Database APIs

Contents

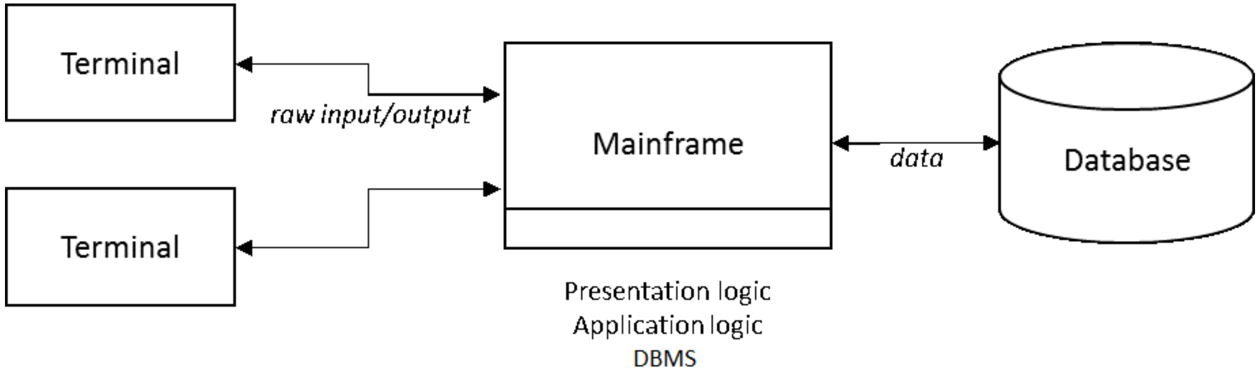
- Database System Architectures
- Database APIs
- Object Persistence and Object Relational Mappers
- Exercise

Contents

- **Database System Architectures**
- Database APIs
- Object Persistence and Object Relational Mappers
- Exercises

Database System Architectures

- Centralized System Architecture
  - all responsibilities of the DBMSs are handled by one centralized entity (e.g. mainframe database)
  - has become rare, expensive, and difficult to maintain



Database System Architectures

- Tiered System Architectures
  - aim to decouple the centralized setup by combining the computing capabilities of powerful central computers with the flexibility of PCs
- Multiple variants of this architecture
  - E.g., two-tier architecture (a.k.a. client-server architecture)

Database System Architectures

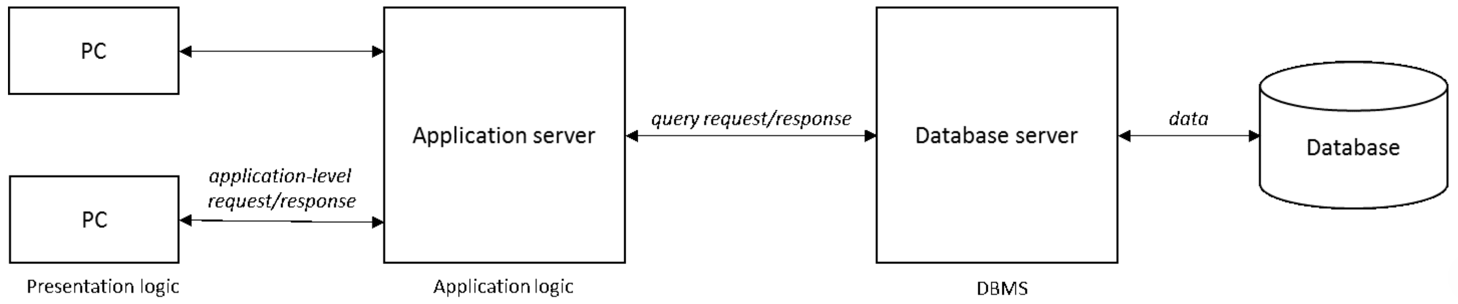
- "Fat" client variant
  - presentation logic and application logic are handled by the client
  - common in cases where it makes sense to couple an application's workflow with its look-and-feel
  - DBMS now fully runs on the database server

Database System Architectures

- “Thin” client variant
  - only the presentation logic is handled by the client
  - applications and database commands executed on the server
  - common when application logic and database logic are tightly coupled or similar

Database System Architectures

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



Contents

- Database System Architectures
- **Database APIs**
- Object Persistence and Object Relational Mappers
- Exercise

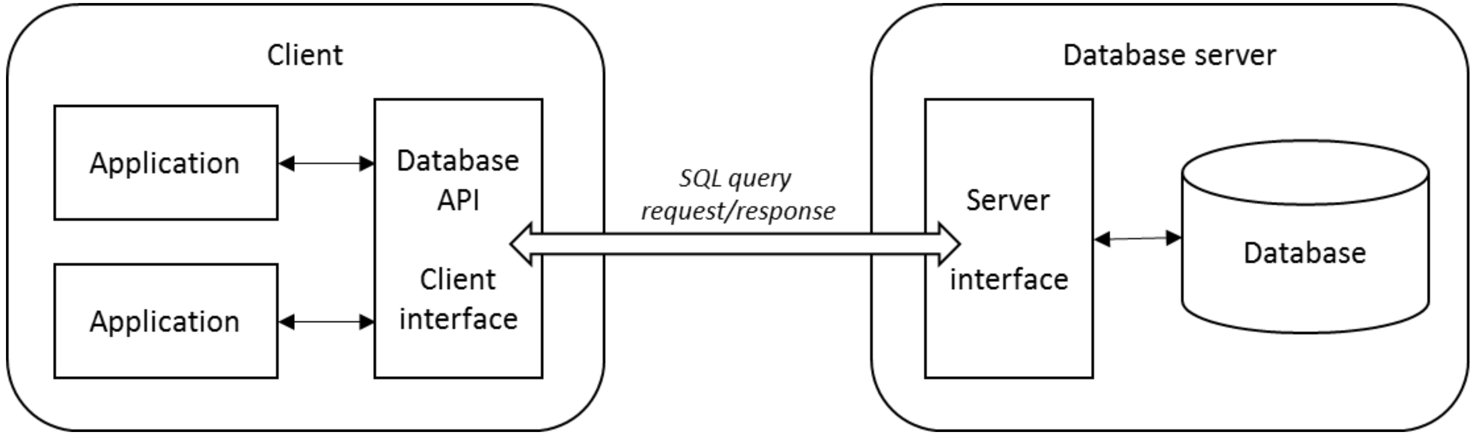
Database APIs

- In a tiered DBMS system architecture, client applications are able to query database servers and receive the results
- Client applications that wish to utilize the services provided by a DBMS use a specific API provided by the DBMS
- This database API exposes an interface through which client applications can access and query a DBMS

Database APIs

- Database server receives calls made by clients and executes the relevant operations before returning the results
- In many cases, the client- and server-interfaces are implemented to work over a computer network using network sockets
- The main goal of database APIs is to expose an interface through which other parties can utilize the services provided by the DBMS

Database APIs



Proprietary Versus Universal APIs

- Most DBMS vendors provide a proprietary, DBMS-specific API
  - disadvantage is that client applications must be aware of the DBMS that will be utilized on the server-side
- Alternatively, generic, vendor-agnostic universal APIs have been proposed,
  - allow to easily port applications to multiple DBMSs

Embedded Versus Call-level APIs

- APIs can be embedded or call-level
- Embedded API embeds SQL statements in the host programming language, meaning that SQL statement(s) will be part of the source code
  - 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
  - converted source code is then sent to the actual compiler

Embedded Versus Call-level APIs

- Advantages of embedded APIs
  - pre-compiler can perform specific syntax checks
  - pre-compiler can also perform an early binding step which helps to generate an efficient query plan before the program is run, hence improving performance (see chapter about performance)
- Disadvantages of embedded APIs
  - Harder to maintain code
- Embedded database APIs not very popular
- Example: SQLJ

Embedded Versus Call-level APIs

- Call-level APIs work 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
- Example: ODBC, Python PyODBC….

Early Binding Versus 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 field names, checking whether the user or client has sufficient access rights, and generating a query plan to access the physical data in the most performant way possible.
- Early versus late binding then refers to the actual moment when this binding step is performed

Early Binding Versus Late Binding

- Early binding is possible in case a pre-compiler is used and can hence only be applied with an embedded API
  - beneficial in terms of performance
  - binding only needs to be performed once
  - pre-compiler can perform specific syntax checks

Early Binding Versus Late Binding

- Late binding performs the binding of SQL-statements at runtime
  – additional flexibility is offered (“dynamic SQL”)
  – syntax errors or authorisation issues will remain hidden until the program is executed
  – testing the application can be harder
  – less efficient for queries that must be executed multiple times because the execution plan can’t be cached

Early Binding Versus Late Binding

- For embedded APIs, the involvement of a pre-compiler implies early binding
- For call-level APIs, late-binding will be used
  - it is possible even when using call-level APIs to pre-compile SQL statements and call these at runtime, by defining such statements as “stored procedures” in the DBMS

Early Binding Versus Late Binding

	Embedded APIs	Call-level APIs
Early binding (**“static” SQL**)	**Possible if 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**
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

Universal Database APIs

- Many different universal API standards have been proposed over the years, which differ in terms of
  - embedded or call-level
  - programming language(s)
  - functionalities

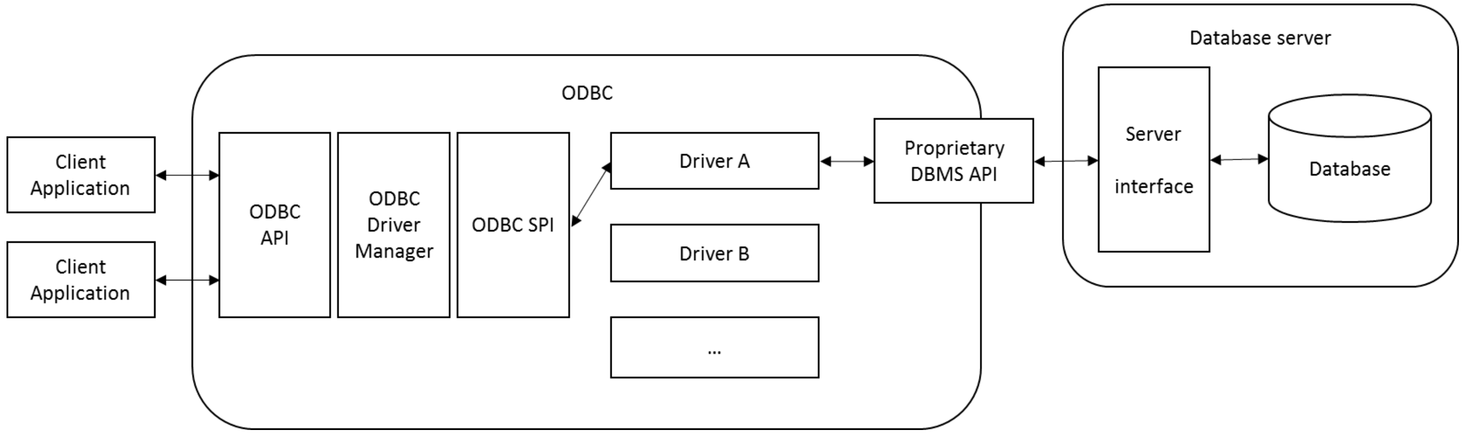
ODBC

- Open DataBase Connectivity (ODBC)
  - 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 (SPI): separate interface implemented by the DBMS vendor by which the Driver Manager interacts with various drivers

ODBC


Client Application
Client Application
ODBC
ODBC API
ODBC Driver Manager
ODBC SPI
Driver A
Driver B
...
Proprietary DBMS API
Database server
Server interface
Database

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 ↓)

OLE DB and ADO

- OLE DB (Object Linking and Embedding for DataBases) was a follow-up specification to ODBC to allow uniform access to a variety of data sources using Microsoft’s Component Object Model (COM)
- OLE DB also supports object databases, spreadsheets and other data sources
- Functionality such as querying can be provided by the data provider, but also by other components
- As such, OLE DB represents Microsoft’s attempt to move towards a “Universal Data Access” approach

OLE DB and ADO

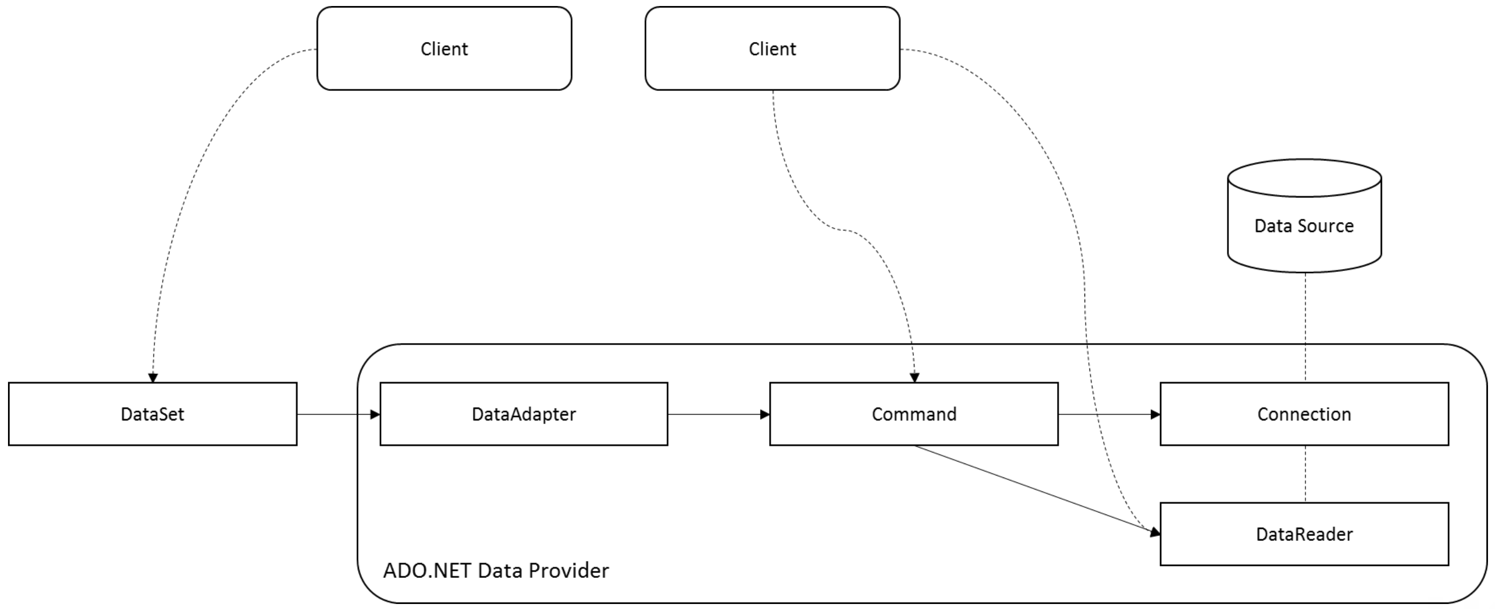
- OLE DB can be combined with ActiveX Data Objects (ADO), which provides a richer, more ‘programmer-friendly’ programming model on top of OLE DB

```
Dim conn As ADODB.Connection
Dim recordSet As ADODB.Recordset

Set conn = New ADODB.Connection
conn.Open(“my_database”)

Set qry = “select nr, name from suppliers where status < 30”
Set recordSet = cnn.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
conn.Close
```

ADO.NET

- OLE DB and ADO were merged into ADO.NET (based on the .NET framework)
- Like OLE DB, ADO.NET breaks down all database-related access features into a set of components.
  - To access data, ADO.NET offers a series of *data providers*, which are broken down into a series of objects handling creation of database connections, sending queries and reading results

ADO.NET


Client
Client
Data Source
DataSet
DataAdapter
Command
Connection
DataReader
ADO.NET Data Provider

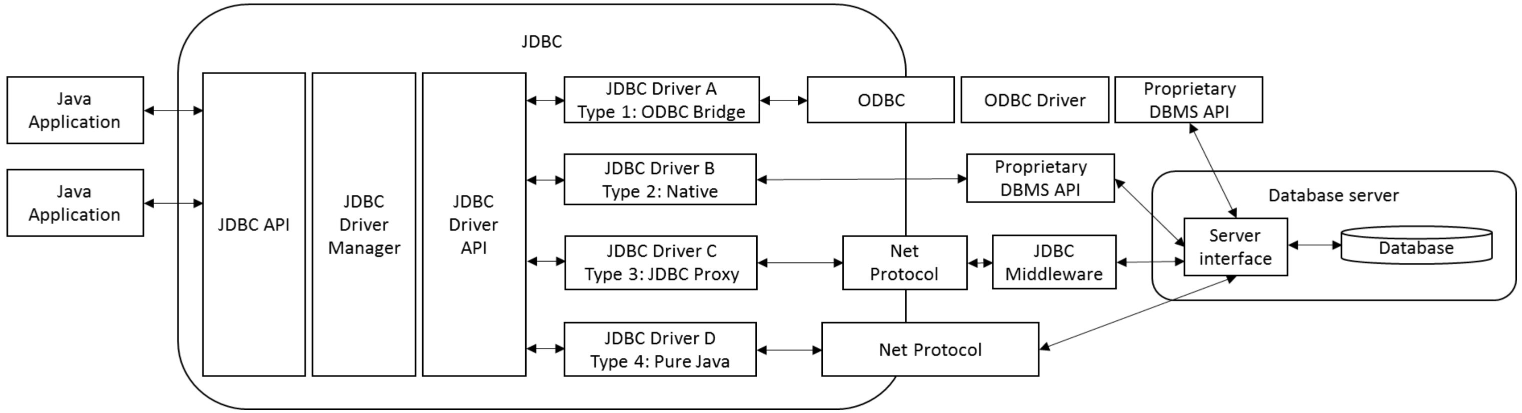
ADO.NET

```
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();
```

JDBC

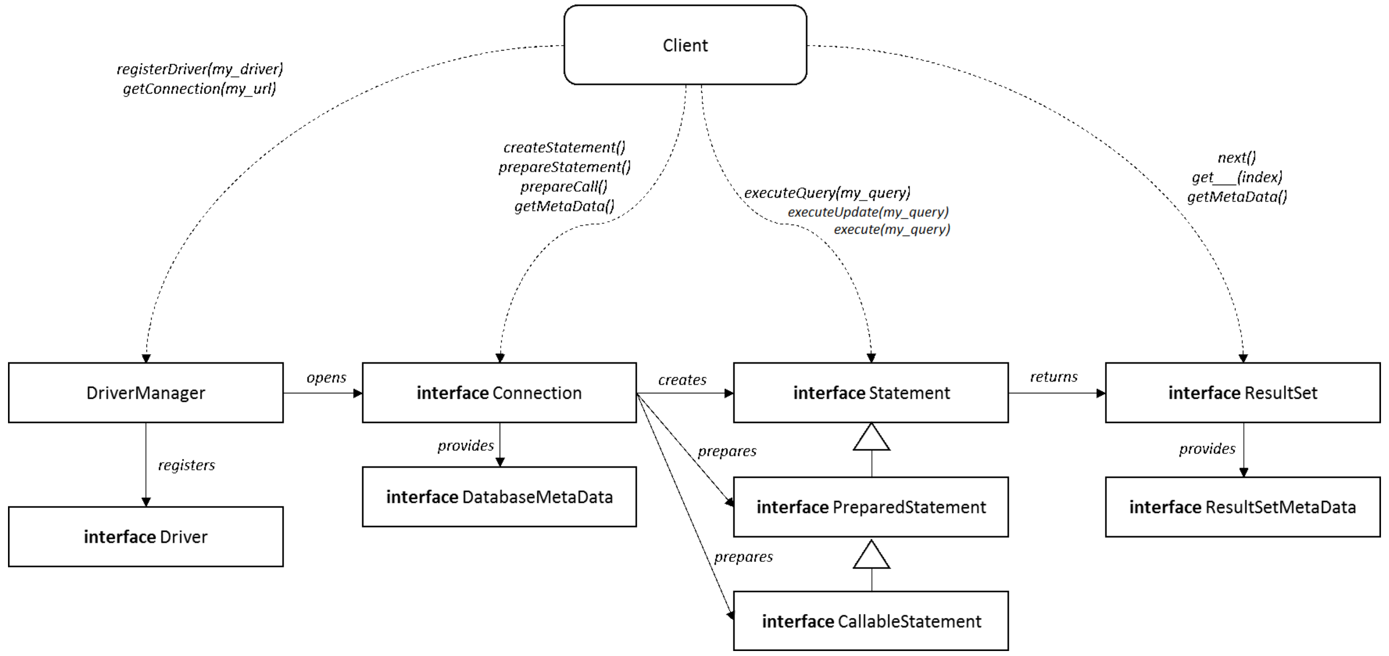
- Java DataBase Connectivity (JDBC) offers a call-level database API
  - inspired by ODBC but developed to be used in Java
  - high portability and the ability to program in an OO way
  - database connections, drivers, queries and results are thus all expressed as objects, based on uniform interfaces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

JDBC


JDBC
Java Application
Java Application
JDBC API
JDBC Driver Manager
JDBC Driver API
JDBC Driver A Type 1: ODBC Bridge
JDBC Driver B Type 2: Native
JDBC Driver C Type 3: JDBC Proxy
JDBC Driver D Type 4: Pure Java
ODBC
ODBC Driver
Proprietary DBMS API
Proprietary DBMS API
Net Protocol
JDBC Middleware
Net Protocol
Database server
Server interface
Database

JDBC

- JDBC exposes a series of object interfaces through which drivers, connections, SQL statements and results are expressed



JDBC

- *DriverManager* is a singleton object which acts as the basic service to manage JDBC drivers
- DBMS driver must be registered with the DriverManager, using the *registerDriver* method
- Database connections can be created using one of the registered drivers by means of the *getConnection* method
  - takes a String parameter representing connection URL: “jdbc:subprotocol:subname” (e.g. “jdbc:sqlite:my_database”)
  - can also take username and password parameters

JDBC

```
DriverManager.registerDriver(new org.sqlite.JDBC());

String dbURL = "jdbc:sqlite:my_database";

Connection conn = DriverManager.getConnection(dbURL);
if (conn != null) {
 System.out.println("Connected to the database");
 DatabaseMetaData dm = conn.getMetaData();
 System.out.println("Driver name: " + dm.getDriverName);
 conn.close();
}
```

JDBC

- Opening a connection returns a *Connection* object, representing a session to a specific database
- The *createStatement* method can be used to create SQL statements
- The *prepareStatement and prepareCall* methods can be used to create objects representing prepared statements and stored procedure calls
- A *Statement* object represents an SQL instruction
  - For a SELECT query, the *executeQuery* method should be invoked, which returns a *ResultSet*

```
Statement selectStatement = conn.createStatement(“select * from books”);
ResultSet selectResult = selectStatement.executeQuery();
```

JDBC

- A ResultSet object contains the result of a SELECT query that was executed by a *Statement* object
- 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
- To overcome this impedance mismatch, JDBC (like ODBC) uses a cursor mechanism in order to loop through result sets
  - A cursor is a programmatic control structure that enables one by one traversal over the records in a query result set (cf. database cursors)

JDBC

```
while (selectResult.next())
{

     String bookTitle = selectResult.getString("title");
     // or: .getString(1);

     int bookPages = selectResult.getInt("num_pages");
     // or: .getInt(2);

     System.out.println(bookTitle + “ has ” + bookPages + “ pages”);
}
```

JDBC

- For INSERT, UPDATE or DELETE queries, the *executeUpdate* method should be called
  - return value is an integer representing the number of rows affected

```
String deleteQuery = “delete from books where num_pages <= 30”;
Statement deleteStatement = conn.createStatement();
int deletedRows = deleteStatement.executeUpdate(deleteQuery);
System.out.println(deletedRows + “ books were deleted”);
```

- Generic *execute* method returns a Boolean value representing whether the just-executed query was a SELECT query, based on which the program can decide to call the *getResultSet* method

JDBC

- *PreparedStatement* interface extends *Statement* with functionalities to bind a query once and then execute it multiple times in an efficient manner
- Prepared statements also provide support to pass query parameters, which should then be instantiated using “setter methods” such as *setInt*, *setString,* …
  - use question marks (“?”) inside the SQL query to indicate that this represents a parameter value

JDBC

```
String selectQuery = “select * from books where num_pages < ? and num_pages > ?”;
Statement preparedSelectStatement = conn.prepareStatement(selectQuery);

int min_pages = 50;
int max_pages = 200;
// Set the value to the first parameter (1):
preparedSelectStatement.setInt(1, min_pages);
// Set the value to the second parameter (2):
preparedSelectStatement.setInt(2, max_pages);

ResultSet resultSet1 = preparedSelectStatement.executeQuery();

// Execute the same query a second time with different parameter values:
preparedSelectStatement.setInt(1, 10);
preparedSelectStatement.setInt(2, 20);
ResultSet resultSet2 = preparedSelectStatement.executeQuery();
```

JDBC

- *CallableStatement* extends *PreparedStatement* and offers support to execute stored procedures
- JDBC supports *updatable result sets*, where rows in a result set can be updated on the fly
- Connection interface also defines *commit*, *rollback*, *setTransactionIsolation*, *setAutoCommit*, *setSavepoint* and *releaseSavepoint* methods
- JDBC API provides no explicit method to start a transaction
- JDBC remains widely popular!

SQLJ

- SQLJ, Java’s embedded, static SQL API, was developed after JDBC and allows embedding SQL statements directly into Java programs

```
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

- Queries directly embedded in Java source code
  - **pre-compiler converts these statements into native Java code**
  - pre-compiler also performs additional checks
  - arguments for embedded SQL statements are passed through host variables (using “:” prefix)
  - compile-time checking cannot be performed when using late-bound parameters
- SQLJ never experienced the success of JDBC due to
  - lack of support for dynamic SQL
  - extra overhead for programmer

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

Language-integrated Querying

- Example: 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)
  – Python SQLAlchemy

Contents

- Database System Architectures
- Database APIs
- **Object Persistence and Object Relational Mappers**
- Exercise

Object Persistence and ORMs

- 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

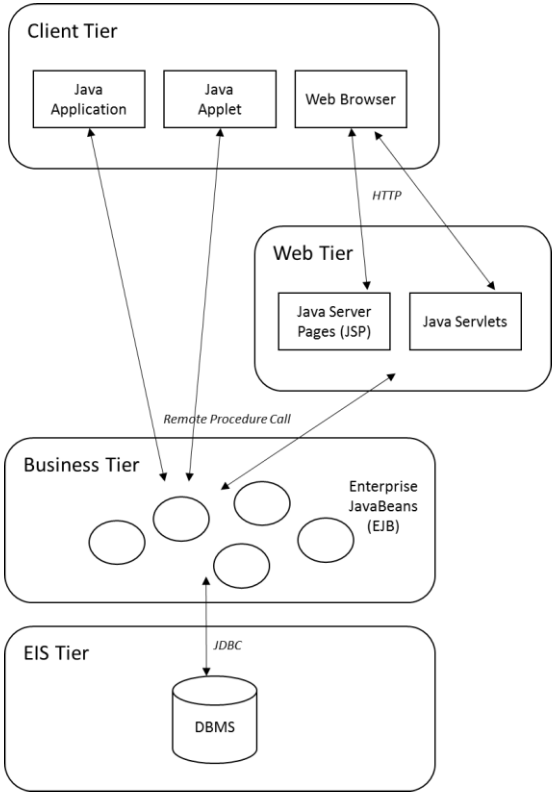
Object Persistence and ORMs

- 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
  - to allow for efficient querying of objects, such entities are frequently mapped to a relational database model using a so-called Object Relational Mapper (ORM)
  - not strictly necessary to utilize an ORM to enable object persistence, though most APIs tightly couple both concepts
- However, the SQL statements generated by the ORM tool might have a negative impact on the performance (see chapter about Database Performance).

Object Persistence with EJB

- Java’s ecosystem was an early adopter of object persistence, built on top of Enterprise JavaBeans (EJB)
- A Java Bean is Java’s term to refer to reusable OO software components.
- Enterprise JavaBeans are ‘business’ components that run within the Java Enterprise Edition (EE) platform

Object Persistence with EJB



Object Persistence with EJB

- Enterprise JavaBeans expand the concept of Java Beans, which encapsulate a piece of re-usable, modular logic
- Beans are essentially nothing more than a normal Java class definition, following some additional rules
- Once a Java Bean is defined, outside frameworks know how to access and modify its properties
- The Enterprise JavaBeans (EJB) standard extends the concept of Java Beans with the goal of utilizing these components in a server environment

Object Persistence with EJB

```
public class BookBean implements java.io.Serializable {

  private String title = null;
  private int numPages = 0;
  private boolean inStock = false;

  /* Default constructor without arguments */

  public BookBean() {
  }

  /* Getters and setters */

  public String getTitle() {
    return title;
  }

  public void setTitle(String value) {
    this.title = value;
  }
```

```
public boolean isInStock() {
    return inStock;
  }


public void setInStock(boolean value) {
    this.inStock = value;
  }


  public int getNumPages() {
    return numPages;
  }

  public void setNumPages(int value) {
    this.numPages = value;
  }
}
```

Object Persistence with JPA

- EJB 3.0 introduces a radical change, inspired by application frameworks such as Spring
- The Java Persistence API forms the replacement for the entity Beans in EJB 3.0, which were removed from this version of the standard
- JPA is just a specification defining a set of interfaces and annotations, and requires an implementation
- Most of the persistence vendors have released implementations of JPA, including Hibernate (Red Hat), TopLink (Oracle), Kodo JDO (Oracle), Cocobase and JPOX.

Object Persistence with JPA

Annotation	Description
@Entity	Declares a persistent POJO class.
@Table	Allows to explicitly specify the name of the relational table to map the persistent POJO class to.
@Column	Allows to explicitly specify the name of the relational table column.
@ID	Maps a persistent POJO class field to a primary key of a relational table.
@Transient	Allows to define POJO class fields that are transient and should not be made persistent.

Object Persistence with JPA

```java
import java.util.List;
import javax.persistence.*;

@Entity // Book is an entity mapped to a table
@Table
public class Book {

  @Id // Use id as the primary key
  // Generate id values automatically:
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int id;
  private String title;

  // Define a many-to-many relation
  @ManyToMany(cascade = {CascadeType.ALL})
  private List<Author> authors;

  public Book(String title) {
    setTitle(title);
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public List<Author> getAuthors() {
    return authors;
  }

  public void setAuthors(List<Author> authors) {
    this.authors = authors;
  }

  public String toString() {
    String r = "Book [id="+id+", title="+title+"]";
    for (Author a : getAuthors()) {
      r += "\nBy author: "+a.toString();
    }
    return r;
  }
}
```

Object Persistence with JPA

```java
import java.util.List;
import javax.persistence.*;

@Entity
@Table
public class Author {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private int id;
        private String name;

        // Many-to-many relation in the other direction
        @ManyToMany(cascade = {CascadeType.ALL})
        private List<Book> books;

        public Author(String name) {
                setName(name);
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public List<Book> getBooks() {
                return books;
        }

        public void setBooks(List<Book> books)
        {
                this.books = books;
        }

        public String toString() {
                return "Author [id=" + id +  ",
name=" + name + "]";
        }
        }
```

Object Persistence with JPA

- Persistence.xml

```
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
         version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
<persistence-unit name="app">
 <!-- We have the following persistable classes: -->
 <class>Book</class>
 <class>Author</class>
 <!-- Settings to connect to the database -->
 <properties>
  <property name="hibernate.archive.autodetection" value="class" />
  <property name="hibernate.connection.driver_class"
       value="org.apache.derby.jdbc.EmbeddedDriver" />
  <property name="hibernate.connection.url"
       value="jdbc:derby:memory:myDB;create=true" />
  <property name="hibernate.show_sql" value="true" />
  <property name="hibernate.flushMode" value="FLUSH_AUTO" />
  <property name="hibernate.hbm2ddl.auto" value="create" />
 </properties>
 </persistence-unit></persistence>
```

Object Persistence with JPA

```
import java.util.ArrayList;
import javax.persistence.*;

public class Test {

       public static void main(String[] args) {

              EntityManagerFactory emfactory =
        Persistence.createEntityManagerFactory("app");

              EntityManager entitymanager = emfactory.createEntityManager();
              entitymanager.getTransaction().begin();

              final Author author1 = new Author("Seppe vanden Broucke");
              final Author author2 = new Author("Wilfried Lemahieu");
              final Author author3 = new Author("Bart Baesens");
              final Book book = new Book("My first book");

              book.setAuthors(new ArrayList<Author>(){{
                     this.add(author1);
                     this.add(author2);  }});
```

Object Persistence with JPA

```
            // Persist the book object, the first two authors will be
// persisted as well as they are linked to the book
            entitymanager.persist(book);
            entitymanager.getTransaction().commit();

            System.out.println(book);

            // Now persist author3 as well
            entitymanager.persist(author3);

            entitymanager.close();
            emfactory.close();
      }
}
```

Object Persistence with JPA

```
Hibernate: create table Author (id integer not null, name varchar(255), primary key (id))
Hibernate: create table Author_Book (Author_id integer not null, books_id integer not null)
Hibernate: create table Book (id integer not null, title varchar(255), primary key (id))
Hibernate: create table Book_Author (Book_id integer not null, authors_id integer not null)
Hibernate: alter table Author_Book add constraint FK3wjtcus6sftdj8dfvthui6335 foreign key (books_id)
references Book
Hibernate: alter table Author_Book add constraint FKo3f90h3ibr9jtq0u93mjgi5qd foreign key (Author_id)
references Author
Hibernate: alter table Book_Author add constraint FKt42qaxhbq87yfijncjfrs5ukc foreign key (authors_id)
references Author
Hibernate: alter table Book_Author add constraint FKsbb54ii8mmfvh6h2lr0vf2r7f foreign key (Book_id)
references Book

Hibernate: values next value for hibernate_sequence
Hibernate: values next value for hibernate_sequence
Hibernate: insert into Book (title, id) values (?, ?)
Hibernate: insert into Author (name, id) values (?, ?)
Hibernate: insert into Book_Author (Book_id, authors_id) values (?, ?)

Book [id=1, title=My first book]
By author: Author [id=0, name=Seppe vanden Broucke]
Author [id=1, name=Wilfried Lemahieu]
Hibernate: values next value for hibernate_sequence
Author [id=2, name=Bart Baesens]
Hibernate: insert into Author (name, id) values (?, ?)
```

Object Persistence with JPA

- *EntityManager.find* method is used to look up entities by the entity’s primary key:
  ```
  entitymanager.find(Author.class, 2)
  ```
- The *EntityManager.createQuery* method can be used to query the datastore using Java Persistence query language queries:

```
 entitymanager.createQuery(
         "SELECT c FROM Author c WHERE c.name LIKE :authName")
                       .setParameter("authName", "%vanden%")
                       .setMaxResults(10)
                       .getResultList()
```

Object Persistence with JPA

- JPA comes with its own query language (JPQL) which closely resembles SQL, including support for SELECT, UPDATE and DELETE statements
- Why JPQL? The reason has to do with portability
  - Contrary to earlier approaches and universal APIs, where it was – in theory – easy to migrate an application to a different DBMS, differences in SQL support might still cause a client application to fail in new DBMS environments
  - JPQL tries to prevent this by inserting itself as a more pure, vendor-agnostic SQL, and will translate JPQL queries to appropriate SQL statements
  - Also still possible to use raw SQL statements

Object Persistence with JDO

- Just as JPA, the Java Database Objects (JDO) API also arose from the failed adoption of the ODMG standard and the desire to "break out" object persistence capabilities from EJB
- Unlike JPA, which is primarily targeted towards relational DBMS data stores, JDO is agnostic to the technology of the data store used
- Like JPA, JDO comes with a query language called JDOQL or Java

Object Persistence: Others

- Java is not the only programming language whose ecosystem offers object persistence APIs using ORM
  - The Ruby on Rails ecosystem makes use of the ActiveRecord library
  - The .NET framework comes with the Entity Framework (EF)
  - Python also comes with many libraries such as SQLAlchemy

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	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
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	Relational databases supporting SQL or other data sources	Examples: jOOQ 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
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
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, 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
Java Persistence API (JDA 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, 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
Java Database Objects (JDO)	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

Database API Summary

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent	Data sources	Other
ORM APIs (ActiveRecord, Entity Framework**)**	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

Demo in Python

See `sqlserver-with-pyodbc-and-sqlalchemy.py`

Contents

- Database System Architectures
- Database APIs
- Object Persistence and Object Relational Mappers
- **Exercise**

Exercise

Indicate the properties of each Python technology in the table below

Technology	Embedded or Call-level	Early or late binding	Objects in host programming language represent
PyODBC with plain SQL			
PyODBC with stored procedure			
SQL Alchemy Core			
SQL Alchemy ORM			