

CS544

LESSON 3

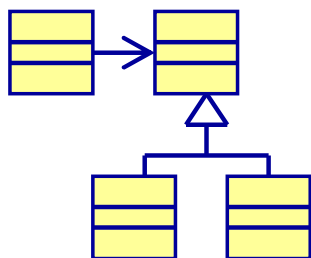
JDBC AND JPA

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
March 28 Lesson 1 Enterprise Architecture introduction and Spring Boot	March 29 Lesson 2 Dependency injection AOP	March 30 Lesson 3 JDBC JPA	March 31 Lesson 4 JPA mapping 1	April 1 Lesson 5 JPA mapping 2	April 2 Lesson 6 JPA queries	April 3
April 4 Lesson 7 Transactions	April 5 Lesson 8 MongoDB	April 6 Midterm Review	April 7 Midterm exam	April 8 Lesson 9 REST webservises	April 9 Lesson 10 SOAP webservises	April 10
April 11 Lesson 11 Messaging	April 12 Lesson 12 Scheduling Events Configuration	April 13 Lesson 13 Monitoring	April 14 Lesson 14 Testing your application	April 15 Final review	April 16 Final exam	April 17
April 18 Project	April 19 Project	April 20 Project	April 21 Presentations			

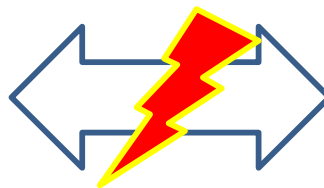
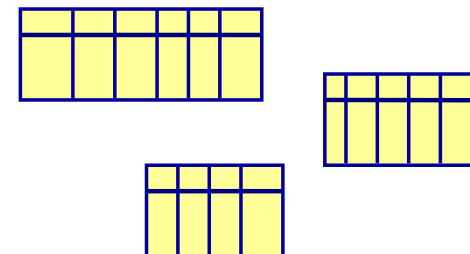
Object-Relational Mismatch

Object Oriented	Relational Database
Objects are instantiations of classes and have identity (object1 == object2)	In the relational model the table name and primary key are used to identity a row in a table
Objects have associations (one-to-one, many-to-one, ...)	Relational model has foreign keys and link tables
OO has inheritance	Relational model has no such thing
Data can be accessed by following object associations	Data can be accessed using queries and joins

Object Model



Relational Schema

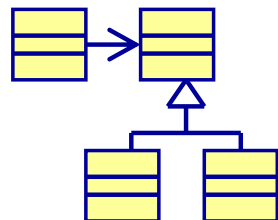


Java Persistence Possibilities

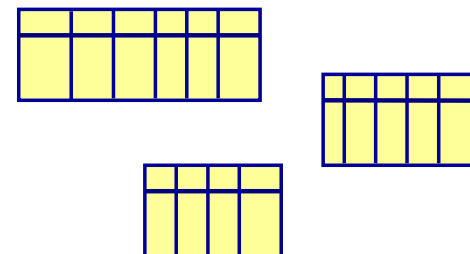
Possibility	Example
Stored Procedures	Stored PL/SQL or Transact-SQL procedures
SQL in the Application	Putting SQL in strings inside the application, using the JDBC API straight or wrapped by the Spring JDBC template
Object Relational Mapping	Using tools such as Hibernate, Toplink, JDO, and JPA to map an Object Model onto a Relational Schema

More OO Friendly
↓

Object Model

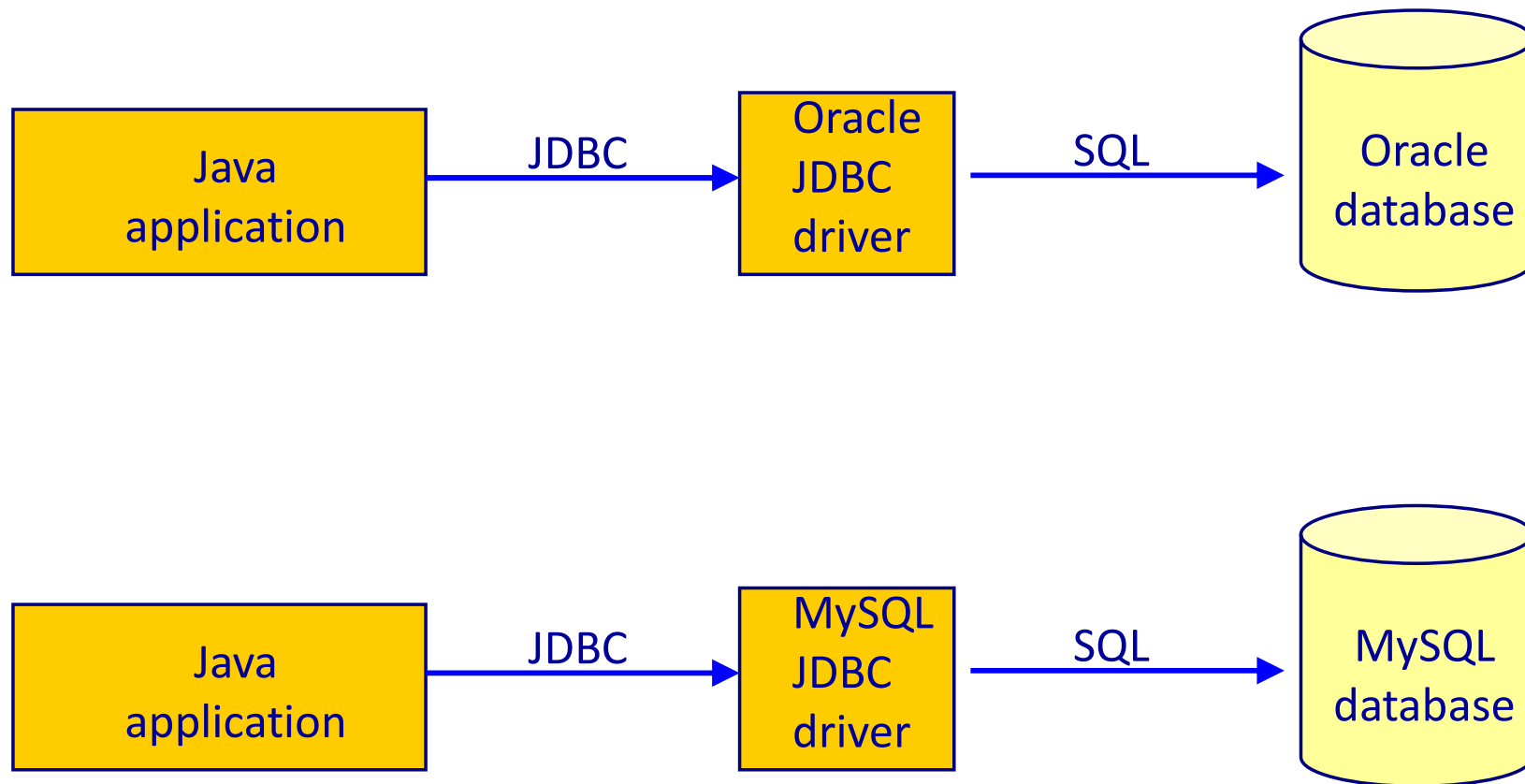


Relational Schema

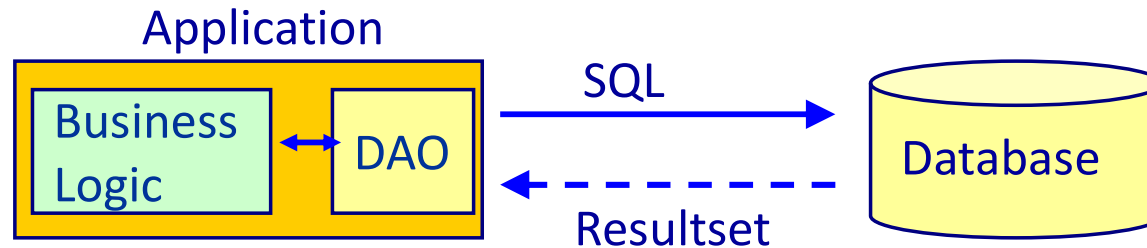


JDBC

JDBC



Data Access Object (DAO)



```
public class ProductDAO {
    public void save(Product product) {...}
    public void update(Product product) {...}
    public Product load(int productNumber) {...}
    public void delete(int productNumber) {...}
    public Collection<Product> getAllProducts() {...}
}
```

Typical JDBC code

```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET firstname= ?,
                                                         lastname= ? WHERE employeenumber=?");

        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e1);
        }
    }
}
```


Try-catch-finally-try-catch

```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET firstname= ?,
                                                    lastname= ? WHERE employeenumber=?");

        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e);
        }
    }
}
```

Open and closing the connection

```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET firstname= ?,
                                                    lastname= ? WHERE employeenumber=?");

        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e1);
        }
    }
}
```

Open connection

Close connection

Transaction handling

```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET firstname= ?,
                                                    lastname= ? WHERE employeenumber=?");

        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e1);
        }
    }
}
```

Start transaction

Commit transaction

Rollback transaction

Exception handling

```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET firstname= ?,
                                                    lastname= ? WHERE employeenumber=?");

        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e1);
        }
    }
}
```

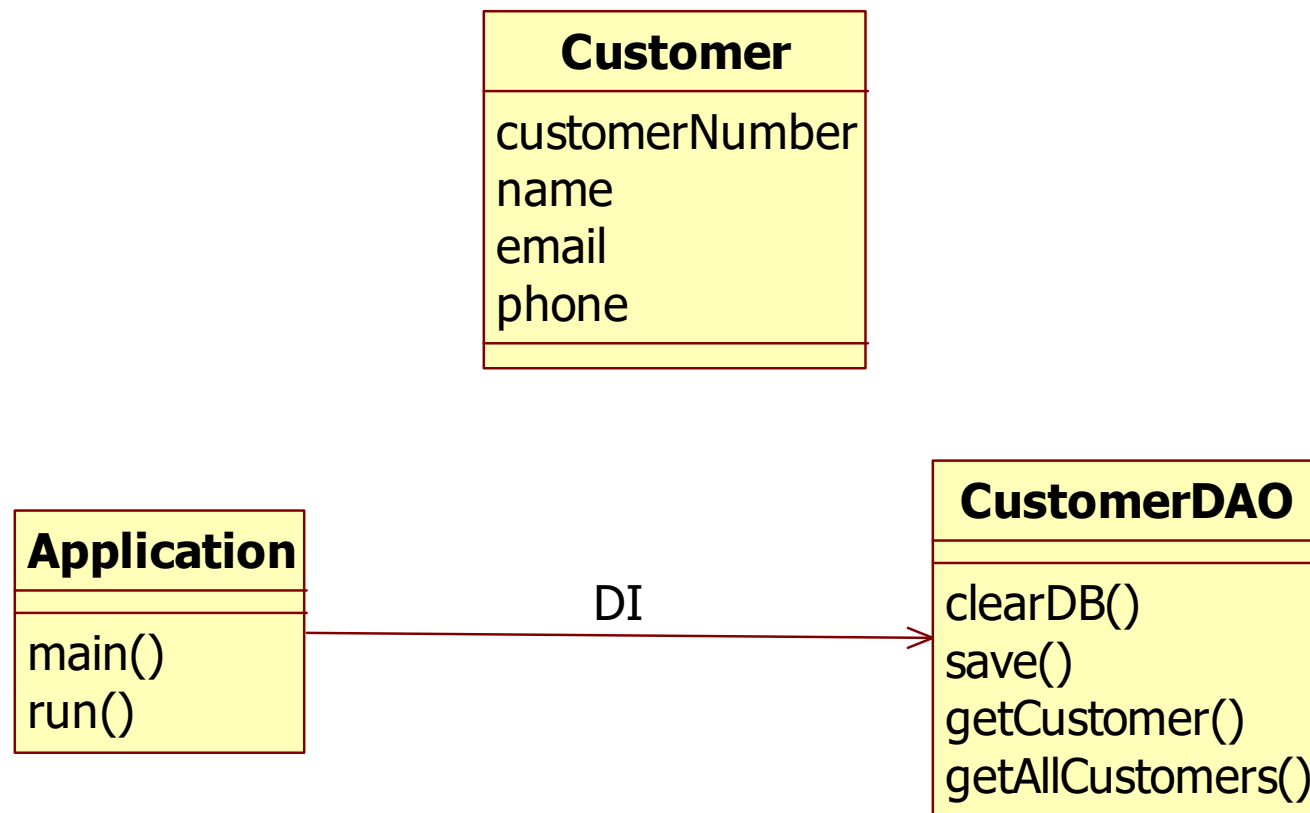
Disadvantages of JDBC

- Code duplication
 - Open and closing connection
 - Transaction handling
 - Exception handling
- Try-catch-finally-try-catch code
- Large number of lines of JDBC code
- In case of a SQLException, the database returns a database-specific error code

Spring JDBC

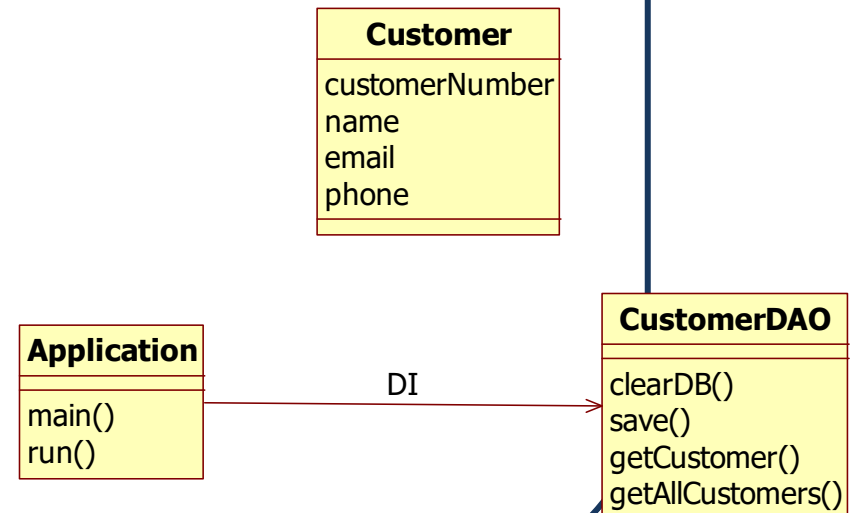
- JDBCTemplate takes care of
 - Open and closing JDBC connection
 - Transaction handling
 - Exception handling
 - Clear, human readable error messages

Spring Boot JDBC example

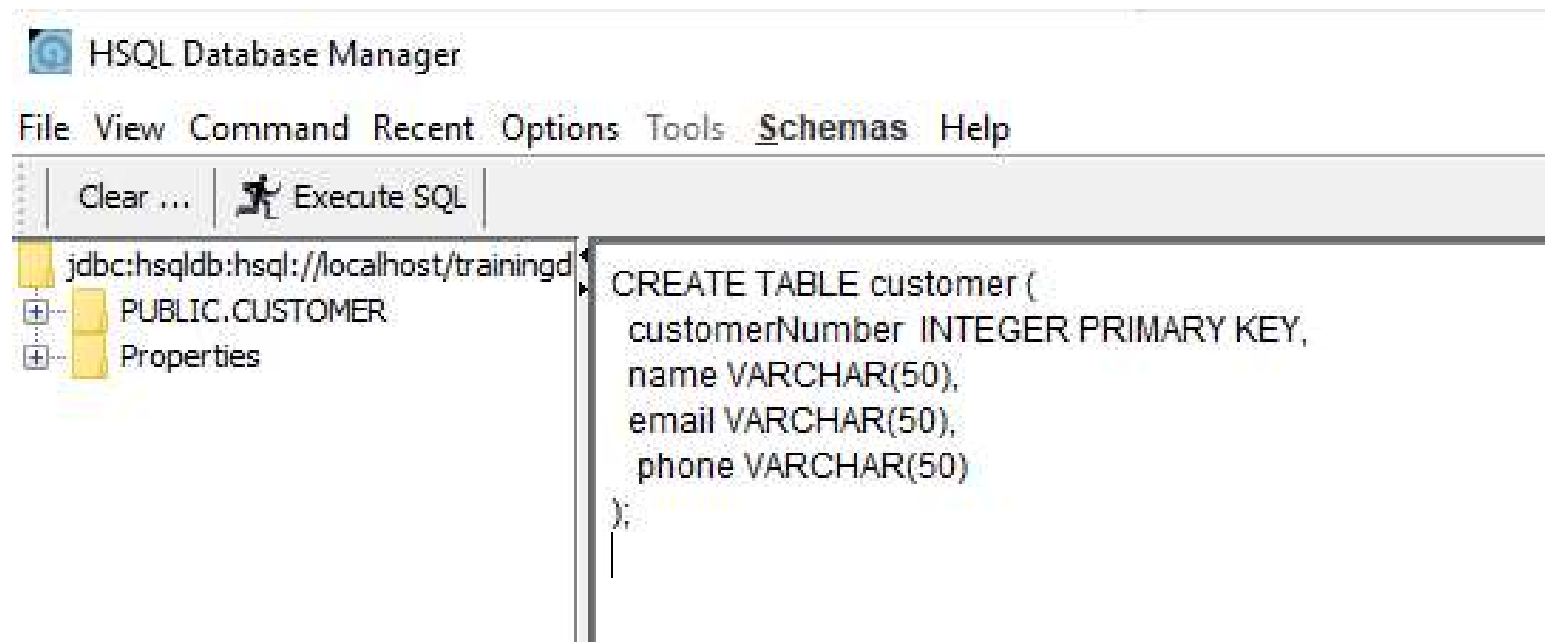


Customer

```
public class Customer {  
    private int customerNumber;  
    private String name;  
    private String email;  
    private String phone;  
  
    public Customer(int customerNumber, String name, String email, String phone) {  
        this.customerNumber = customerNumber;  
        this.name = name;  
        this.email = email;  
        this.phone = phone;  
    }  
  
    @Override  
    public String toString() {  
        return "Customer{" +  
            "customerNumber=" + customerNumber +  
            ", name='" + name + '\" +  
            ", email='" + email + '\" +  
            ", phone='" + phone + '\" +  
            '}';  
    }  
    ...  
}
```



Create a table in the database



CustomerDAO

@Repository

```
public class CustomerDAO {
```

@Autowired

```
private NamedParameterJdbcTemplate jdbcTemplate;
```

```
public void clearDB() {
```

```
    Map<String,Object> namedParameters = new HashMap<String,Object>();
```

```
    jdbcTemplate.update("DELETE from customer",namedParameters);
```

```
}
```

```
public void save(Customer customer) {
```

```
    Map<String,Object> namedParameters = new HashMap<String,Object>();
```

```
    namedParameters.put("customernumber", customer.getCustomerNumber());
```

```
    namedParameters.put("name", customer.getName());
```

```
    namedParameters.put("email", customer.getEmail());
```

```
    namedParameters.put("phone", customer.getPhone());
```

```
    int updateresult = jdbcTemplate.update("INSERT INTO customer VALUES ( :customernumber, :name, :email, :phone)",namedParameters);
```

```
}
```

Customer

customerNumber
name
email
phone

Application

main()
run()

DI

CustomerDAO

clearDB()
save()
getCustomer()
getAllCustomers()

CustomerDAO

```
public Customer getCustomer(int customerNumber){
    Map<String,Object> namedParameters = new HashMap<String,Object>();
    namedParameters.put("customerNumber", customerNumber);
    Customer customer = jdbcTemplate.queryForObject("SELECT * FROM customer WHERE "
        + "customerNumber=:customerNumber ",
        namedParameters,
        (rs, rowNum) -> new Customer( rs.getInt("customerNumber"),
            rs.getString("name"),
            rs.getString("email"),
            rs.getString("phone")));
    return customer;
}

public List<Customer> getAllCustomers(){
    List<Customer> customers = jdbcTemplate.query("SELECT * FROM customer",
        new HashMap<String, Customer>(),
        (rs, rowNum) -> new Customer( rs.getInt("customerNumber"),
            rs.getString("name"),
            rs.getString("email"),
            rs.getString("phone")));
    return customers;
}
}
```

Application

@SpringBootApplication

```
public class Application implements CommandLineRunner {
```

@Autowired

```
private CustomerDAO customerDao;
```

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

```
    SpringApplication.run(Application.class, args);
```

```
}
```

@Override

```
public void run(String... args) throws Exception {
```

```
    customerDao.clearDB();
```

```
    Customer customer = new Customer(101, "John doe", "johnd@acme.com", "0622341678");
```

```
    customerDao.save(customer);
```

```
    customer = new Customer(66, "James Johnson", "jj123@acme.com", "068633452");
```

```
    customerDao.save(customer);
```

```
    System.out.println(customerDao.getCustomer(101));
```

```
    System.out.println(customerDao.getCustomer(66));
```

```
    System.out.println("-----All customers -----");
```

```
    System.out.println(customerDao.getAllCustomers());
```

```
}
```

```
}
```

application.properties



```
spring.datasource.url=jdbc:hsqldb:hsql://localhost/trainingdb  
spring.datasource.username=SA  
spring.datasource.password=  
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
```

```
logging.level.root=ERROR  
logging.level.org.springframework=ERROR
```

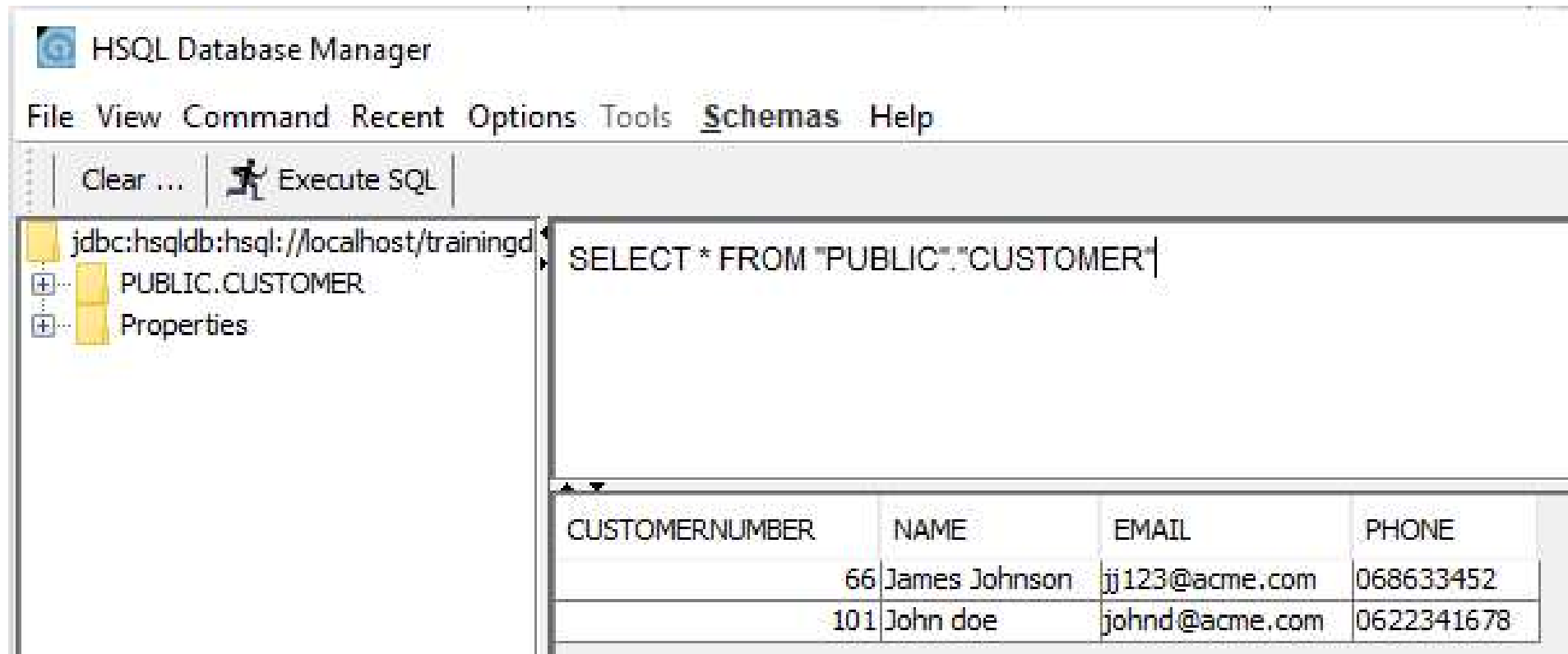
output

@Override

```
public void run(String... args) throws Exception {  
    customerDao.clearDB();  
    Customer customer = new Customer(101, "John doe", "johnd@acme.com", "0622341678");  
    customerDao.save(customer);  
    customer = new Customer(66, "James Johnson", "jj123@acme.com", "068633452");  
    customerDao.save(customer);  
    System.out.println(customerDao.getCustomer(101));  
    System.out.println(customerDao.getCustomer(66));  
    System.out.println("-----All customers -----");  
    System.out.println(customerDao.getAllCustomers());  
}
```

```
Customer{customerNumber=101, name='John doe', email='johnd@acme.com', phone='0622341678'}  
Customer{customerNumber=66, name='James Johnson', email='jj123@acme.com', phone='068633452'}  
-----All customers -----  
[Customer{customerNumber=66, name='James Johnson', email='jj123@acme.com', phone='068633452'},  
Customer{customerNumber=101, name='John doe', email='johnd@acme.com', phone='0622341678'}]
```

The database



The screenshot shows the HSQL Database Manager interface. The title bar reads "HSQL Database Manager". The menu bar includes "File", "View", "Command", "Recent", "Options", "Tools", "Schemas", and "Help". Below the menu bar is a toolbar with "Clear ..." and "Execute SQL" buttons. The left pane shows a tree view with the following structure:

- jdbc:hsqldb:hsq://localhost/trainingd
 - PUBLIC.CUSTOMER
 - Properties

The main SQL editor contains the query:

```
SELECT * FROM "PUBLIC"."CUSTOMER"
```

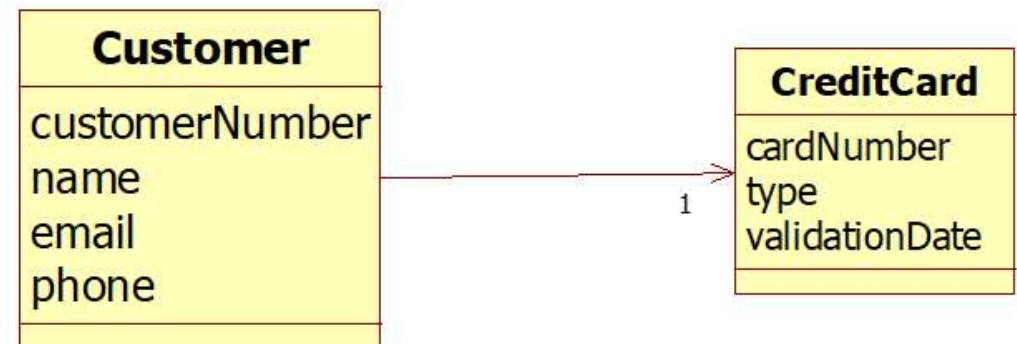
Below the editor, the results are displayed in a table:

CUSTOMERNUMBER	NAME	EMAIL	PHONE
66	James Johnson	jj123@acme.com	068633452
101	John doe	johnd@acme.com	0622341678

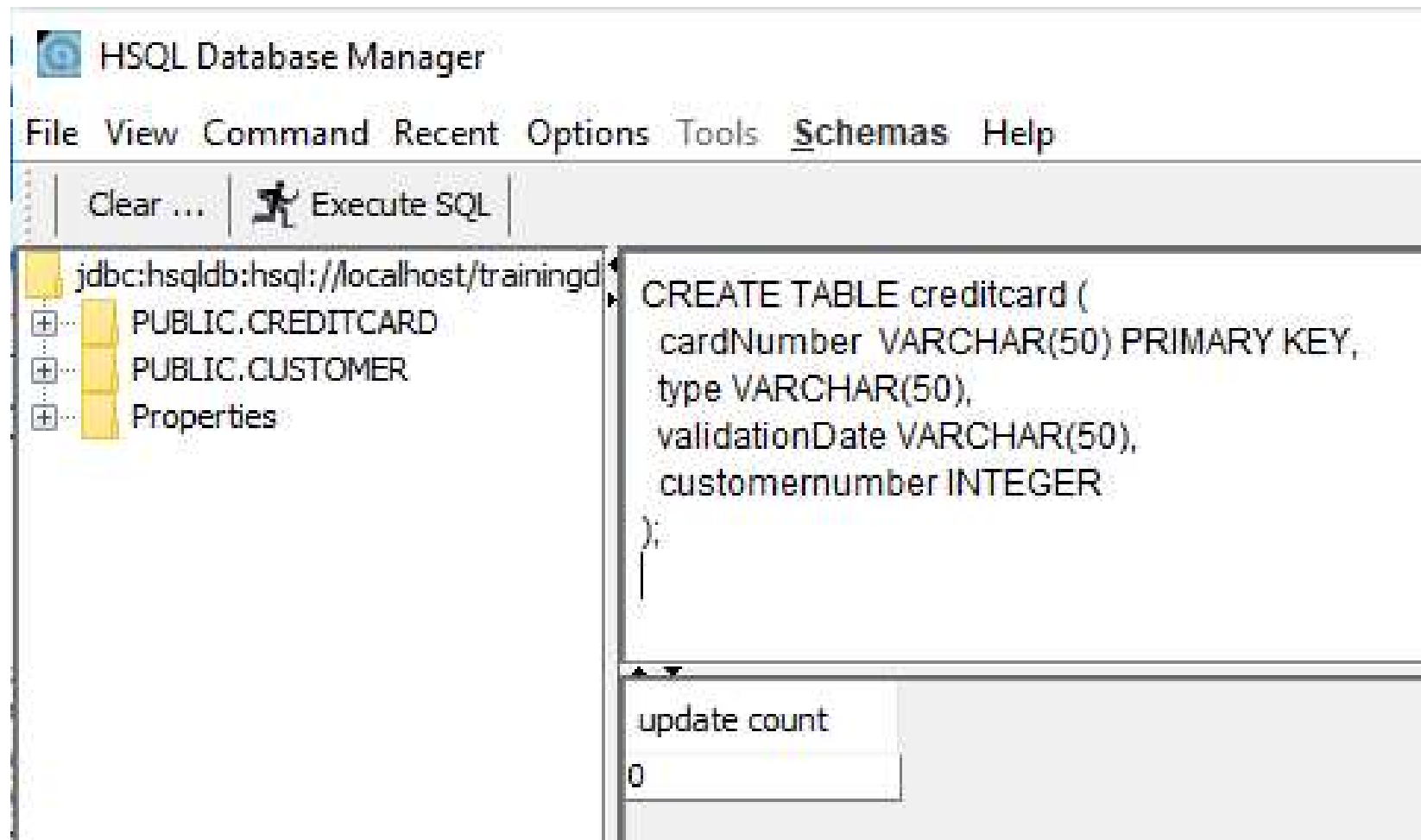
Let's add a class

```
public class Customer {  
    private int customerNumber;  
    private String name;  
    private String email;  
    private String phone;  
    private CreditCard creditCard;  
    ...  
}
```

```
public class CreditCard {  
    private String cardNumber;  
    private String type;  
    private String validationDate;  
    ...  
}
```



Create a new table



CustomerDAO save()

```
public void save(Customer customer) {
    Map<String,Object> namedParameters = new HashMap<String,Object>();
    namedParameters.put("customernumber", customer.getCustomerNumber());
    namedParameters.put("name", customer.getName());
    namedParameters.put("email", customer.getEmail());
    namedParameters.put("phone", customer.getPhone());
    jdbcTemplate.update("INSERT INTO customer VALUES ( :customernumber, :name, :email,
:phone)",namedParameters);

    // save creditcard
    Map<String,Object> namedParameterscc = new HashMap<String,Object>();
    namedParameterscc.put("customernumber", customer.getCustomerNumber());
    namedParameterscc.put("cardnumber", customer.getCreditCard().getCardNumber());
    namedParameterscc.put("type", customer.getCreditCard().getType());
    namedParameterscc.put("validationDate", customer.getCreditCard().getValidationDate());
    jdbcTemplate.update("INSERT INTO creditcard VALUES ( :cardnumber, :type, :validationDate,
:customernumber)",namedParameterscc);
}
```

CustomerDAO getCustomer()

```
public Customer getCustomer(int customerNumber){
    Map<String,Object> namedParameters = new HashMap<String,Object>();
    namedParameters.put("customerNumber", customerNumber);
    Customer customer = jdbcTemplate.queryForObject("SELECT * FROM customer WHERE "
        + "customerNumber=:customerNumber ",
        namedParameters,
        (rs, rowNum) -> new Customer( rs.getInt("customerNumber"),
            rs.getString("name"),
            rs.getString("email"),
            rs.getString("phone")));

    CreditCard creditCard = getCreditCardForCustomer(customer.getCustomerNumber());
    customer.setCreditCard(creditCard);
    return customer;
}
```

CustomerDAO

getCreditCardForCustomer

```
CreditCard getCreditCardForCustomer(int customerNumber){
    Map<String,Object> namedParameters = new HashMap<String,Object>();
    namedParameters.put("customerNumber", customerNumber);
    CreditCard creditCard = jdbcTemplate.queryForObject("SELECT * FROM creditcard WHERE "
        + "customerNumber=:customerNumber ",
        namedParameters,
        (rs, rowNum) -> new CreditCard( rs.getString("cardnumber"),
            rs.getString("type"),
            rs.getString("validationDate")));

    return creditCard;
}
```

Application



```
public void run(String... args) throws Exception {  
    customerDao.clearDB();  
    Customer customer = new Customer(101, "John doe", "johnd@acme.com", "0622341678");  
    CreditCard creditCard = new CreditCard("12324564321", "Visa", "11/23");  
    customer.setCreditCard(creditCard);  
    customerDao.save(customer);  
    customer = new Customer(66, "James Johnson", "jj123@acme.com", "068633452");  
    creditCard = new CreditCard("99876549876", "MasterCard", "01/24");  
    customer.setCreditCard(creditCard);  
    customerDao.save(customer);  
    System.out.println(customerDao.getCustomer(101));  
    System.out.println(customerDao.getCustomer(66));  
    System.out.println("-----All customers -----");  
    System.out.println(customerDao.getAllCustomers());  
}
```

Output and tables

```
Customer{customerNumber=101, name='John doe', email='johnd@acme.com', phone='0622341678',  
creditCard=CreditCard{cardNumber='12324564321', type='Visa', validationDate='11/23'}}  
Customer{customerNumber=66, name='James Johnson', email='jj123@acme.com', phone='068633452',  
creditCard=CreditCard{cardNumber='99876549876', type='MasterCard', validationDate='01/24'}}
```

-----All customers -----

```
[Customer{customerNumber=66, name='James Johnson', email='jj123@acme.com', phone='068633452',  
creditCard=CreditCard{cardNumber='99876549876', type='MasterCard', validationDate='01/24'}},  
Customer{customerNumber=101, name='John doe', email='johnd@acme.com', phone='0622341678',  
creditCard=CreditCard{cardNumber='12324564321', type='Visa', validationDate='11/23'}}]
```

```
select * from customer|
```

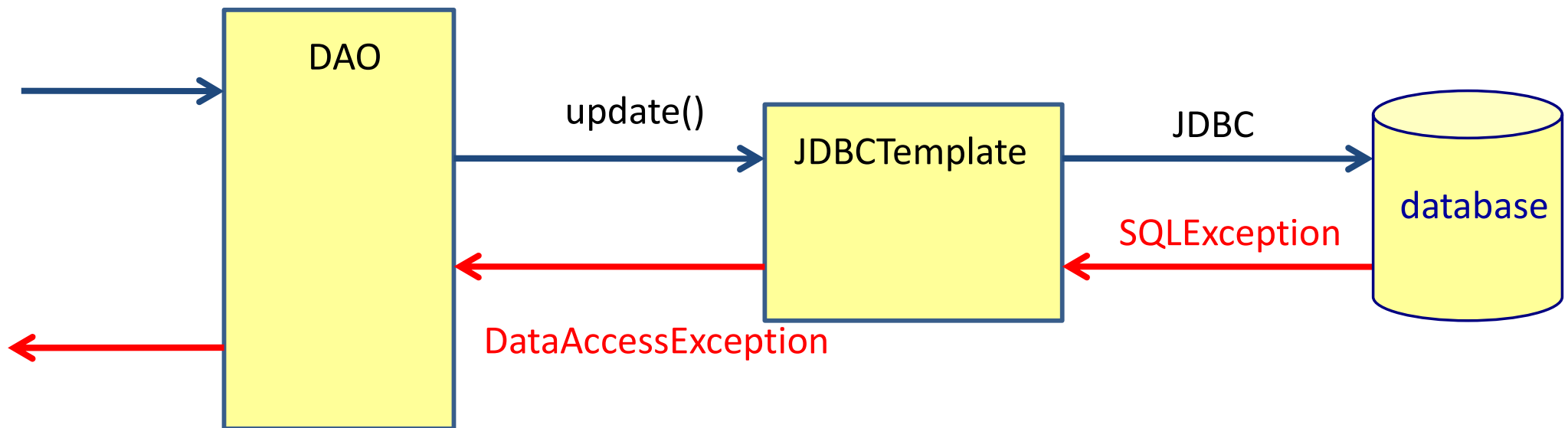
CUSTOMERNUMBER	NAME	EMAIL	PHONE
66	James Johnson	jj123@acme.com	068633452
101	John doe	johnd@acme.com	0622341678

```
select * from creditcard|
```

CARDNUMBER	TYPE	VALIDATIONDATE	CUSTOMERNUMBER
12324564321	Visa	11/23	101
99876549876	MasterCard	01/24	66

Exception handling

- JDBC throws a `java.sql.SQLException`
 - Checked exception, must be caught
- JDBCTemplate throws a `DataAccessException`
 - Runtime exception



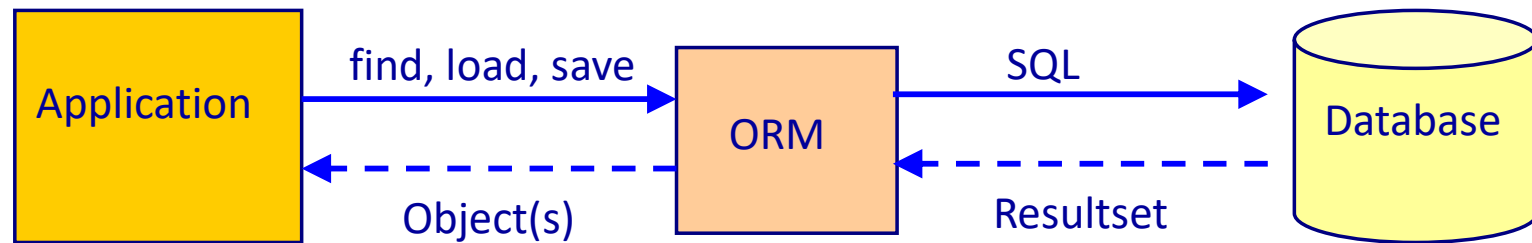
Main point

- The Spring JdbcTemplate takes care of all the necessary JDBC plumbing.

Science of Consciousness: By watering the root one can enjoy the fruit.

JPA

Object Relational Mapper (ORM)

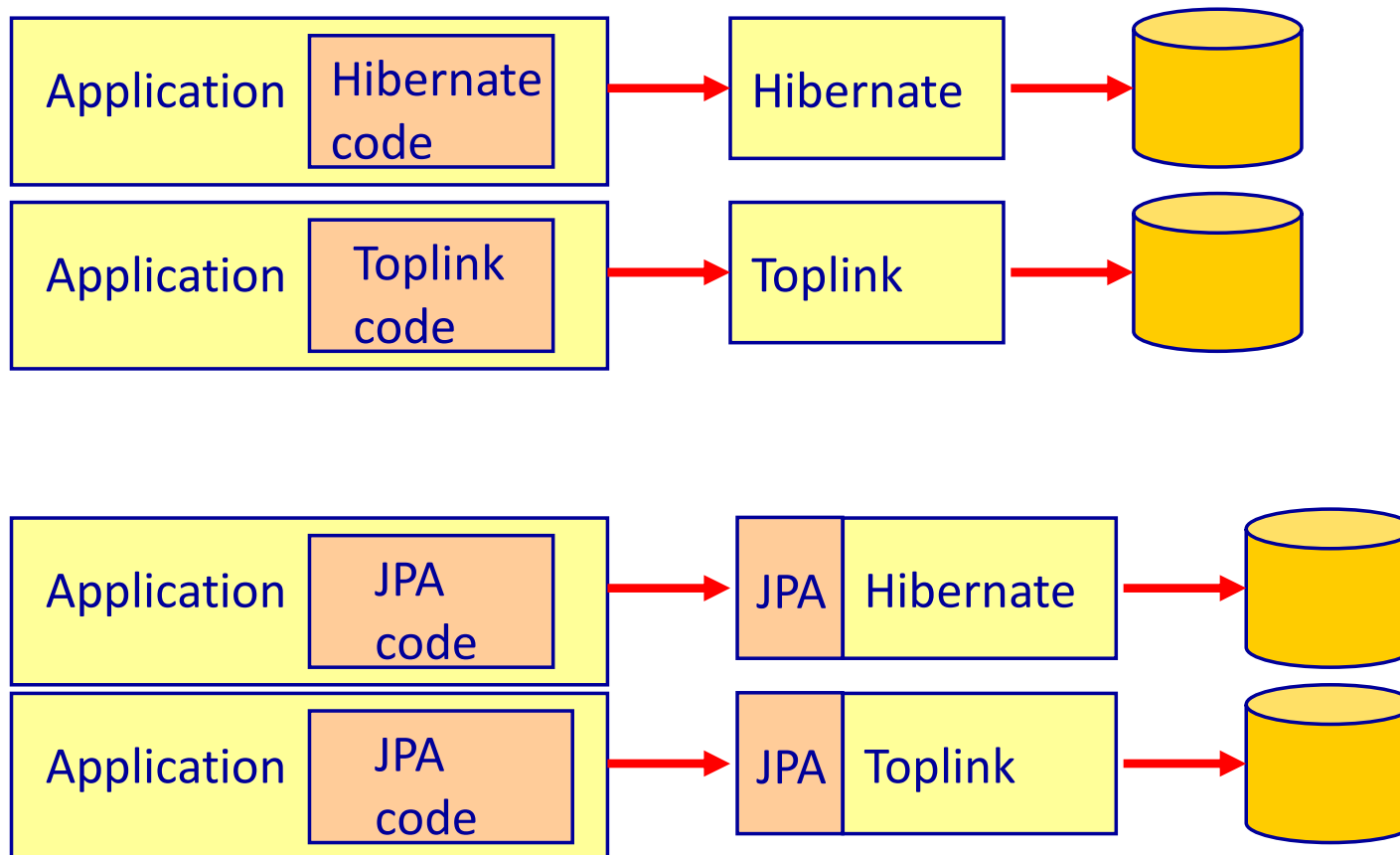


Advantages of an ORM

- Productivity
 - Less lines of persistence code
- Maintenance is easier
 - Less lines of persistence code
 - Mapping is defined at one place
- Performance
 - Caching
 - Higher productivity gives time to optimize
 - Projects under time pressure often don't have time to optimize
 - The ORM developers spend a lot of time in optimizing the ORM tool.

What is JPA?


- Java Persistence API
- Java EE standard for persistency with an ORM



JPA persistence methods

- `find()`
- `persist()`
- `merge()`
- `remove()`
- `createQuery()`

find()



```
Employee employee = entityManager.find(Employee.class, employeeid );
```

- Retrieve the entity object given the id

persist()

```
Employee employee = new Employee();  
employee.setFirstname("Frank");  
employee.setLastname("Miller");  
//save the employee  
entityManager.persist(employee);
```

- Saves the entity object in the database
- `persist()` corresponds with an INSERT in the database

merge()

```
entityManager.merge(employee);
```

- Updates the entity object in the database
- update() corresponds with an UPDATE in the database

remove()

```
entityManager.remove(employee);
```

- Removes the entity object from the database
- delete() corresponds with an DELETE from the database

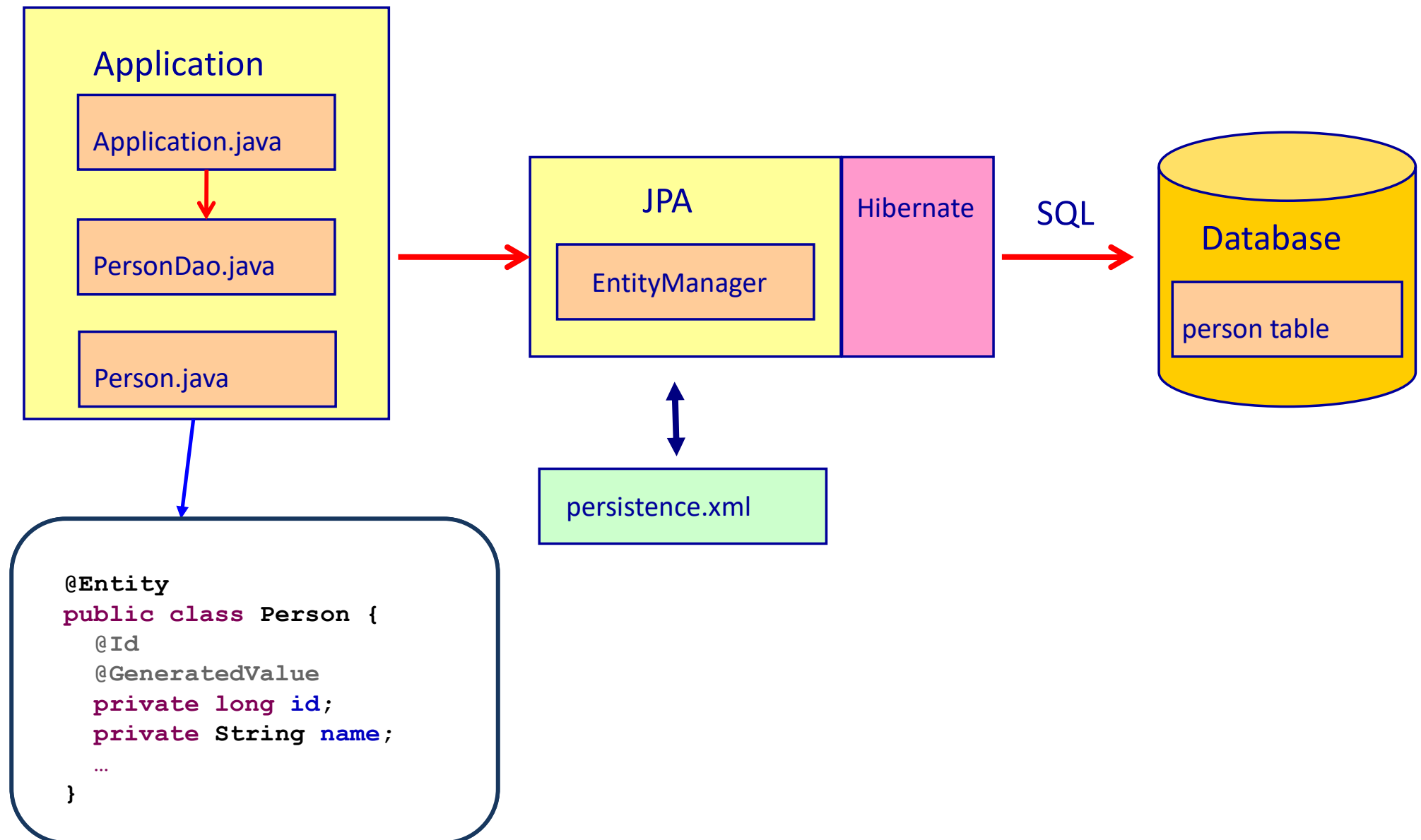
createQuery



```
List<Person> personList = entityManager.createQuery("select p from Person p")  
                                              .getResultList();
```

- Lets you specify a JPQL query which will be translated into a SQL statement that is sent to the database

A simple JPA example



HelloWorld JPA: Person.java and persistence.xml

META-INF/persistence.xml

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    ...
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="jpaunit">
    <class>domain.Person</class>
    <properties>
      <property name="hibernate.connection.driver_class" value="org.hsqldb.jdbcDriver"/>
      <property name="hibernate.connection.url"
        value="jdbc:hsqldb:hsql://localhost/trainingdb"/>
      <property name="hibernate.connection.username" value="SA"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <!-- Always drop and recreate the database schema on startup -->
      <property name="hibernate.hbm2ddl.auto" value="create"/>
      <!-- Show all SQL DML executed by Hibernate -->
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.use_sql_comments" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Name of the JPA Unit

HelloWorld JPA: PersonDaoImpl.java



```
public class PersonDaoImpl implements PersonDao {
    private EntityManagerFactory emf;

    public PersonDaoImpl(EntityManagerFactory emf) {
        this.emf=emf;
    }

    public Person getPerson(long id){
        Person person=null;
        EntityManager em= emf.createEntityManager();
        EntityTransaction tx= em.getTransaction();
        try {
            tx.begin();
            person = em.find(Person.class, id);
            tx.commit();
        } catch (Exception e) {
            tx.rollback();
            throw new DaoException(e);
        } finally {
            em.close();
        }
        return person;
    }
}
```

...

HelloWorld JPA : PersonDaoImpl.java

```
public Collection<Person> getAllPersons() {
    Collection<Person> personList = null;
    EntityManager em= emf.createEntityManager();
    EntityTransaction tx= em.getTransaction();
    try {
        tx.begin();
        personList = em.createQuery("from Person").getResultList();
        tx.commit();
    } catch (Exception e) {
        tx.rollback();
        throw new DaoException(e);
    } finally {
        em.close();
    }
    return personList;
}
```

```
public void savePerson(Person person){
    EntityManager em= emf.createEntityManager();
    EntityTransaction tx= em.getTransaction();
    try {
        tx.begin();
        em.persist(person);
        tx.commit();
    } catch (Exception e) {
        tx.rollback();
        throw new DaoException(e);
    } finally {
        em.close();
    }
}
```

...

HelloWorld JPA : PersonDaoImpl.java

```
public void updatePerson(Person person){
    EntityManager em= emf.createEntityManager();
    EntityTransaction tx= em.getTransaction();
    try {
        tx.begin();
        em.merge(person);
        tx.commit();
    } catch (Exception e) {
        tx.rollback();
        throw new DaoException(e);
    } finally {
        em.close();
    }
}

public void deletePerson(Person person){
    EntityManager em= emf.createEntityManager();
    EntityTransaction tx= em.getTransaction();
    try {
        tx.begin();
        Person thePerson=em.getReference(Person.class, person.getId());
        em.remove(thePerson);
        tx.commit();
    } catch (Exception e) {
        tx.rollback();
        throw new DaoException(e);
    } finally {
        em.close();
    }
}
}
```

HelloWorld JPA: Application.java

```
public class Application {
    public static void main(String[] args) {
        //create an EntityManagerFactory
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpaunit");

        // create the DAO
        PersonDao personDao = new PersonDaoImpl(emf);
        // create 2 persons and save them in the database
        Person fbperson = new Person(1, "Frank Brown");
        personDao.savePerson(fbperson);
        Person mjperson = new Person(2, "Mary Jones");
        personDao.savePerson(mjperson);
        // load and print all persons from the database
        System.out.println("All persons in the database:");
        Collection<Person> personList= personDao.getAllPersons();
        for (Person person : personList){
            System.out.println(person.getId()+" - "+person.getName());
        }
        // delete Mary Jones
        System.out.println("Delete Mary Jones");
        personDao.deletePerson(mjperson);
        // update Frank Brown to Frank Johnson
        System.out.println("Update Frank Brown to Frank Johnson");
        fbperson.setName("Frank Johnson");
        personDao.updatePerson(fbperson);
        // load and print all persons from the database
        System.out.println("All persons in the database:");
        Collection<Person> newPersonList= personDao.getAllPersons();
        for (Person person : newPersonList){
            System.out.println(person.getId()+" - "+person.getName());
        }
    }
}
```

Name of the JPA Unit

JPA configuration

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="jpaunit">
    <class>domain.Person</class>
    <properties>
      <property name="hibernate.connection.driver_class" value="org.hsqldb.jdbcDriver"/>
      <property name="hibernate.connection.url"
        value="jdbc:hsqldb:hsql://localhost/trainingdb"/>
      <property name="hibernate.connection.username" value="SA"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <!-- Always drop and recreate the database schema on startup -->
      <property name="hibernate.hbm2ddl.auto" value="create"/>
      <!-- Show all SQL DML executed by Hibernate -->
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.use_sql_comments" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Create the database when we startup the application

Show the SQL that Hibernate sends to the database

JPA DAO with Spring 3

```
@Transactional
@Repository
public class EmployeeDAO implements IEmployeeDAO {
    private EntityManager entityManager;

    @PersistenceContext
    public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
    }
    public void addEmployee(Employee employee) {
        entityManager.persist(employee);
    }
    public void update(Employee employee) {
        entityManager.merge(employee);
    }
    public Employee load(int employeeNumber) {
        return entityManager.find(Employee.class, employeeNumber);
    }
    public void delete(Employee employee) {
        entityManager.remove(employee);
    }
    public Collection<Employee> getAllEmployees() {
        return entityManager.createQuery("from Employee").getResultList();
    }
}
```

@Repository tells Spring to translate all exceptions to Spring exceptions

Spring injects the entityManager

Spring 4 DAO interface

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
}
```

Entity
type

Id type

You only specify the interface, Spring generates the implementation with methods like

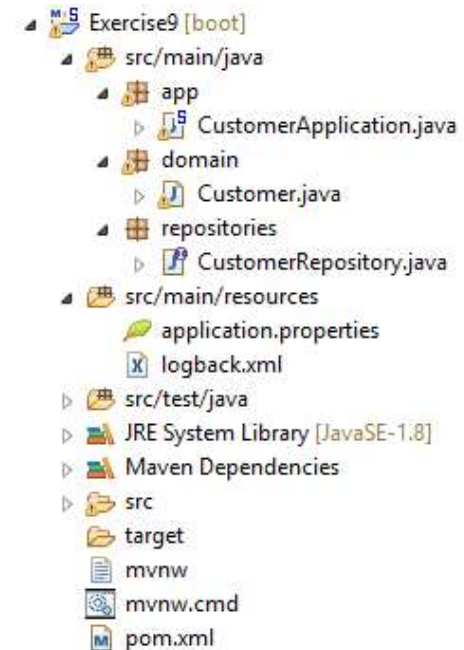
- save(Customer customer)
- delete(Customer customer)
- findAll()
- findOne(Long id)
- exists(Long id)

Example: The entity and the repository

@Entity

```
public class Customer {  
    @Id  
    @GeneratedValue  
    private long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    ...  
}
```

The Entity



```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    List<Customer> findByLastName(String lastName);  
}
```

Query method: Spring creates the implementation based on the method name

The application

Package where to find the repositories

Package where to find the entities

```
@SpringBootApplication
@EnableJpaRepositories("repositories")
@EntityScan("domain")
public class CustomerApplication implements CommandLineRunner{

    @Autowired
    CustomerRepository customerrepository;

    public static void main(String[] args) {
        SpringApplication.run(CustomerApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        customerrepository.save(new Customer("Jack", "Bauer", "jack@acme.com"));
        customerrepository.save(new Customer("Chloe", "O'Brian", "chloe@acme.com"));
    }
}
```

application.properties



```
spring.datasource.url=jdbc:hsqldb:hsqldb://localhost/trainingdb
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver

spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.HSQLDialect
```

Spring JPA libraries

```
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>  
  
<dependency>  
<groupId>org.hsqldb</groupId>  
<artifactId>hsqldb</artifactId>  
</dependency>
```

ENTITY CLASS MAPPING

Class Requirements

- JPA requires that entity classes have:
 - A field that can be used as identifier
 - A default constructor

```
@Entity
public class Customer {

    @Id
    @GeneratedValue
    private long id;
    private String firstName;
    private String lastName;
    private String email;

    protected Customer() {
    }

    public Customer(String firstName, String lastName, String email) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
}
```

An identifier

A default constructor

Annotation based Mapping

```
@Entity
@Table(name="MY_PERSON")
public class Person {
    @Id
    @Column(name="PERSON_ID")
    private long id;
    @Column(name="FULLNAME")
    private String name;

    public Person() {}
    public Person(String name) { this.name = name; }

    public long getId() { return id; }
    private void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Map class to 'MY_PERSON' table

Map to PERSON_ID column

Map to FULLNAME column

Table: MY_PERSON

PERSON_ID	FULLNAME
1	Frank Brown
2	John Smith

Defaults to 'Person' table

```
@Entity
public class Person {
    @Id
    private long id;
    private String name;
    ...
}
```

Defaults to 'id' column
name (same as property)

No annotation needed, persisted
to the 'name' column by default

Table: PERSON

ID	NAME
1	Frank Brown
2	John Smith

MAPPING IDENTITY

Primary key

- A primary key is
 - Unique
 - No duplicate values
 - Constant
 - Value never changes
 - Required
 - Value can never be null
- Primary key types:
 - Natural key
 - Has a meaning in the business domain
 - Surrogate key
 - Has no meaning in the business domain
 - Best practice



Mapping Primary Keys

- Object / Relational mismatch
 - Hibernate requires you to specify the property that will map to the primary key
- Prefer surrogate keys
 - Natural keys often lead to a brittle schema

```
@Entity
public class Person {
    @Id
    private String name;

    ...
}
```

Name as a natural primary key for Person can give problems

```
@Entity
public class Person {
    @Id
    private long id;
    private String name;

    ...
}
```

Instead use id as a surrogate key for Person

Generating Identity

- Generated identity values
 - Ensure identity uniqueness
- Private setId() methods
 - Ensure identity immutability

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;

    public Person() {}
    public Person(String name) { this.name = name; }

    public long getId() { return id; }
    private void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Id is generated

Id can not be set by the application

Generation Strategies



JPA	Description
AUTO	Selects the best strategy for your database
IDENTITY	Use an identity column (MS SQL, <u>MySQL</u> , HSQL, ...)
SEQUENCE	Use a sequence (Oracle, <u>PostgreSQL</u> , SAP DB, ...)
TABLE	Uses a table to hold last generated values for PKs

Specifying Identity Generation

■ @GeneratedValue

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    private String name;
```

Specify the generation strategy

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;
```

Defaults to 'AUTO' when not specified

Identity Column


- Identity columns are columns that can automatically generate the next unique id

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    private String name;
```

JPA identity strategy

- If your database support identity columns the native strategy will default to using them

Sequences

- 
- By default Hibernate only uses a single sequence called 'hibernate-sequence'
 - You can specify additional custom sequences

Using Custom Sequences

Create Custom Sequence

```
@Entity
@SequenceGenerator(name="personSeq", sequenceName="PERSON_SEQUENCE")
public class Person_annotated_sequence {
    @Id
    @GeneratedValue(generator="personSeq")
    private long id;
    ...
}
```

Use Custom Sequence

```
<hibernate-mapping>
  <class name="identity.Person">
    <id name="id" >
      <generator class="sequence">
        <param name="sequence">PERSON_SEQUENCE</param>
      </generator>
    </id>
    <property name="name" />
  </class>
</hibernate-mapping>
```

Use 'sequence' strategy

Specify the custom sequence

Main point

- JPA is a layer on top of JDBC that makes database access much simpler. Spring JPA is a layer on top of JPA that makes data access even more simpler.

Science of Consciousness: The intelligence of pure consciousness is available to every human being. By daily transcending to this pure consciousness, one gains more and more support of Nature.

Connecting the parts of knowledge with the wholeness of knowledge

1. JDBC gives developers full control of database access but results in tight coupling between the application and the database
2. JPA unites the differences between the OO model and the relational model

-
3. **Transcendental consciousness** is where all differences are united in infinite harmony.
 4. **Wholeness moving within itself:** In Unity Consciousness, one realizes that both the diversity of life, and its underlying unity are nothing but the Self.

