SQLExecute(hstmt);
...
return 0;
}
```

ODBC

- ODBC allows applications to be easily ported between DBMSs
- Disadvantages
  - ODBC is native to Microsoft-based platforms
  - ODBC is based on the C language (↔ OO)
  - ODBC middleware introduces an extra layer of indirection (performance ↓)

ODBC Reference

- https://docs.oracle.com/database/121/ADFNS/adfns_odbc.htm#ADFNS1136
- https://www.easysoft.com/developer/languages/c/examples/index.html
- https://help.interfaceware.com/v6/connect-to-oracle-with-easy-connect#setup-windows-oci

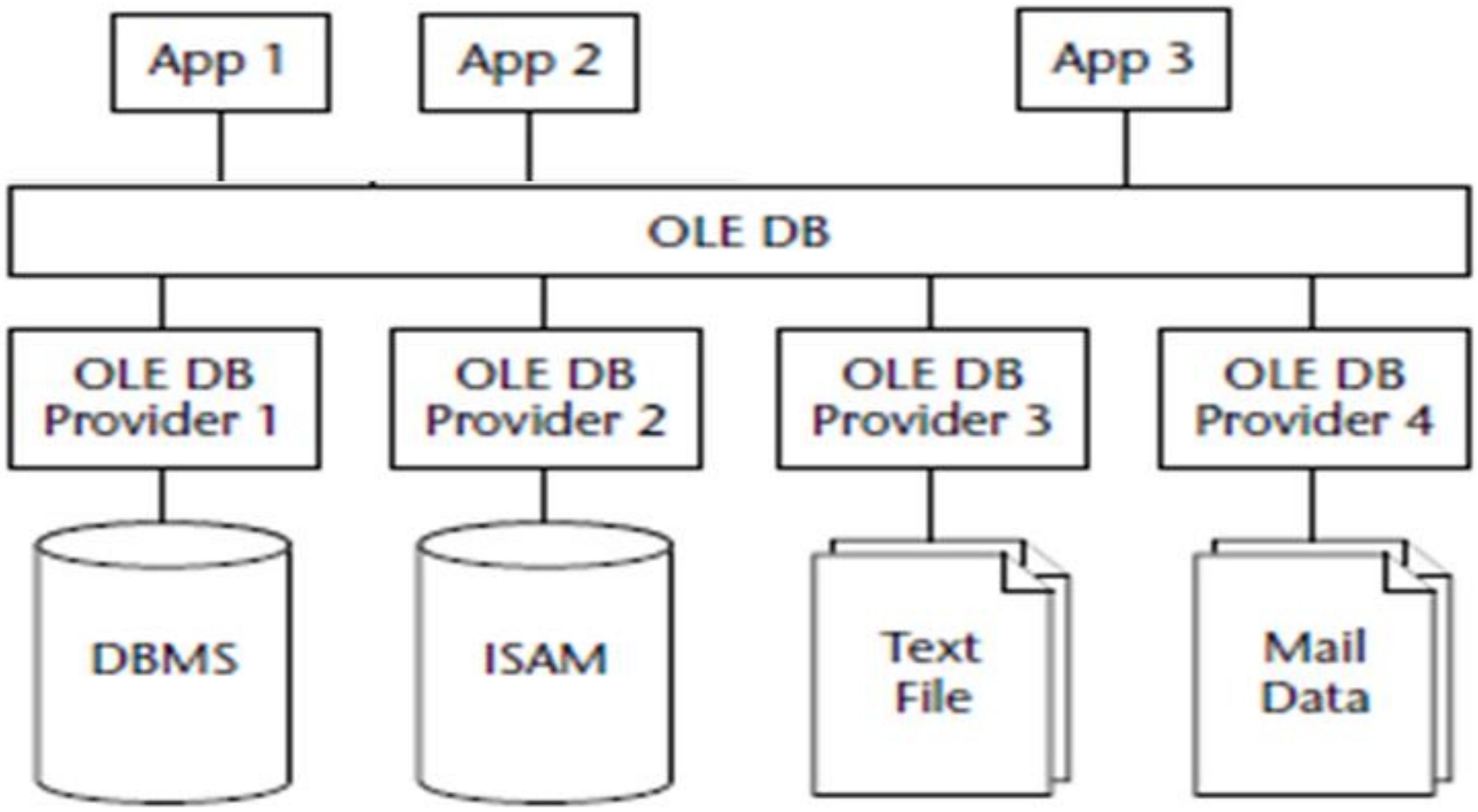
OLE DB

- Microsoft originally intended OLE DB as a higher-level replacement for, and successor to, ODBC.
- **OLE DB (Object Linking and Embedding for DataBases**, sometimes written as OLEDB or OLE-DB**)** is an API designed by Microsoft, allows accessing data from a variety of sources in a uniform manner.
  - OLE DB represents Microsoft's attempt to move towards a "Universal Data Access" approach
  - OLE DB extends its feature set to support a wider variety of non-relational databases, such as object databases and spreadsheets, as well as relational databases and offers unified access capabilities.

OLE DB (cont.)

- The OLE DB API provides a set of (COM) interfaces implemented using Microsoft's **Component Object Model (COM)**
  - COM is a programming framework for specifying software components, which represent highly modular and re-suable building bocks to applications running on a Microsoft platform

OLE DB Environment



OLE DB Provider

- An **OLE DB provider** is a software component that enables an OLE DB consumer (i.e., the application that need access to the data) to interact with a data source.
- OLE DB providers are analogous to ODBC drivers, JDBC drivers, and ADO.NET data providers.
- OLE DB providers can be created to access such simple data stores as a text file and speared sheet, through to such complex databases as Oracle, MS SQL Server, and many others

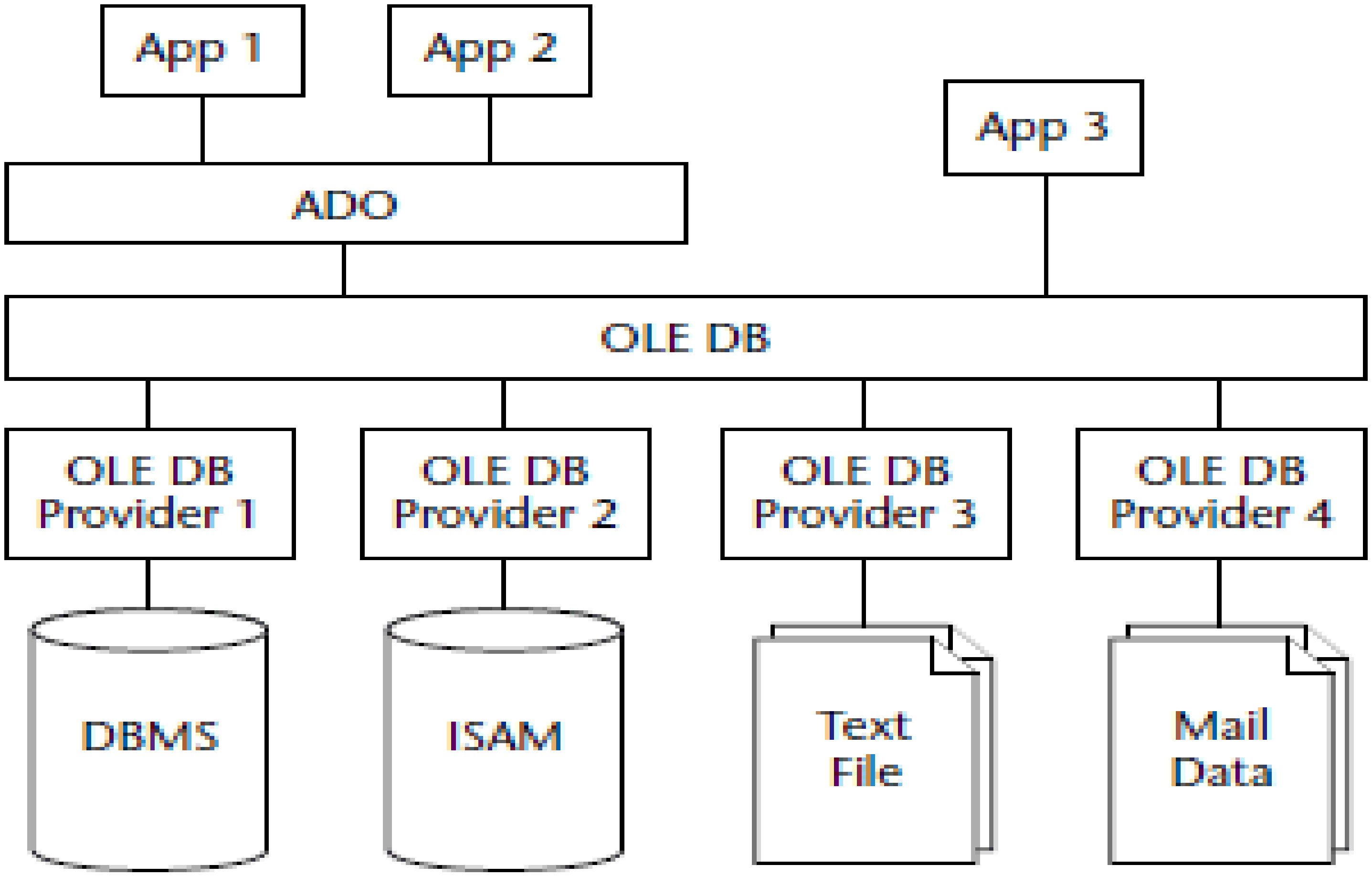
OLE DB example with C#

```
public void InsertRow(string connectionString, string insertSQL)
{
    using (OleDbConnection connection = new OleDbConnection(connectionString))
    {
        // The insertSQL string contains a SQL statement that
        // inserts a new row in the source table.
        OleDbCommand command = new OleDbCommand(insertSQL);

        // Set the Connection to the new OleDbConnection.
        command.Connection = connection;

        // Open the connection and execute the insert command.
        try {
            connection.Open();
            command.ExecuteNonQuery();
        }
        catch (Exception ex) {
            Console.WriteLine(ex.Message);
        }
        // The connection is automatically closed when the
        // code exits the using block.
    }
}
```

ADO with OLE DB

- OLE DB can also combined with **ActiveX Data Objects (ADO)**, which provides a richer, more ‘programmer-friendly’ programming model on top of OLE DB.
- ADO provides a consistent, higher level interface to the various OLE DB providers.
- For example,
  - The data resides in a relational database for which there currently exists an ODBC driver. The application uses ADO to talk to the OLE DB Provider for ODBC, which then loads the appropriate ODBC driver. The driver passes the SQL statement to the DBMS, which retrieves the data
  - The data resides in Microsoft SQL Server for which there is a native OLE DB provider: The application uses ADO to talk directly to the OLE DB Provider for Microsoft SQL Server. No intermediaries are required.

ADO with OLE DB



Example: Using ADO with OLE DB for SQL Server

```
Dim con As New ADODB.Connection
Dim recordSet As ADODB.Recordset

con.ConnectionString=“Provider=MSOLEDBSLQ;” _
      & “Sever=(local);” _
      & “Database=CompanyWorks;” _
      & “Integrated Security=SSPI;” _
      & “DataTypeCompatibility=80;”

con.Open

Set qry = “select nr, name from suppliers where status < 30”
Set recordSet = con.Execute(qry)

Do While Not recordSet.EOF
 MsgBox(recordSet.Fields(0).Name & “ = “ & recordSet.Fields(0).Value & vbCrLf & _
       recordSet.Fields(1).Name & “ = “ & recordSet.Fields(1).Value)
 recordSet.MoveNext
Loop

recordSet.Close
con.Close
```

https://docs.microsoft.com/en-us/sql/connect/oledb/applications/using-ado-with-oledb-driver-for-sql-server?view=sql-server-ver15

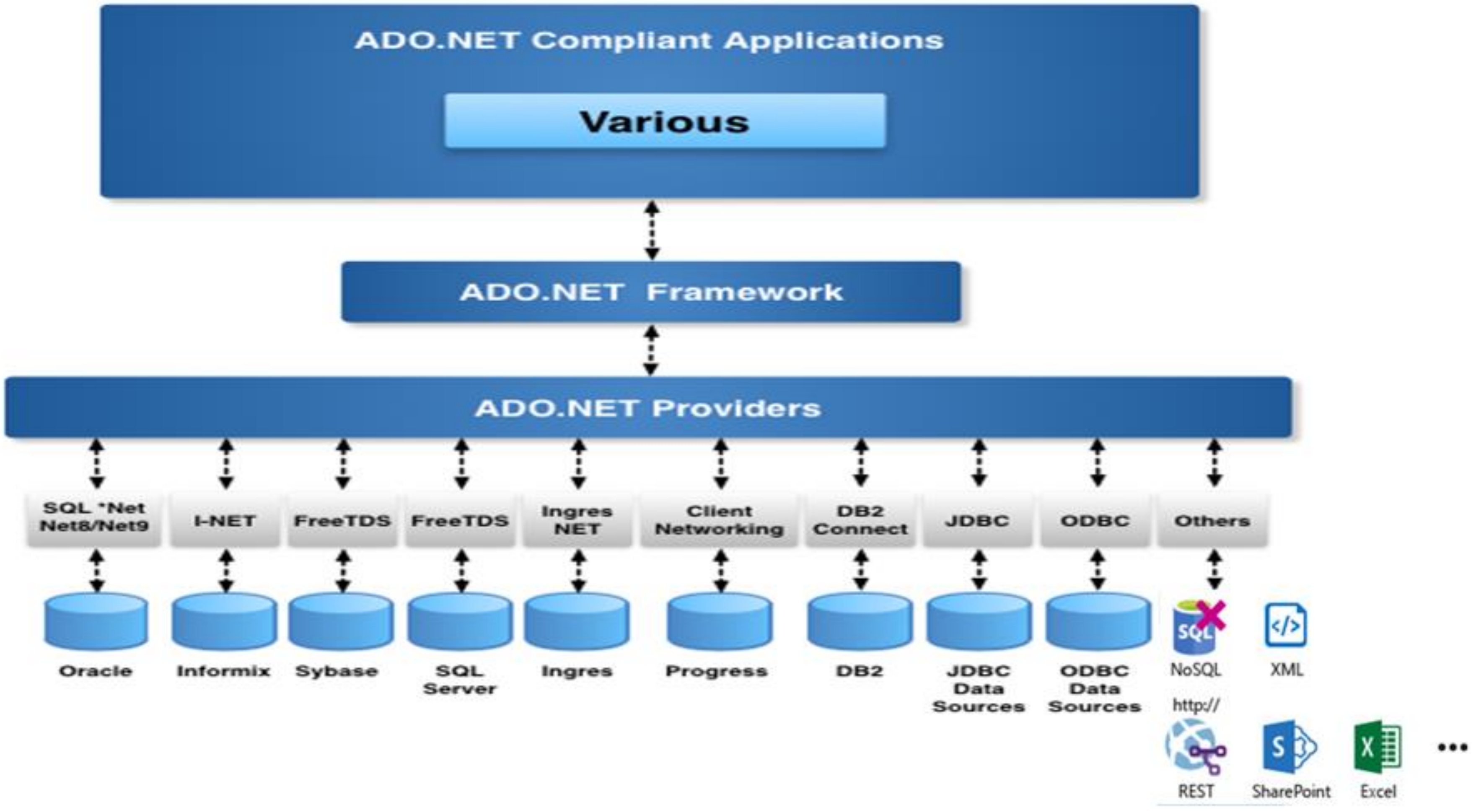
ADO.NET

- OLE DB and ADO were merged into **ADO.NET** (based on the .NET framework)
  - The .NET framework was developed by Microsoft with the objective to perform a modern overhaul of all core components that make up the Windows and related technologies stack.
- ADO.NET provides communication between relational and non-relational systems through a common set of components.

ADO.NET (cont.)

- ADO.NET is a set of computer software components that programmers can use to access data.
- ADO.NET consists of the <u>Common Language Runtime (CLR) </u>and <u>a set of class libraries.</u>
  - CLR provides an execution environment for .NET
    - Source codes are written in a programing language that support the CLR, such as C# or VB.Net
  - The .NET class libraries offer a hierarchy of libraries containing a plethora of generic, re-usable components

ADO.NET Environment



ADO.NET Provider

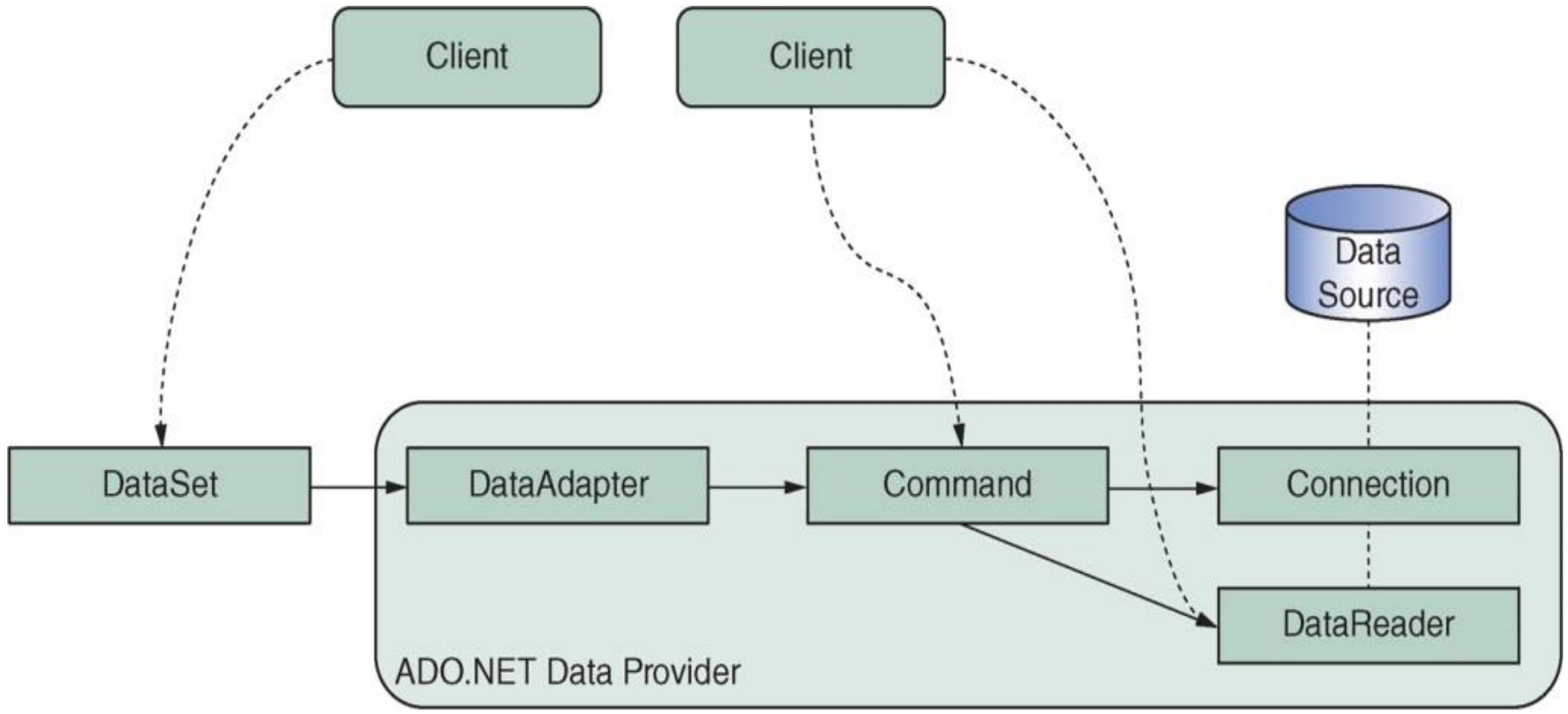
- A **ADO.NET provider** is a software component that interacts with a data source.
- ADO.NET providers can be created to access such simple data stores as a text file and spreadsheet, through to such complex databases as Oracle Database, Microsoft SQL Server, MySQL, PostgreSQL, SQLite, IBM DB2, Sybase ASE, and many others.
- ADO.NET providers are analogous to ODBC drivers, JDBC drivers, and OLE DB providers.
- ADO.NET offers a series of services, which are broken down into a series of objects handling creation of database connections, sending queries, and reading results.

Example of C# Program for ADO.NET

```
String connectionString = “Data Source=(local);Initial Catalog=example;”
SqlConnection conn = new SqlConnection(connectionString)
conn.Open();
String query1 = “select avg(num_pages) from books”;
String query2 = “select title, author from books where num_pages > 30”;
SqlCommand command1 = conn.CreateCommand();
SqlCommand command2 = conn.CreateCommand();
command1.CommandText = query1;
command2.CommandText = query2;
int average_pages = command1.ExecuteScalar();
SqlDataReader dataReader = command2.ExecuteReader();

String title;
String author;
while (dataReader.Read()) {
 title = dataReader.GetString(0);
 author = dataReader.GetString(1);
 Console.Writeln(title + “ by “ + author);
}
dataReader.Close();
conn.Close();
```

This code shows the **Connection**, **Command**, and **DataReader** objects in action using the .NET framework Data Provider for SQL Server

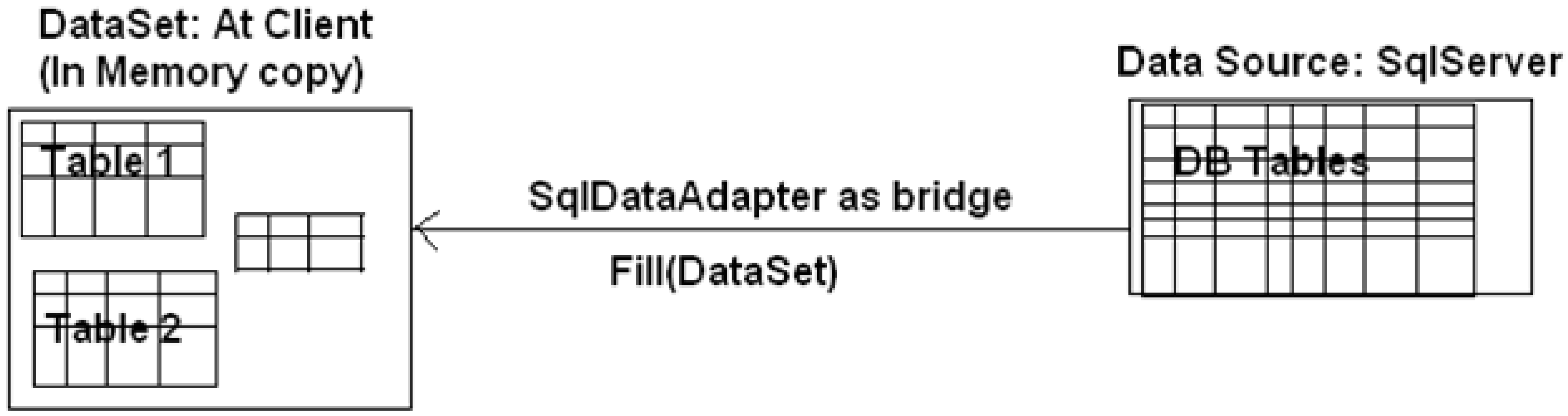
ADO.NET Classes



- We can create specialized subclasses of the **DataSet** classes for a particular database schema.
- The specialized subclasses allow convenient access to each field in the schema through strongly typed properties.

C# Program on ADO.NET-Alterative Approach

- A **DataAdapter** provides an alternative approach that can be used to retrieve data from a data source and store the results into a DataSet object.



  - **Dataset** represents a collection of data(tables) retrieved from the Data Source.

https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/populating-a-dataset-from-a-dataadapter
https://www.c-sharpcorner.com/uploadfile/61b832/sqldataadapter-fill-method/

C# Example I - Populating a DataSet from a DataAdapter

```
String connectionString = “Data Source=(local);Initial Catalog=example;”
SqlConnection conn = new SqlConnection(connectionString)
conn.Open();
String queryString = “select title, author from books”;
sqlDataAdapter ada = new SqlDataAdapter(queryString, conn);
// Create a DataSet object and fill it with query result from the books tale
DataSet booksDataSet = new DataSet();
ada.Fill(booksDataSet, “myBooks”); //The query result stores in myBooks table
in the DataSet named booksDataSet
DataTable tb1 = booksDataSet.Tables[“myBooks”]; //Fetch the DataTable from
Foreach (DataRow bookRow in tbl.Rows) {         //theDataSet and loop over it
   Console.WriteLine(bookRow[“title”]);
}
conn.Close();
```

- **Steps:**
  - Create a query string.
  - Create a connection object and open it.
  - Create a SqlDataAdapter object accompanying the query string and connection object.
  - Create a DataSet object.
  - Use the Fill method of SqlDataAdapter object to fill the DataSet with the result of query string.
  - Close the connection object

C# Example II- Populating a DataSet from Multiple DataAdapters

```
// Assumes that customerConnection is a valid SqlConnection object.
// Assumes that orderConnection is a valid OleDbConnection object.
SqlDataAdapter custAdapter = new SqlDataAdapter(
  "SELECT * FROM dbo.Customers", customerConnection);
OleDbDataAdapter ordAdapter = new OleDbDataAdapter(
  "SELECT * FROM Orders", orderConnection);

DataSet customerOrders = new DataSet();
custAdapter.Fill(customerOrders, "Customers");
ordAdapter.Fill(customerOrders, "Orders");

//The filled tables are related with a DataRelation, and the list of
//customers is then displayed with the orders for that customer.
DataRelation relation = customerOrders.Relations.Add("CustOrders",
  customerOrders.Tables["Customers"].Columns["CustomerID"],
  customerOrders.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow pRow in customerOrders.Tables["Customers"].Rows)
{
   Console.WriteLine(pRow["CustomerID"]);
   foreach (DataRow cRow in pRow.GetChildRows(relation))
      Console.WriteLine("\t" + cRow["OrderID"]);
}
```

https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/populating-a-dataset-from-a-dataadapter

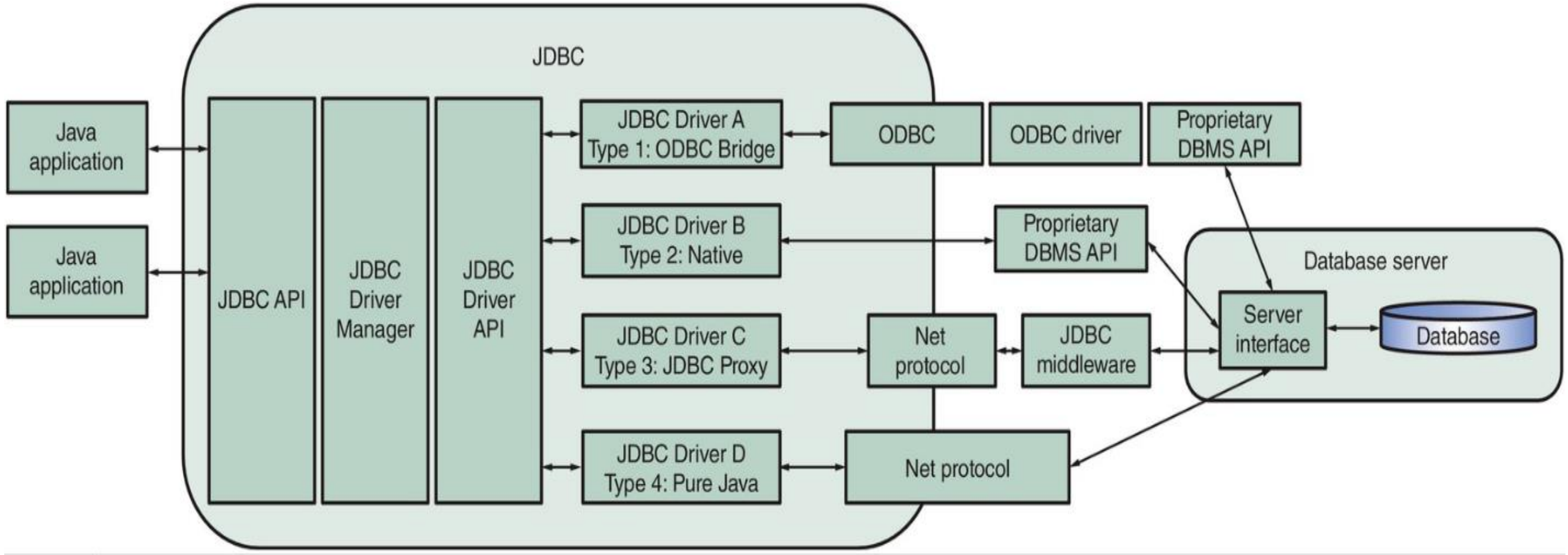
Outline

- Introduction
- Database API Types
- Embedded API
- Call-level API (Universal API)
  - ODBC
  - OLE DB
  - ADO.NET
  - ☞ **JDBC**
  - Language-integrated Querying
  - Object Persistence and ORMS

JDBC

- Like ODBC, **Java DataBase Connectivity (JDBC)** offers a call-level database API
  - inspired by ODBC but developed to be used in Java
  - high portability and the ability to program in an OO way
  - database connections, drivers, queries and results are thus all expressed as objects, based on uniform interfaces and hence exposing a uniform set of methods, no matter which DBMS is utilized

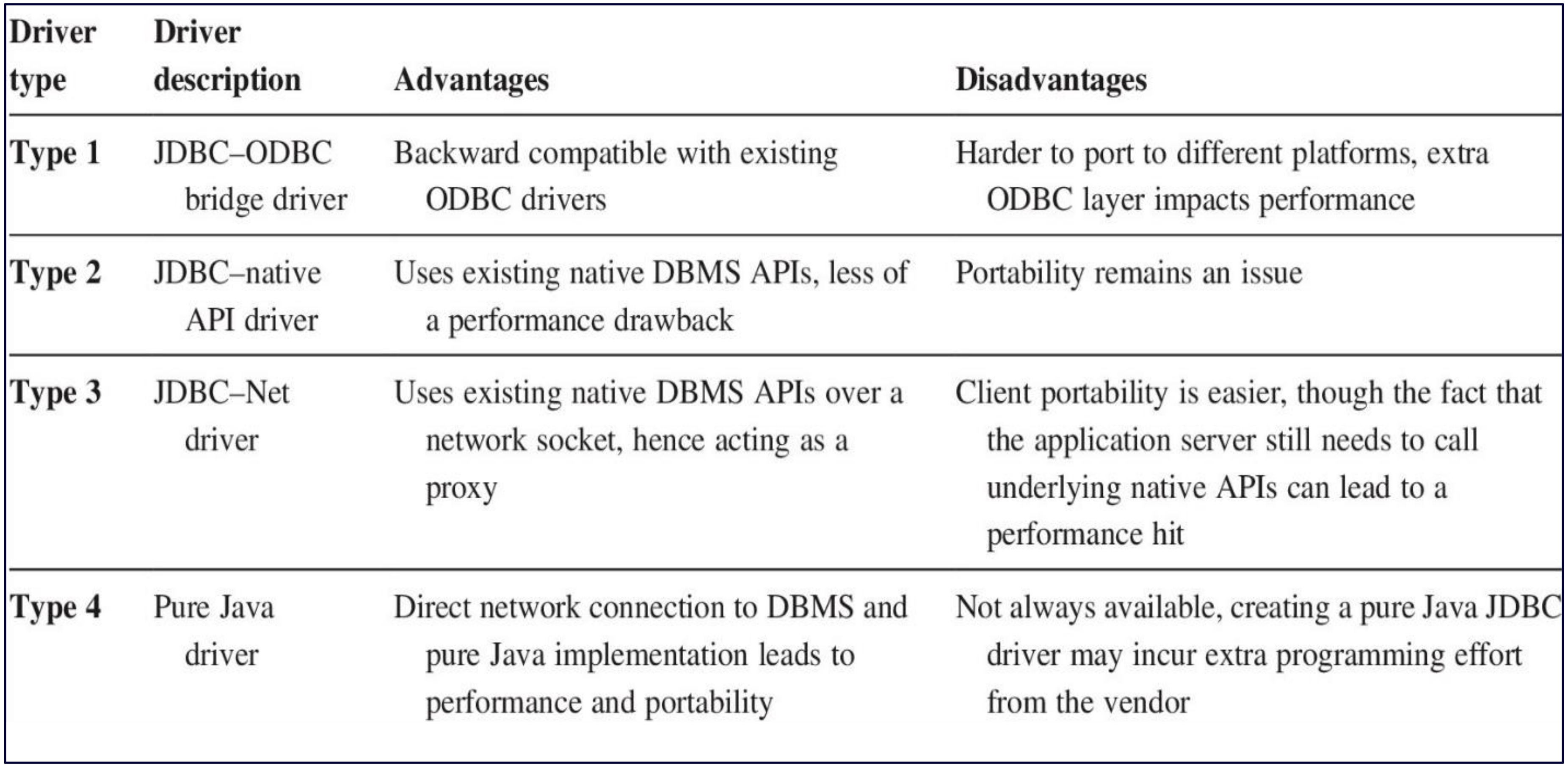
JDBC Architecture



JDBC Architecture

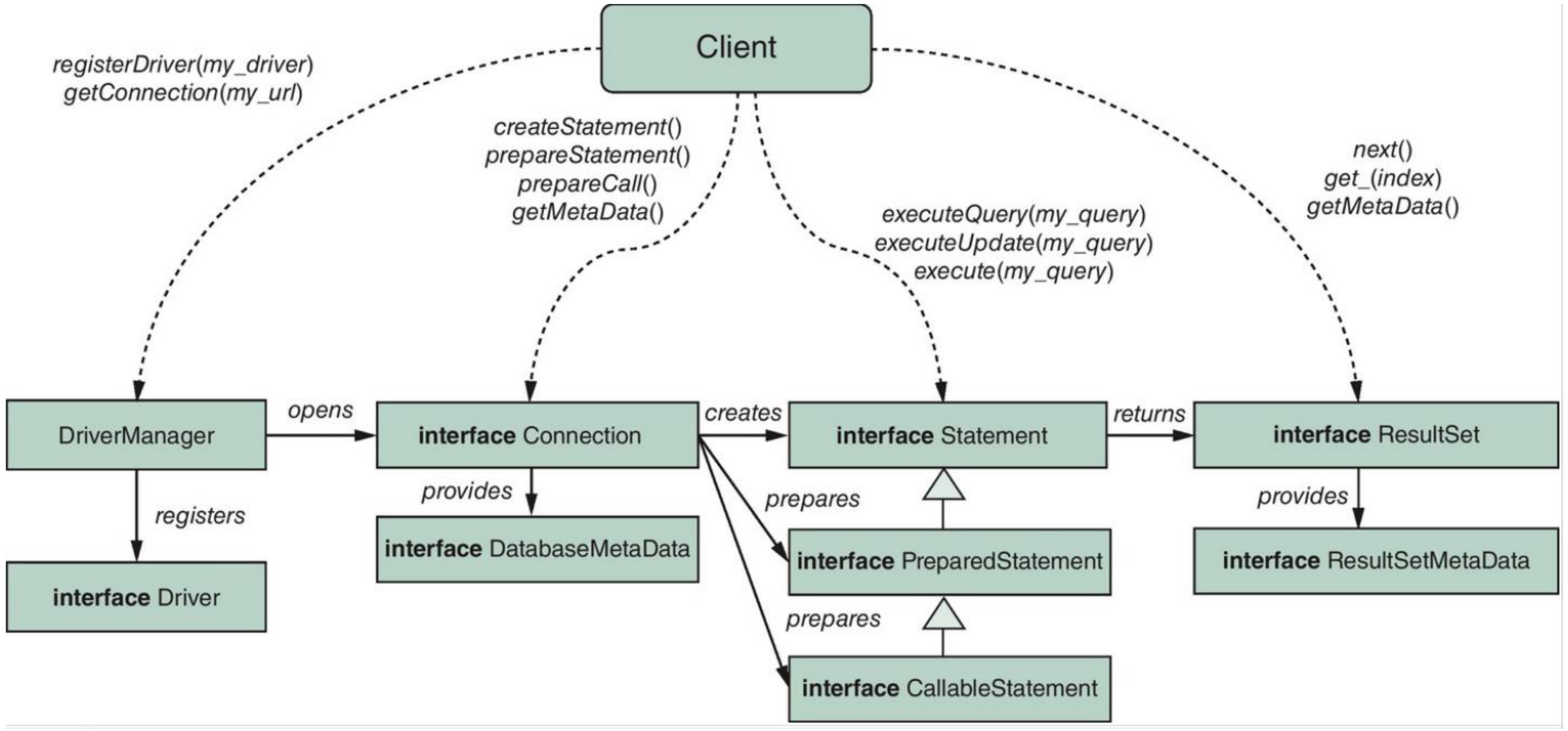
- The DriverManger acts as the basic service to mange JDBC drivers.
- The driver objects registered with the DriverManager implement the Driver interface and enable the communication between the DriverManager and the DBMS.
- Driver Types
  - Tpye-1 drivers (JDBC-ODBC bridge drivers)
  - Type-2 drivers (JDBC-native API drivers)
  - Type-3 drivers (JDBC-Net Drivers)
  - Type-4 drivers

Different JDBC Driver Types



Driver type	Driver description	Advantages	Disadvantages
Type 1	JDBC–ODBC bridge driver	Backward compatible with existing ODBC drivers	Harder to port to different platforms, extra ODBC layer impacts performance
Type 2	JDBC–native API driver	Uses existing native DBMS APIs, less of a performance drawback	Portability remains an issue
Type 3	JDBC–Net driver	Uses existing native DBMS APIs over a network socket, hence acting as a proxy	Client portability is easier, though the fact that the application server still needs to call underlying native APIs can lead to a performance hit
Type 4	Pure Java driver	Direct network connection to DBMS and pure Java implementation leads to performance and portability	Not always available, creating a pure Java JDBC driver may incur extra programming effort from the vendor

Overview of JDBC's Classes



JDBC Classes and Interfaces

- JDBC is a collection of Java classes and interfaces that enables database access from programs written in the Java language.
  - Methods for connecting to a remote data source, executing SQL statements, etc.
  - A part of the java.sql package

```
import java.sql.*
```

JDBC Driver Management

- All drivers are managed by the DriverManager class which maintains a list of all currently loaded drivers
  - Methods : registerDriver, deregisterDriver, getDrivers to enable dynamic addition and deletion of drivers
- Loading a JDBC driver:
  - In the Java code: `Class.forName("oracle.jdbc.OracleDriver");`
  - Another way to load
    - When starting the Java application, we can include the driver with `-Djdbc.drivers=oracle.dbc.driver`
- A recent version of a vendor specific JDBC (e.g., Oracle thing JDBC driver) may not require loading JDBC driver explicitly.

Opening a Connection to a Database

- We interact with a data source through sessions. Each connection identifies a logical session.
- A section with a data source is started through creation of a Connection object.
- Connections are specified through a JDBC URL (a URL that uses the jdbc URL)

  JDBC URL form is jdbc:<subprotocol>:<otherParameters>

Example:

```
String url="jdbc:oracle:thin";
Connection  conn;
try{
    conn = DriverManager.getConnection(url:usedId:password);
} catch SQLException excpt { …}
```

// Assume that userId and password variables are set to valid values.

Three Ways for Preparing a SQL Statement

- Opening a connection returns a *Connection* object, representing a session to a specific database. To create SQL statement,
- **Statement** object

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

  - The base class for the other two statement classes
  - Allows to query the data source <u>with any static generated SQL query</u>
- **PreparedStatement** object

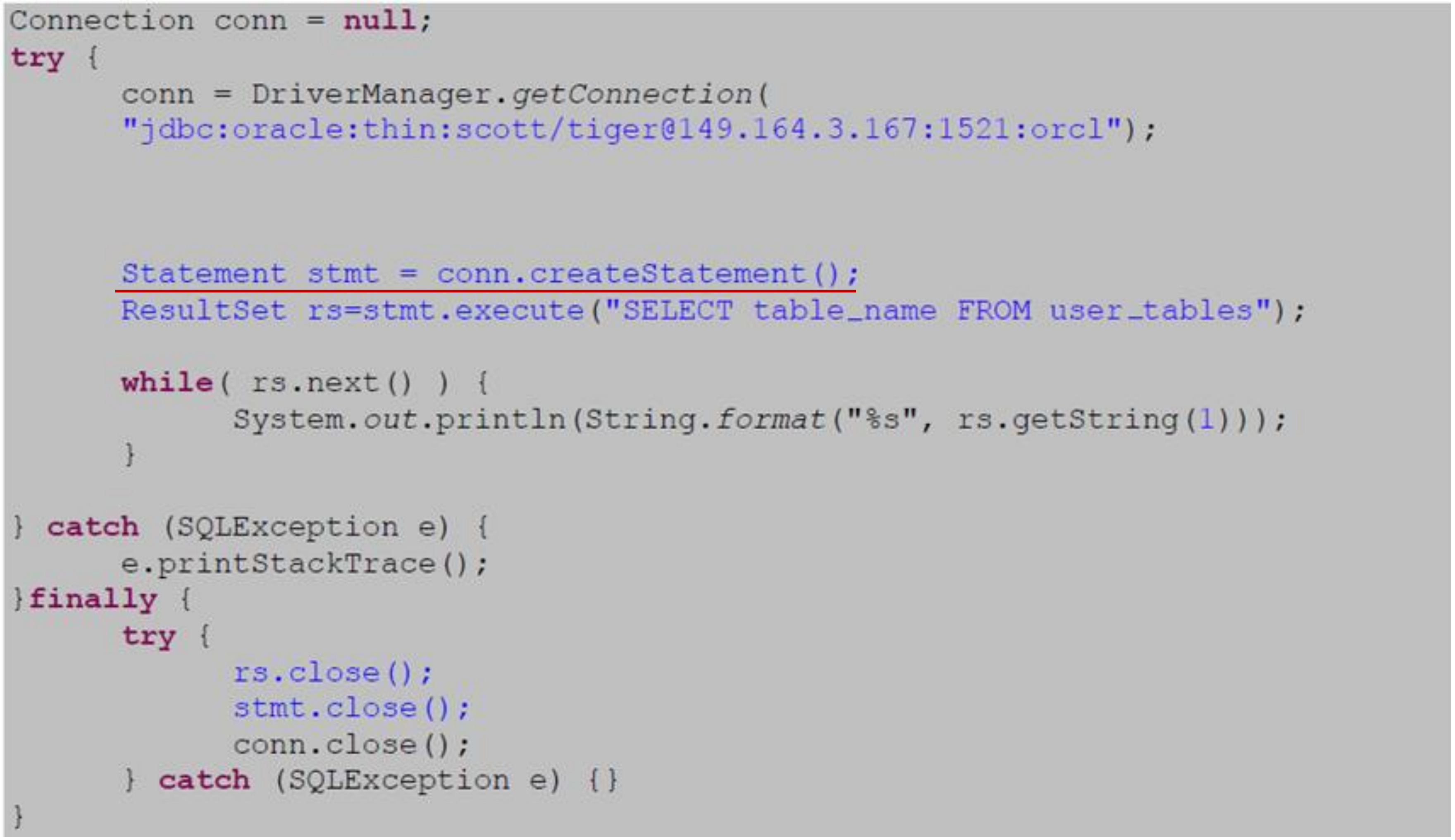
  ```
  PreparedStatement stmt = conn.preparedStatement(…);
  ```

  - Dynamically generates precomiled SQL statements
  - PreparedStatement object enables to run <u>a statement with varying sets of input parameters.</u>
  - Values of parameters are determined at run-time
- **CallableStatment** object

  ```
  CallableStatement stmt = conn.prepareCall(…);
  ```

  - A way <u>to call stored procedures</u>

Example



```
Connection conn = null;
try {
      conn = DriverManager.getConnection(
      "jdbc:oracle:thin:scott/tiger@149.164.3.167:1521:orcl");



      Statement stmt = conn.createStatement();
      ResultSet rs=stmt.execute("SELECT table_name FROM user_tables");

      while( rs.next() ) {
            System.out.println(String.format("%s", rs.getString(1)));
      }

} catch (SQLException e) {
      e.printStackTrace();
}finally {
      try {
            rs.close();
            stmt.close();
            conn.close();
      } catch (SQLException e) {}
}
```

Methods to Execute a Statement Object

Depends on the type of SQL statement being executed

- If the `Statement` object represents a SQL query returning a `ResultSet` object, the **`executeQuery`** method should be used.
- If the SQL is known to be a DDL statement or a DML statement returning an update count, the **`executeUpdate`** method should be used.
- If the type of the SQL statement is not known, the **`execute`** method should be used.

Running a Query and Retrieving a ResultSet Object

- The **`executeQuery`** method takes a SQL statement as input and returns a **`ResultSet`** object.

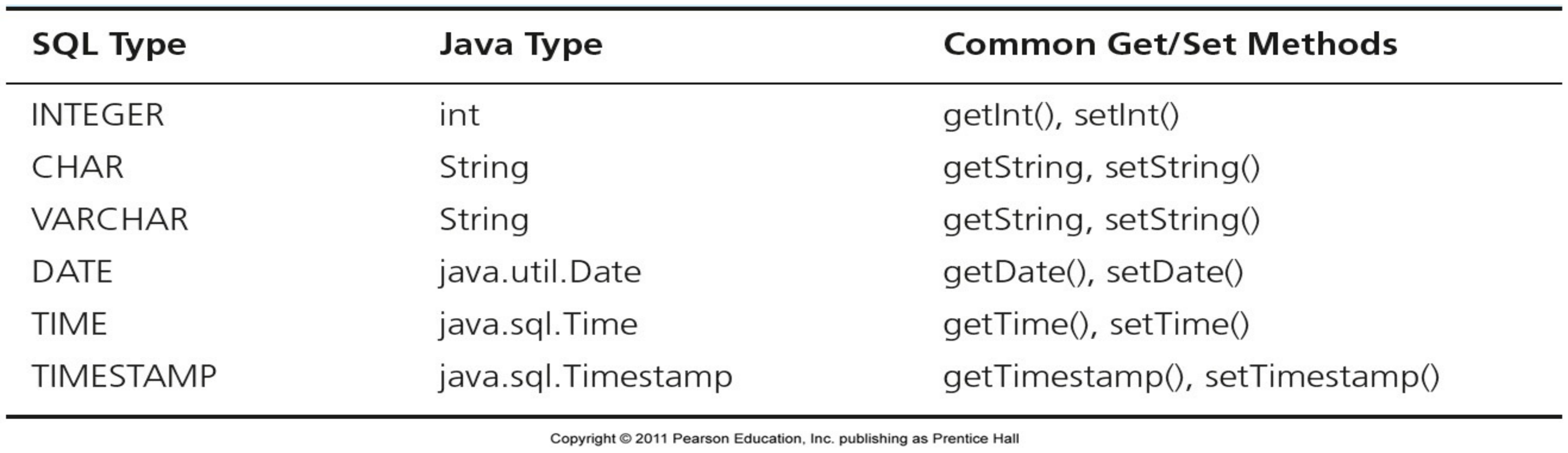
```
String sql="SELECT ENAME FROM EMPLOYEE";
Statement stmt=conn.createStatement(sql);
stmt.executeQuery(sql);
ResultSet rs=stmt.getResultSet();
```

- Because SQL is set-oriented, the query result will generally comprise multiple tuples. Host languages such as Java are record-oriented cannot handle more than one tuple at a time
- For processing the ResultSet object, use a cursor mechanism in order to loop through result sets

```
While ( rs.next() ) {
     System.out.println( rs.getString(1));
}
```

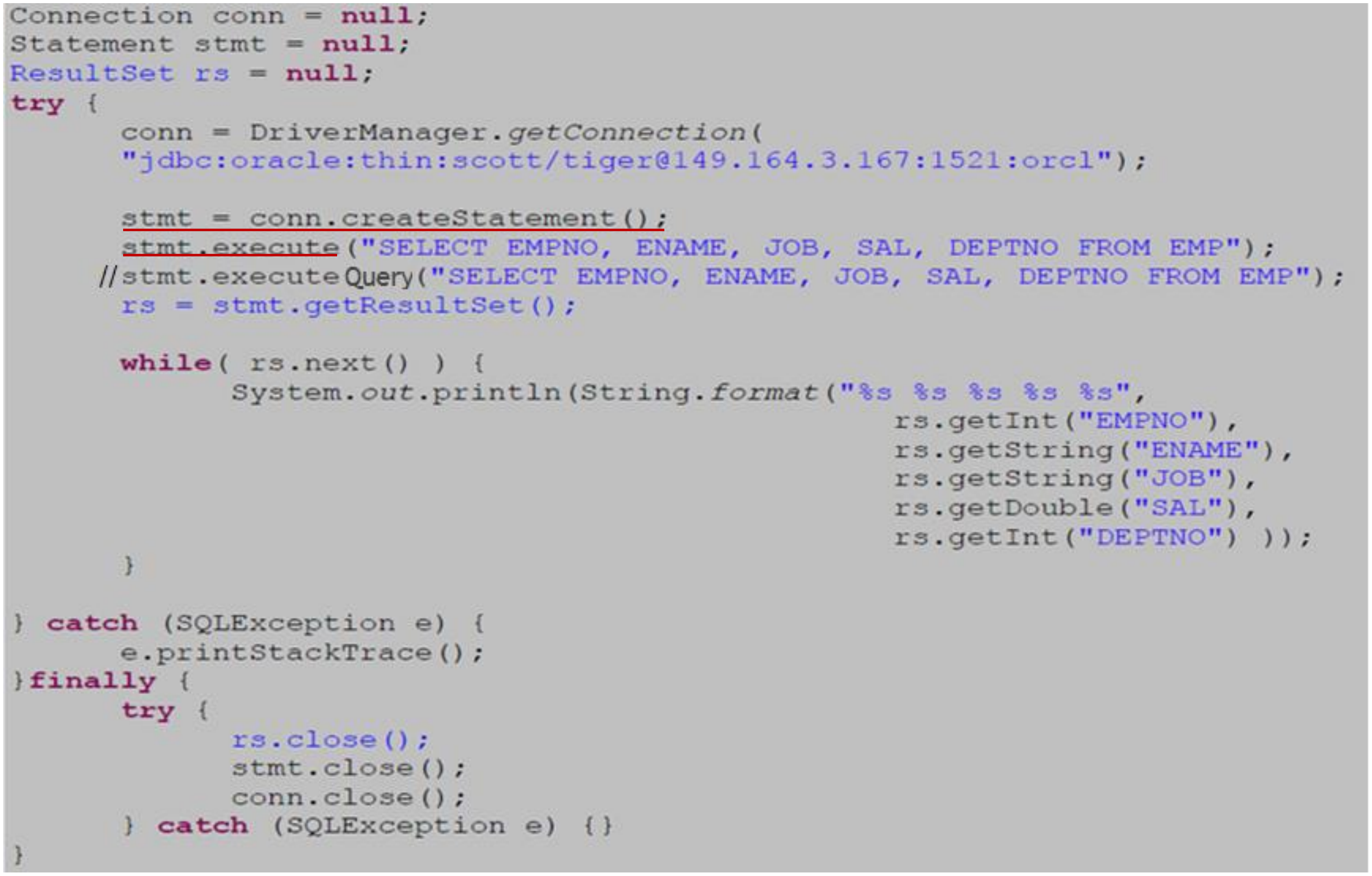
the first field of the result row

Common Java-to-SQL Mappings



SQL Type	Java Type	Common Get/Set Methods
INTEGER	int	getInt(), setInt()
CHAR	String	getString, setString()
VARCHAR	String	getString, setString()
DATE	java.util.Date	getDate(), setDate()
TIME	java.sql.Time	getTime(), setTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp(), setTimestamp()

Processing a Static SQL Query Statement



```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
      conn = DriverManager.getConnection(
      "jdbc:oracle:thin:scott/tiger@149.164.3.167:1521:orcl");

      stmt = conn.createStatement();
      stmt.execute("SELECT EMPNO, ENAME, JOB, SAL, DEPTNO FROM EMP");
    //stmt.executeQuery("SELECT EMPNO, ENAME, JOB, SAL, DEPTNO FROM EMP");
      rs = stmt.getResultSet();

      while( rs.next() ) {
            System.out.println(String.format("%s %s %s %s %s",
                                                rs.getInt("EMPNO"),
                                                rs.getString("ENAME"),
                                                rs.getString("JOB"),
                                                rs.getDouble("SAL"),
                                                rs.getInt("DEPTNO") ));
      }

} catch (SQLException e) {
      e.printStackTrace();
}finally {
      try {
            rs.close();
            stmt.close();
            conn.close();
      } catch (SQLException e) {}
}
```

Closing the Result Set and Statement Objects

- You must explicitly close the ResultSet and Statement objects after you finish using them. The JDBC drivers do not have finalizer methods. The cleanup routines are performed by the `close` method of the ResultSet and Statement classes.
  - If you do not explicitly close the ResultSet andStatement objects, serious memory leaks could occur.
- For example, if your ResultSet object is rs and your Statement object is stmt, then close the result set and statement with the following lines of code:

```
rs.close();
stmt.close();
```

Processing a SQL Query with Input Parameters

- **`PreparedStatement`** object enables to run a statement with varying sets of input parameters.
- Use **`set`**XXX methods on the PreparedStatement object to bind data to variable parameters

```
Connection conn = null;
PreparedStatement stmt = null;
ResultSet rs = null;
try {
      conn = DriverManager.getConnection(
                  "jdbc:oracle:thin:scott/tiger@149.164.3.167:1521:orcl");

      stmt = conn.prepareStatement("SELECT EMPNO, ENAME, JOB, SAL, DEPTNO
                                          FROM EMP WHERE EMPNO = ?");

      stmt.setInt(1, EmpNo);
                    the first parameter
      rs = stmt.executeQuery();

}catch( SQLException e1 ) {
      e1.printStackTrace();
}finally {
      try {
            stmt.close();
            conn.close();
            rs.close();
      }catch( SQLException ex ) {}
}
```

Making Changes to the Database

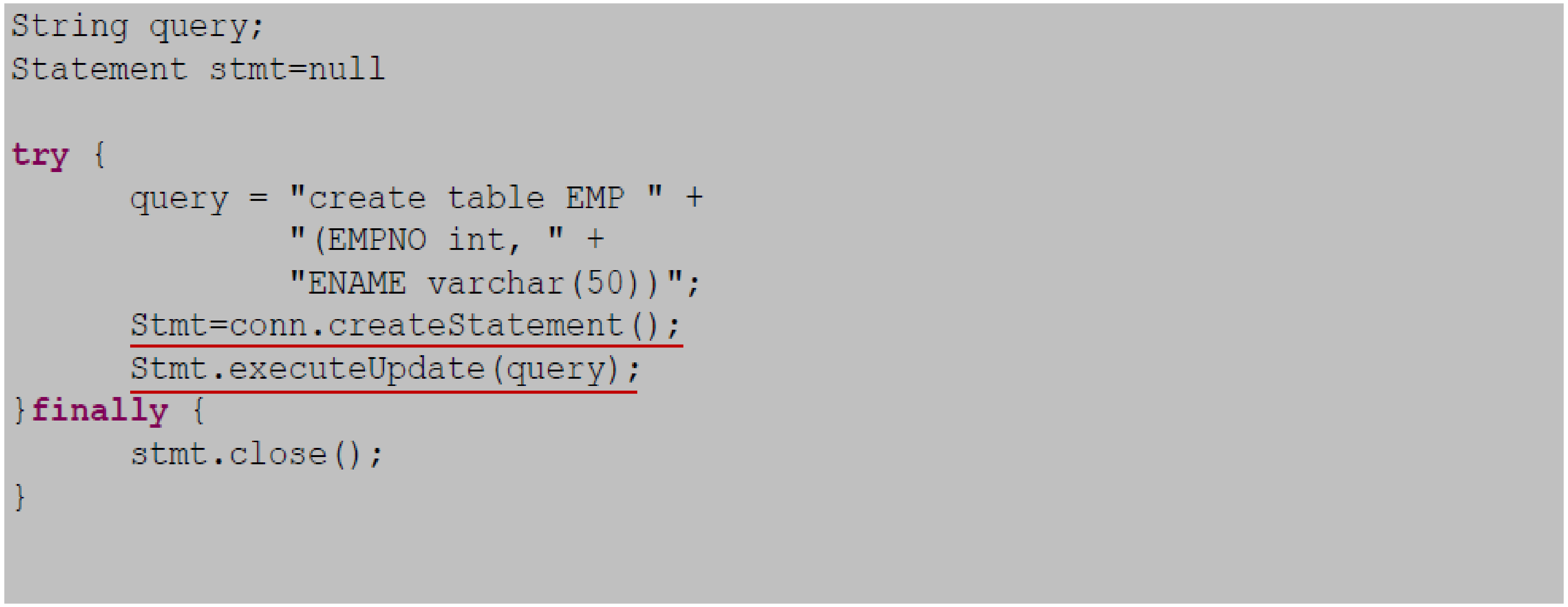
- To perform DML (Data Manipulation Language) operations, such as INSERT, UPDATE or DELETE operations, create either a **`Statement`** or a **`PreparedStatement`** object.

Example for Executing INSERT

```
Connection conn = null;
PreparedStatement stmt = null;
int NumRowsAffected = 0;
try {
      conn = DriverManager.getConnection(
                 "jdbc:oracle:thin:scott/tiger@//192.168.0.193:1521/ORCL");

      stmt = conn.prepareStatement(
       "INSERT INTO EMP( EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO)
                                  VALUES( ?, ?, ?, ?, ?, ?, ?, ? )");

      stmt.setInt(1, e.getEmpNo());
      stmt.setString(2, e.getEmpName());
      stmt.setString(3, e.getJob());
      stmt.setInt(4, e.getEmpManagerNo());
      stmt.setDate(5, e.getHireDate());
      stmt.setDouble(6, e.getSalary());
      stmt.setInt(7, e.getComm());
      stmt.setInt(8, e.getDeptNo());

      NumRowsAffected = stmt.executeUpdate();
      conn.commit();

}catch( SQLException e1 ) {
      e1.printStackTrace();
}finally {
      try {
            stmt.close();
            conn.close();
      }catch( SQLException ex ) {}
}
```

DDL Operations

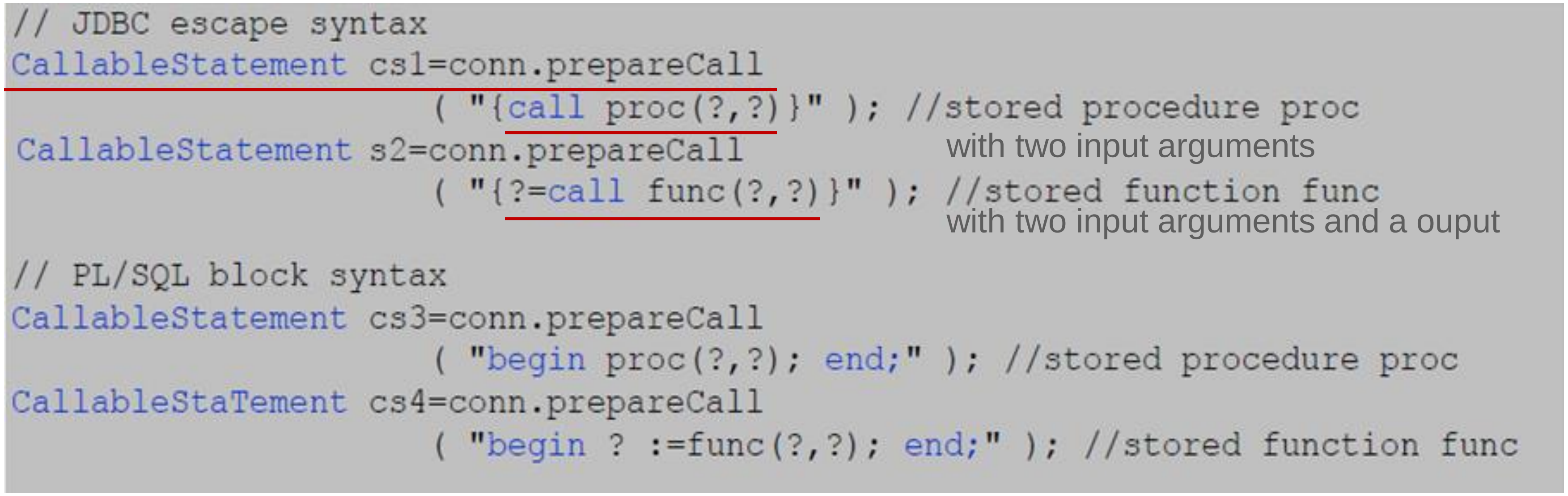
- To perform data definition language (DDL) operations, create either a Statement object or a PreparedStatement object.



```
String query;
Statement stmt=null

try {
      query = "create table EMP " +
              "(EMPNO int, " +
              "ENAME varchar(50))";
      Stmt=conn.createStatement();
      Stmt.executeUpdate(query);
}finally {
      stmt.close();
}
```

Stored Procedure Calls in JDBC Programs

- Oracle JDBC drivers support the processing of PL/SQL stored procedures and anonymous blocks. They support PL/SQL block syntax and most of JDBC escape syntax.



```
// JDBC escape syntax
CallableStatement cs1=conn.prepareCall
                    ( "{call proc(?,?)}" ); //stored procedure proc
                                              with two input arguments
CallableStatement s2=conn.prepareCall
                    ( "{?=call func(?,?)}" ); //stored function func
                                              with two input arguments and a ouput

// PL/SQL block syntax
CallableStatement cs3=conn.prepareCall
                    ( "begin proc(?,?); end;" ); //stored procedure proc
CallableStaTement cs4=conn.prepareCall
                    ( "begin ? :=func(?,?); end;" ); //stored function func
```

Recap: Steps in JDBC Database Access

1. Import JDBC library `(java.sql.*)`
2. Load JDBC driver: `Class.forname("oracle.jdbc.OracleDriver")`
3. Define appropriate variables
4. Create a connect object (via `getConnection`)
5. Create a statement object of one of Statement classs: `Statement (or PreparedStatement or CallableStatement)`
6. Identify statement parameters (designated by question marks)
7. Bound parameters to program variables
8. Execute SQL statement (referenced by an object) via JDBC's `execute (or executeQuery or executeUpdate)`
9. Process query results (returned in an object of type `ResultSet`).
10. Close objects you create using `close()`

Outline

- Introduction
- Database API Types
- Embedded API
- Call-level API (and Universal API)
  - ODBC
  - OLE DB
  - ADO.NET
  - JDBC
  - ☞ **Language-integrated Querying**
  - Object Persistence and ORMS

Language-integrated Querying

- Key JDBC drawback is the lack of compile-time type checking and validation (e.g., no syntactic and semantic SQL checks)
- Modern programming languages use language-native query expressions into their syntax which are often able to operate on any collection of data (e.g., a database, XML documents)
  - when targeting a DBMS, these expressions are converted to SQL, which can then be sent off to the target DBMS using JDBC or another API

Example: JOOQ (Java Object Oriented Querying)

- jOOQ provides the benefits of embedded SQL using pure Java, rather than resorting to an additional pre-compiler
- A code generator is run first that inspects the database schema and reverse-engineers it into a set of Java classes representing tables, records, and other schema entities
- These can then be queried and invoked using plain Java code

```
String sql = create.select(BOOK.TITLE, AUTHOR.NAME)
          .from(BOOK)
          .join(AUTHOR)
          .on(BOOK.AUTHOR_ID.equal(AUTHOR.ID))
          .where(BOOK.NUM_PAGES.greaterThan(100))
          .getSQL();
```

Language-integrated Querying

- Since now only pure Java code is used to express statements, IDEs do not need to be aware of a separate language, no pre-compiler is necessary, and the standard Java compiler can be used to perform type safety checks and generate compilation errors when necessary
- Other examples
  - QueryDSL
  - Microsoft’s LINQ (Language Integrated Query)

Object Persistence and ORMs (Object Relational Mapping)

- API technologies such as JDBC and ADO.NET represent database related entities (e.g. tables, records) in an OO way
- **Object persistence** aims to represent domain entities, such as Book and Author, as plain objects using the used programming language representational capabilities and syntax
  - These objects can then be persisted behind the scenes to a database or other data source

Object Persistence and ORMs

- **Language-integrated query technologies** apply similar ideas
- Object persistence APIs go a step further, as they also describe the full business domain (i.e., the definition of data entities) within the host language
  - <u>to allow for efficient querying of objects</u>, such entities are frequently mapped to a relational database model <u>using a so-called Object Relational Mapper (ORM)</u>
  - not strictly necessary to utilize an ORM to enable object persistence, though most APIs tightly couple both concepts

Object Persistence and ORM APIs

- Object persistence with EJB (Enterprise JavaBeans)
- Object persistence with JPA (Java Persistent API, EJB 3.0)
- Object persistence with JDO (Java Data Objects)
- Object persistence with other host languages such as Ruby and Python

Appendix: Database API Summary

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
ODBC	Call-level	Late binding, though prepared SQL statements possible as well as calling stored procedures	A resultset with rows of fields	Mainly relational databases, though other structured tabular sources possible as well	Microsoft-based technology, not object-oriented, mostly outdated

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
JDBC	Call-level	Late binding, though prepared SQL statements possible as well as calling stored procedures	A resultset with rows of fields	Mainly relational databases, though other structured tabular sources possible as well	Java-based technology, portable, still in wide use

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
SQLJ	Embedded	Early binding	A resultset with rows of fields	Relational databases supporting SQL	Java-based technology, uses a precompiler, mostly outdated

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
OLE DB and **ADO**	Call-level	Late binding, though prepared SQL statements possible as well as calling stored procedures	A resultset with rows of fields	Mainly relational databases, though other structured tabular sources possible as well	Microsoft-based technology, backwards compatible with ODBC, mostly outdated

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
ADO.NET	Call-level	Late binding, though prepared SQL statements possible as well as calling stored procedures	A resultset with rows of fields provided by a DataReader, or a DataSet: a collection of tables, rows, and fields, retrieved and stored by DataAdapters	Various data sources	Microsoft-based technology, backwards compatible with ODBC and OLE DB

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
Language-integrated Query Technologies	Uses an underlying call-level API	Uses an underlying late-binding API	A resultset with rows of fields, sometimes converted to a plain collection of objects representing entities	Relational databases supporting SQL or other data sources	Examples: **jOOQ** (Java Object Oriented Querying) and LINQ, works together with another API to convert expressions to SQL

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
Enterprise JavaBeans (EJB 2.0)	Uses an underlying call-level API	Uses an underlying late-binding API	Plain Java entity Beans as the main representation	Mainly relational databases, though other structured tabular sources possible as well	Java-based technology. EJB Query Language (**EJB-QL**) works together with another API to convert expressions to SQL

Other Database API Technology

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
Java Persistence API (JPA in EJB 3.0)	Uses an underlying call-level API	Uses an underlying late-binding API	Plain Java objects as the main representation	Mainly relational databases, though other structured tabular sources possible as well	Java-based technology. **JPA** works together with another API to convert expressions to SQL

Other Database API Technology

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
Java Database Objects (JDO) API	Uses an underlying call-level API	Uses an underlying late-binding API	Plain Java objects as the main representation	Various data sources	Java-based technology. JDO provides its own object-based query language (JDOQL), designed to have the power of SQL queries

Other Database API Technology

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
ORM (Object-Relational Mapping) APIs - ActiveRecord, Entity Framework, SQL Alchemy	Uses an underlying call-level API	Uses an underlying late-binding API	Plain objects defined in the programming language as the main representation	Relational databases	Various implementations available for each programming language