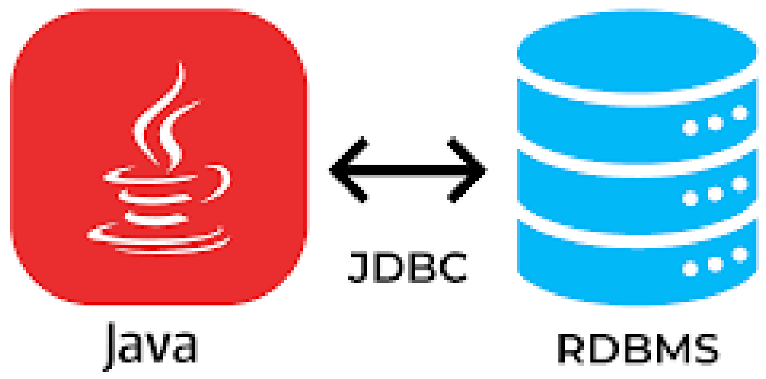
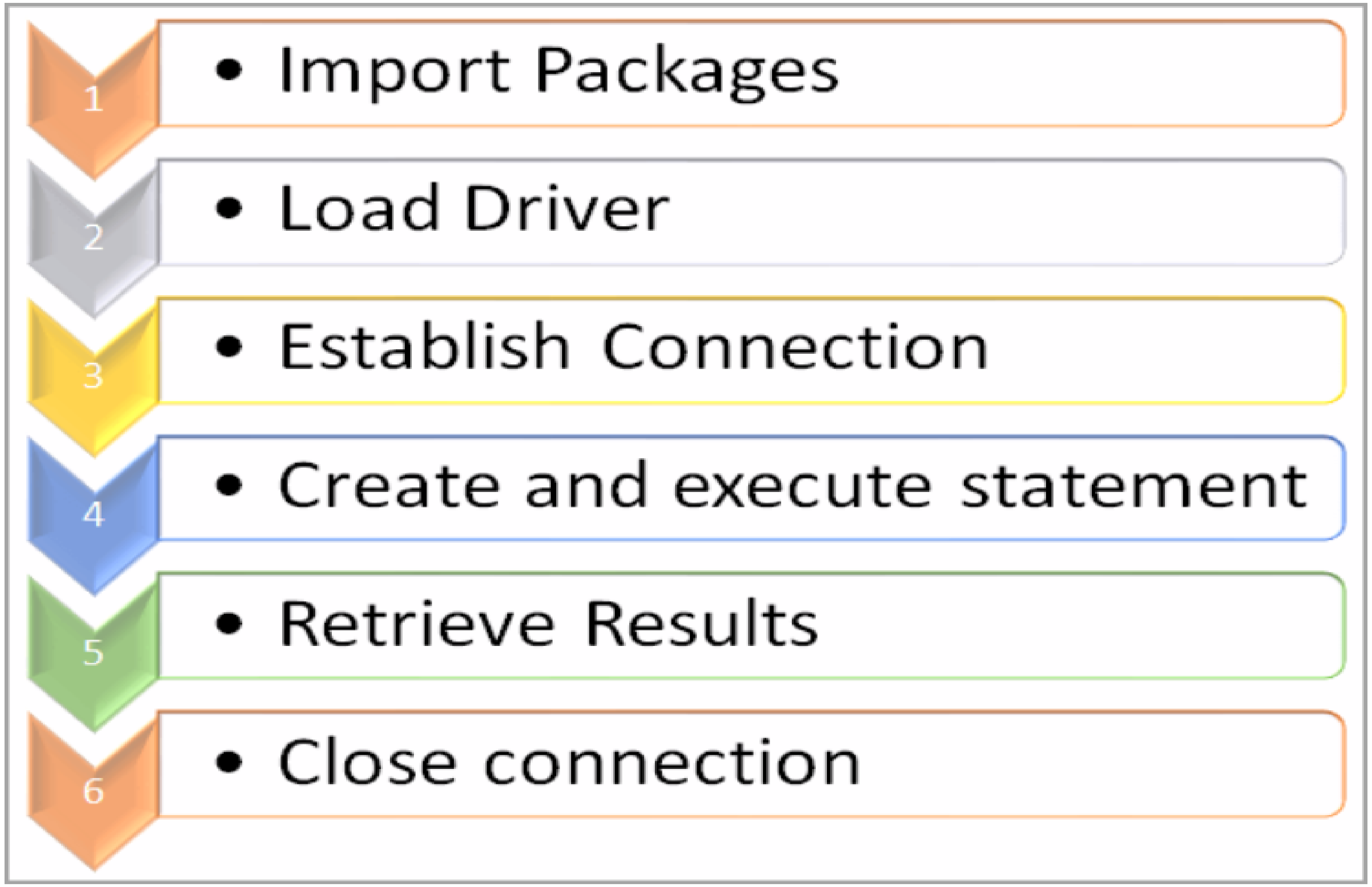

Java
JDBC
RDBMS

Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database.

It is a Java-based data access technology used for Java database connectivity.

JDBC Connection Steps

There are 6 basic steps to connect with JDBC. They are enlisted in the below image:



1. Import Packages
2. Load Driver
3. Establish Connection
4. Create and execute statement
5. Retrieve Results
6. Close connection

```java
// import sql package to use it in our program
import java.sql.*;

public class Sample_JDBC_Program {

public static void main(String[] args) throws ClassNotFoundException, SQLException {
    // store the SQL statement in a string
    String QUERY = "select * from employee_details";
    //register the oracle driver with DriverManager
    Class.forName("oracle.jdbc.driver.OracleDriver");
    //Here we have used Java 8 so opening the connection in try statement
    try(Connection conn = DriverManager.getConnection("jdbc:oracle:thin:system/pass123@localh
    {
            Statement statemnt1 = conn.createStatement();
            //Created statement and execute it
            ResultSet rs1 = statemnt1.executeQuery(QUERY);
            {
                //Get the values of the record using while loop
                while(rs1.next())
                {
                    int empNum = rs1.getInt("empNum");
                    String lastName = rs1.getString("lastName");
                    String firstName = rs1.getString("firstName");
                    String email = rs1.getString("email");
                    String deptNum = rs1.getString("deptNum");
                    String salary = rs1.getString("salary");
                    //store the values which are retrieved using ResultSet and print it
                System.out.println(empNum + "," +lastName+ "," +firstName+ "," +email +","+de
                }
            }
    }
    catch (SQLException e) {
        //If exception occurs catch it and exit the program
        e.printStackTrace();
    }
   }
}
```

spring

JDBC

Java Database Connectivity

Spring JDBC are divided into four separate packages:

- ✓ core — the core functionality of JDBC. Some of the important classes under this package include JdbcTemplate, SimpleJdbcInsert, SimpleJdbcCall and NamedParameterJdbcTemplate.

- ✓ datasource — utility classes to access a data source. It also has various data source implementations for testing JDBC code outside the Jakarta EE container.

- ✓ object — DB access in an object-oriented manner. It allows running queries and returning the results as a business object. It also maps the query results between the columns and properties of business objects.

- ✓ support — support classes for classes under core and object packages, e.g., provides the SQLException translation functionality

```
@Configuration
@ComponentScan(“demo.boot.dao")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource =
                    new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUsername("guest_user");
        dataSource.setPassword("guest_password");

        return dataSource;
    }
}
```

The JDBC template is the main API through which we'll access most of the functionality that we're interested in:

- ✓ creation and closing of connections
- ✓ running statements and stored procedure calls
- ✓ iterating over the ResultSet and returning results

First, let’s start with a simple example to see what the JdbcTemplate can do:

```
int result = jdbcTemplate.queryForObject(
    "SELECT COUNT(*) FROM EMPLOYEE", Integer.class);
```

And here's a simple INSERT:

```
public int addEmplyee(int id) {
    return jdbcTemplate.update(
      "INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)", id, “venkata",
“Ramana", “INDIA");}
```

Queries With Named Parameters

To get support for named parameters, we'll use the other JDBC template provided by the framework — the NamedParameterJdbcTemplate.

```
SqlParameterSource namedParameters = new
MapSqlParameterSource().addValue("id", 1);

return namedParameterJdbcTemplate.queryForObject(
  "SELECT FIRST_NAME FROM EMPLOYEE WHERE ID = :id",
namedParameters, String.class);
```

BeanPropertySqlParameterSource implementations instead of specifying the named parameters manually like before.

```
Employee employee = new Employee();
employee.setFirstName("surya");

String SELECT_BY_ID = "SELECT COUNT(*) FROM
EMPLOYEE WHERE FIRST_NAME = :firstName";

SqlParameterSource namedParameters = new
BeanPropertySqlParameterSource(employee);
return namedParameterJdbcTemplate.queryForObject(
  SELECT_BY_ID, namedParameters, Integer.class);
```

Mapping Query Results to Java Object

Another very useful feature is the ability to map query results to Java objects by implementing the RowMapper interface.

For example, for every row returned by the query, Spring uses the row mapper to populate the java bean:

```
public class EmployeeRowMapper implements RowMapper<Employee> {
   @Override
   public Employee mapRow(ResultSet rs, int rowNum) throws
SQLException {
      Employee employee = new Employee();

      employee.setId(rs.getInt("ID"));
      employee.setFirstName(rs.getString("FIRST_NAME"));
      employee.setLastName(rs.getString("LAST_NAME"));
      employee.setAddress(rs.getString("ADDRESS"));

      return employee;
   }
}
```

we can now pass the row mapper to the query API and get fully populated Java objects:

```
String query = "SELECT * FROM EMPLOYEE WHERE ID = ?";
Employee employee = jdbcTemplate.queryForObject(
  query, new Object[] { id }, new EmployeeRowMapper());
```

Exception Translation

Spring comes with its own data exception hierarchy out of the box — with DataAccessException as the root exception — and it translates all underlying raw exceptions to it.

So, we keep our sanity by not handling low-level persistence exceptions. We also benefit from the fact that Spring wraps the low-level exceptions in DataAccessException or one of its sub-classes.

This also keeps the exception handling mechanism independent of the underlying database we are using.

Besides the default SQLErrorCodeSQLExceptionTranslator, we can also provide our own implementation of SQLExceptionTranslator.

Customizing the error message when there is a duplicate key violation, which results in error code 23505 when using H2:

```
public class CustomSQLErrorCodeTranslator extends
SQLErrorCodeSQLExceptionTranslator {
    @Override
    protected DataAccessException
      customTranslate(String task, String sql, SQLException sqlException)
{
        if (sqlException.getErrorCode() == 23505) {
          return new DuplicateKeyException(
            "Custom Exception translator - Integrity constraint violation.",
sqlException);
        }
        return null;      }  }
```

SimpleJdbcInsert

Let's take a look at running simple insert statements with minimal configuration.

The INSERT statement is generated based on the configuration of SimpleJdbcInsert.

```
SimpleJdbcInsert simpleJdbcInsert =
  new SimpleJdbcInsert(dataSource).withTableName("EMPLOYEE");
```

Next, let's provide the Column names and values, and run the operation:

```
public int addEmplyee(Employee emp) {
    Map<String, Object> parameters = new HashMap<String,
Object>();
    parameters.put("ID", emp.getId());
    parameters.put("FIRST_NAME", emp.getFirstName());
    parameters.put("LAST_NAME", emp.getLastName());
    parameters.put("ADDRESS", emp.getAddress());

    return simpleJdbcInsert.execute(parameters);
```

we can use the executeAndReturnKey() API to allow the database to generate the primary key. We'll also need to configure the actual auto-generated column:

```
SimpleJdbcInsert simpleJdbcInsert = new
SimpleJdbcInsert(dataSource)
                                        .withTableName("EMPLOYEE")

.usingGeneratedKeyColumns("ID");

Number id =
simpleJdbcInsert.executeAndReturnKey(parameters);
System.out.println("Generated id - " + id.longValue());
```

Stored Procedures With SimpleJdbcCall

```
SimpleJdbcCall simpleJdbcCall = new
SimpleJdbcCall(dataSource)

.withProcedureName("READ_EMPLOYEE");
public Employee getEmployeeUsingSimpleJdbcCall(int id) {
   SqlParameterSource in = new
MapSqlParameterSource().addValue("in_id", id);
   Map<String, Object> out = simpleJdbcCall.execute(in);

   Employee emp = new Employee();
   emp.setFirstName((String) out.get("FIRST_NAME"));
   emp.setLastName((String) out.get("LAST_NAME"));

   return emp;
}
```

Basic Batch Operations Using JdbcTemplate

```
public int[] batchUpdateUsingJdbcTemplate(List<Employee>
employees) {
    return jdbcTemplate.batchUpdate("INSERT INTO EMPLOYEE
VALUES (?, ?, ?, ?)",
        new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i)
throws SQLException {
                ps.setInt(1, employees.get(i).getId());
                ps.setString(2, employees.get(i).getFirstName());
                ps.setString(3, employees.get(i).getLastName());
                ps.setString(4, employees.get(i).getAddress();
            }
            @Override
            public int getBatchSize() {
                return 50;
            }
        });
}
```

Batch Operations Using NamedParameterJdbcTemplate

We can pass the parameter values can be passed to the batchUpdate() method as an array of SqlParameterSource.

```
SqlParameterSource[] batch =
SqlParameterSourceUtils.createBatch(employees.toArray());
int[] updateCounts =
namedParameterJdbcTemplate.batchUpdate(
    "INSERT INTO EMPLOYEE VALUES (:id, :firstName,
:lastName, :address)", batch);
return updateCounts;
```

Spring JDBC With Spring Boot

Spring Boot provides a starter spring-boot-starter-jdbc for using JDBC with relational databases.

As with every Spring Boot starter, this one helps us get our application up and running quickly.

Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Configuration

Spring Boot configures the data source automatically for us. We just need to provide the properties in a properties file:

```
spring.datasource.url=jdbc:mysql://localhost:3306/springjdbc
spring.datasource.username=root
spring.datasource.password=root
```

SPRING

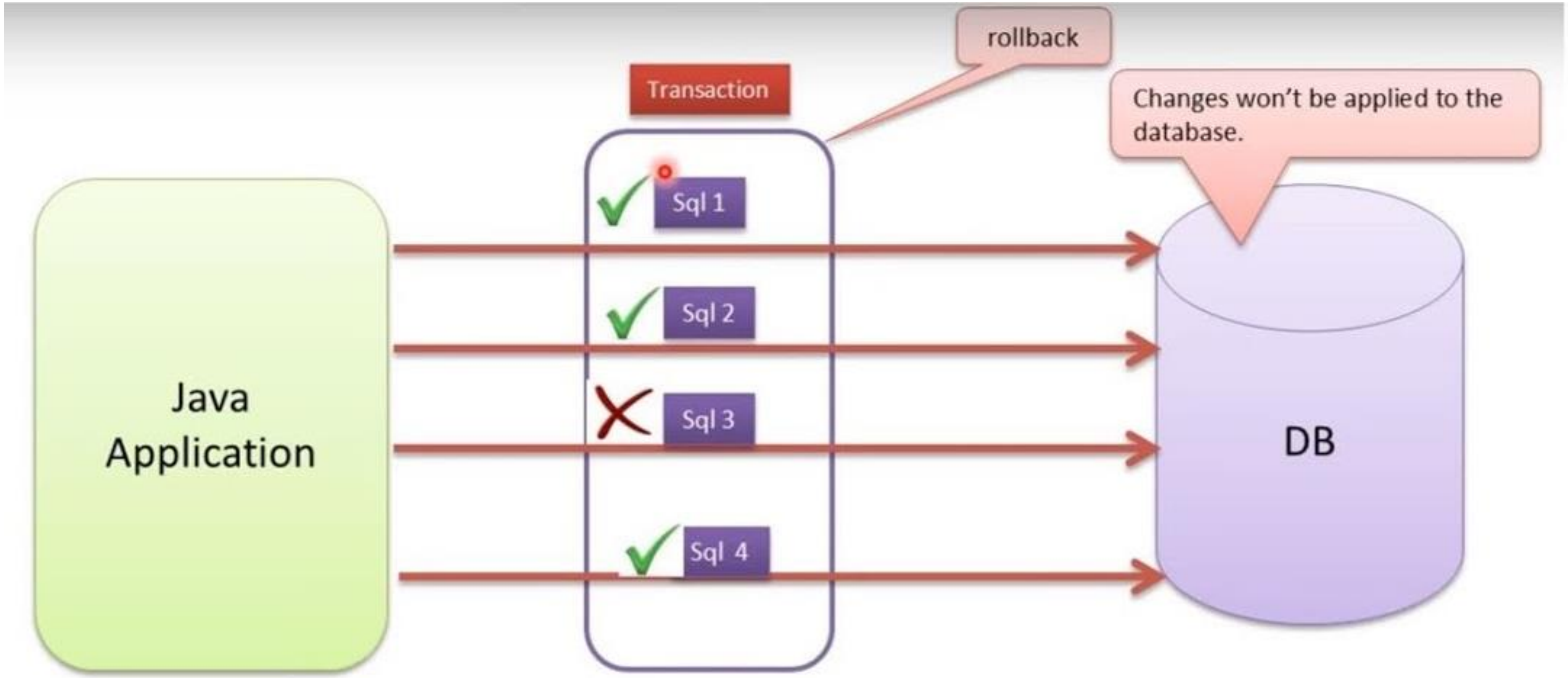
TRANSACTIONS

What Is a Transaction?

Transactions in Java, as in general refer to a series of actions that must all complete successfully. Hence, if one or more action fails, all other actions must back out leaving the state of the application unchanged. It follows
ACID properties.

Transactions are either bean-manged or container-managed.

There are two types of transactions:
Local Transaction
Global/Distributed/XA Trasaction


rollback
Transaction
Changes won't be applied to the database.
Sql 1
Sql 2
Sql 3
Sql 4
Java Application
DB

The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO).

- Support for declarative transaction management.

- Simpler API for programmatic transaction management than complex transaction APIs such as JTA.

- Excellent integration with Spring’s data access abstractions.

@EnableTransactionManagement annotation that we can use in a @Configuration class to enable transactional support:

```
@Configuration
@EnableTransactionManagement
public class PersistenceJPAConfig{

  @Bean
   public PlatformTransactionManager transactionManager(){
      JpaTransactionManager transactionManager
        = new JpaTransactionManager();
      transactionManager.setEntityManagerFactory(
        entityManagerFactoryBean().getObject() );
      return transactionManager;
   }
```

@Transactional Annotation

With transactions configured, we can now annotate a bean with @Transactional either at the class or method level:

```
@Service
@Transactional
public class FooService {
    //...
}
```

The annotation supports further configuration as well:

- ✓ the Propagation Type of the transaction
- ✓ the Isolation Level of the transaction
- ✓ a Timeout for the operation wrapped by the transaction
- ✓ a readOnly flag – a hint for the persistence provider that the transaction should be read only
- ✓ the Rollback rules for the transaction

Transactions and Proxies

At a high level, Spring creates proxies for all the classes annotated with @Transactional, either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method, mainly for starting and committing the transaction.

Changing the Isolation Level

```
@Transactional(isolation = Isolation.SERIALIZABLE)
```

Read-Only Transactions

```
@Transactional( propagation = Propagation.SUPPORTS,readOnly = true )
```

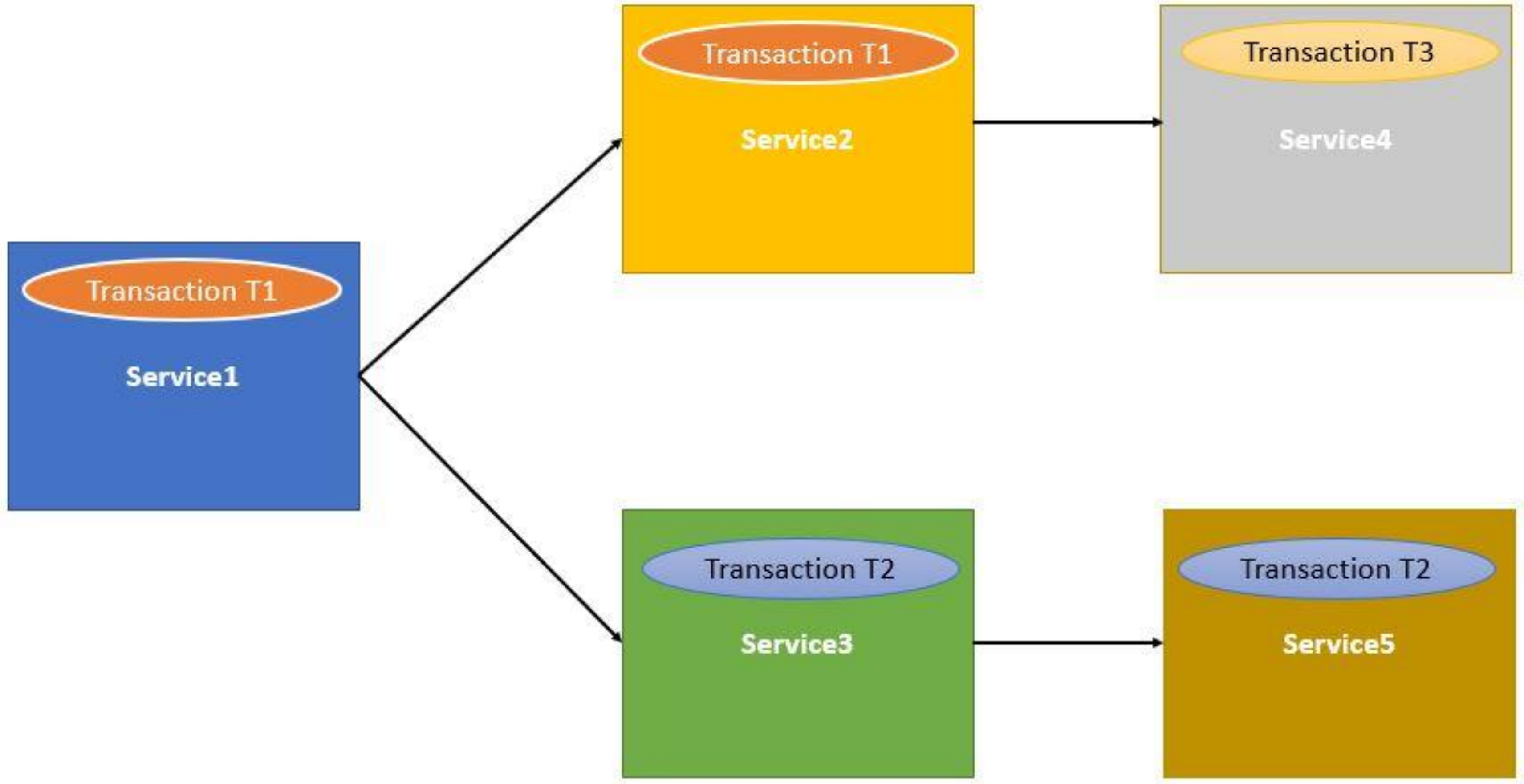
Transaction Rollback

```
@Transactional(noRollbackFor = { SQLException.class })
public void createCourseDeclarativeWithNoRollBack(Course course) throws
SQLException {
    courseDao.create(course);
    throw new SQLException("Throwing exception for demoing rollback");
}
```

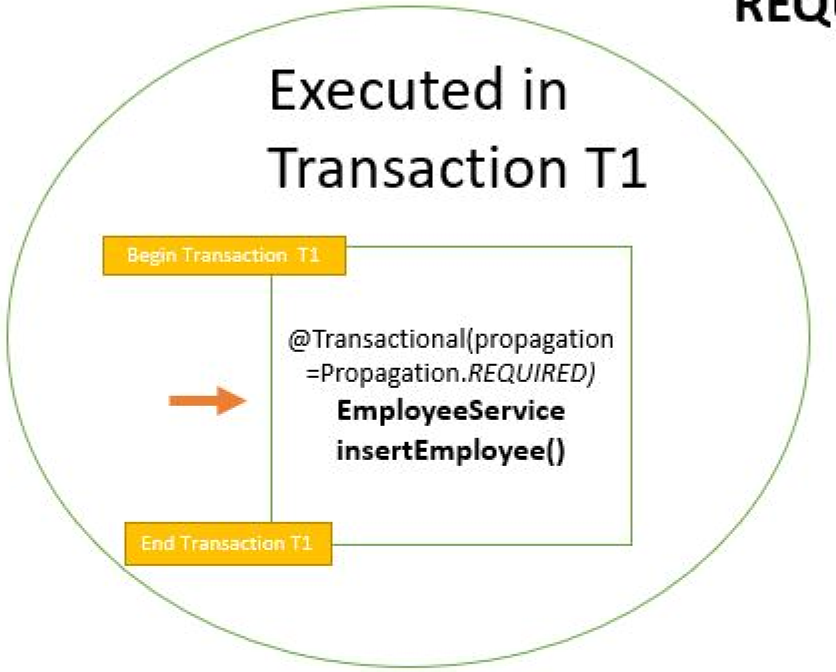
There are six types of Transaction Propagations:

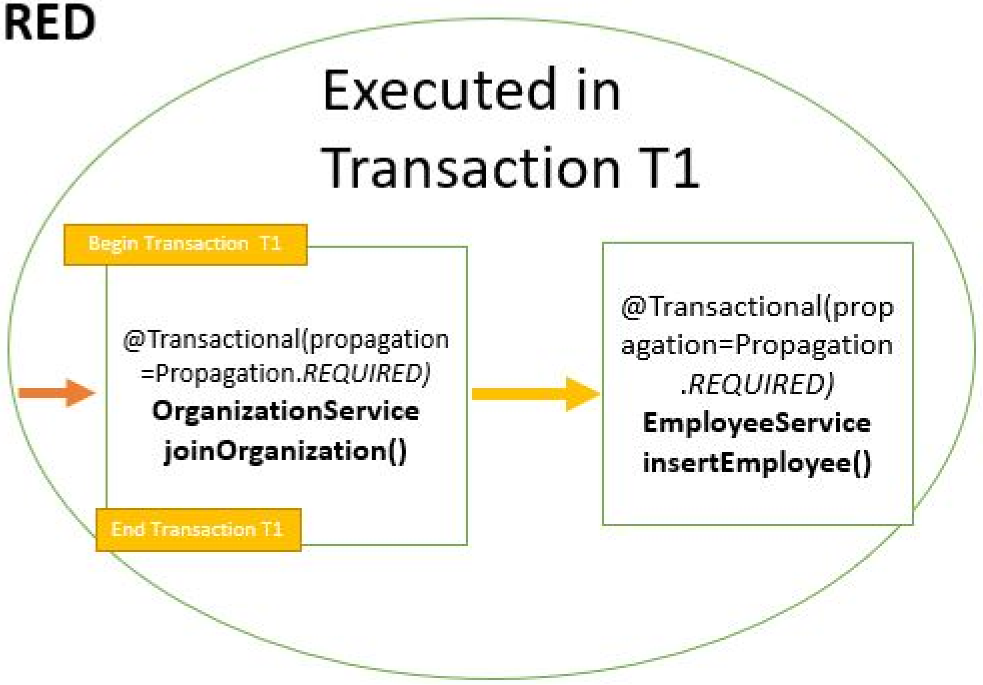
- ✓ REQUIRED
- ✓ SUPPORTS
- ✓ NOT_SUPPORTED
- ✓ REQUIRES_NEW
- ✓ NEVER
- ✓ MANDATORY

TRANSACTION PROPAGATION



TRANSACTION PROPAGATION-REQUIRED


Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.*REQUIRED)*
EmployeeService insertEmployee()
End Transaction T1


Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.*REQUIRED)*
OrganizationService joinOrganization()
End Transaction T1
@Transactional(propagation=Propagation .*REQUIRED)*
EmployeeService insertEmployee()

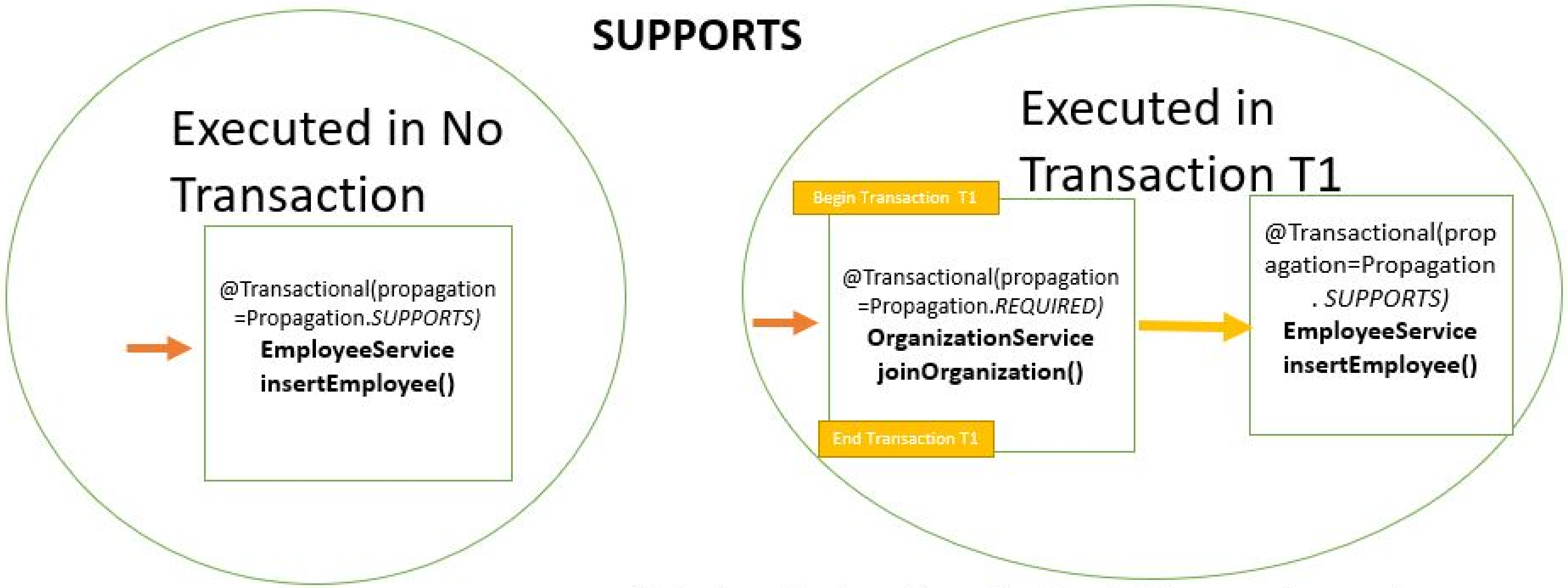
If the insertEmployee() method **is called directly** it creates its **own new transaction**.

If the insertEmployee() **method is called from another service –**

1. If the calling service has a transaction then method **makes use of the existing transaction**.
2. If the calling service does not have a transaction then the method **will create new transaction**.

So in the case of **REQUIRED** the insertEmployee() method makes use of the calling service transaction if it exists else creates its own.

TRANSACTION PROPAGATION- SUPPORTS


Executed in No Transaction
@Transactional(propagation =Propagation.SUPPORTS)
EmployeeService
insertEmployee()
Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.REQUIRED)
OrganizationService
joinOrganization()
End Transaction T1
@Transactional(propagation=Propagation. SUPPORTS)
EmployeeService
insertEmployee()

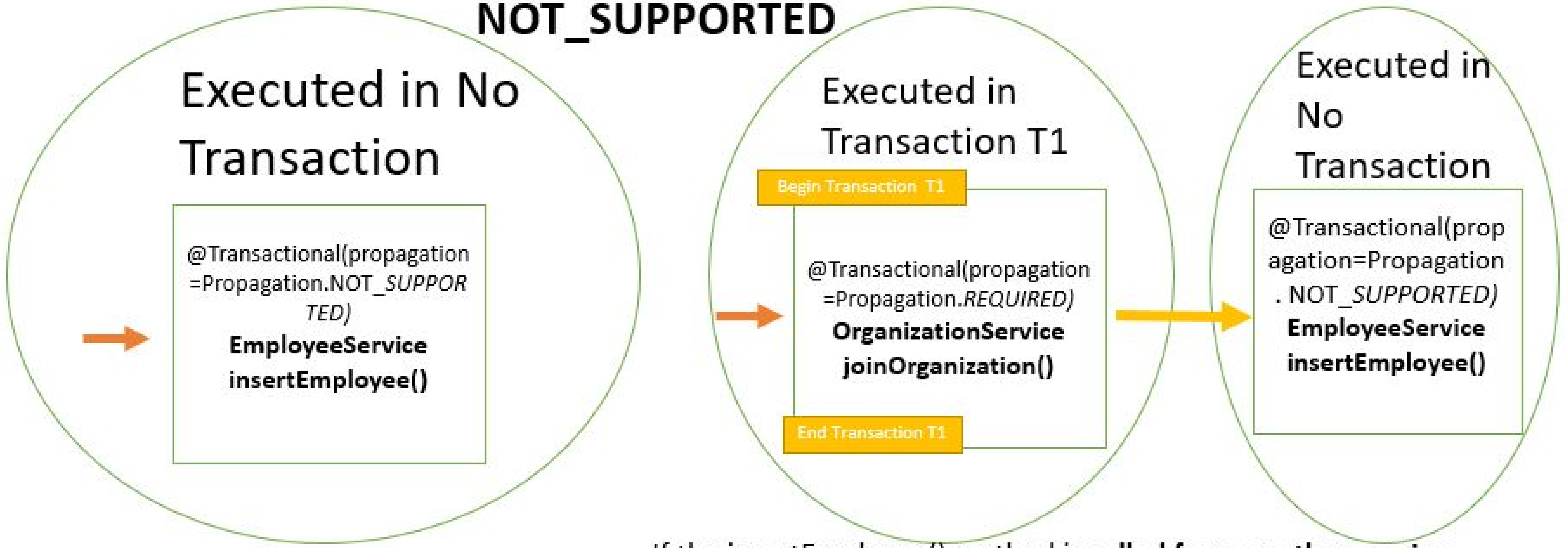
If the insertEmployee() method **is called directly** it does **not create own new transaction**

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **makes use of the existing transaction**.
2. If the calling service method does not have a transaction then the method **will not create a new transaction**.

So in the case of **SUPPORTS** the insertEmployee() method will create make use of calling service transaction if it exists. Else it will not create a new transaction but run without transaction.

TRANSACTION PROPAGATION- NOT_SUPPORTED


Executed in No Transaction
@Transactional(propagation =Propagation.NOT_SUPPORTED)
EmployeeService insertEmployee()
Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.REQUIRED)
OrganizationService joinOrganization()
End Transaction T1
Executed in No Transaction
@Transactional(propagation=Propagation. NOT_SUPPORTED)
EmployeeService insertEmployee()

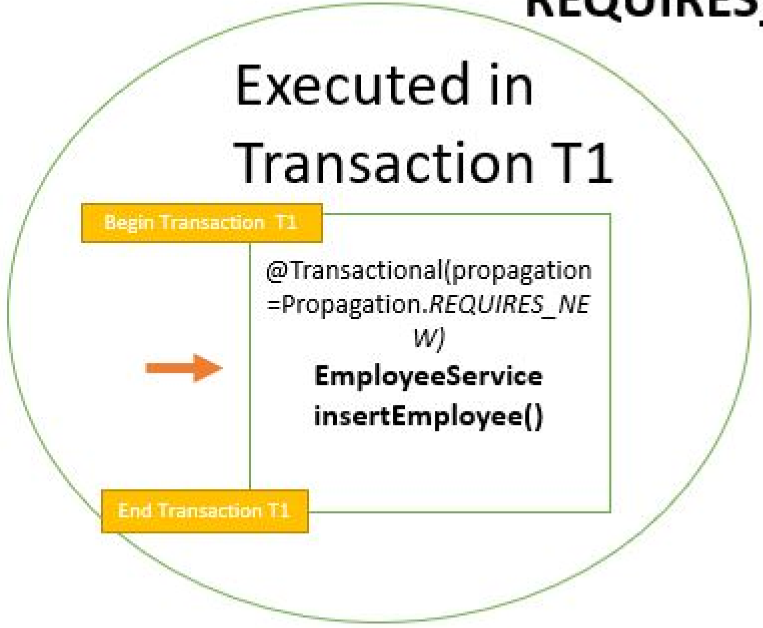
If the insertEmployee() method is **called directly** it does **not create own new transaction**

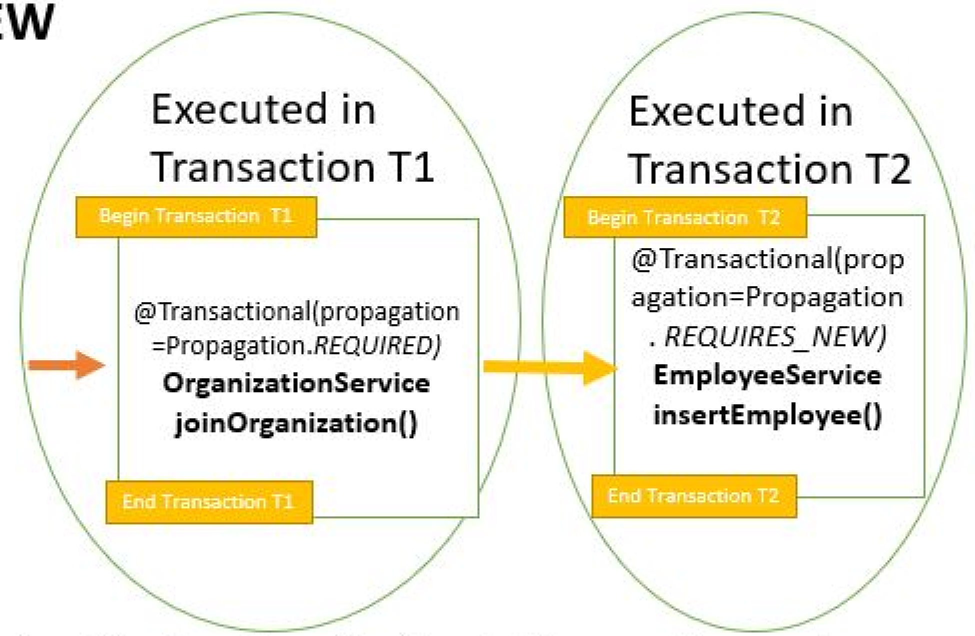
If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **does not make use of the existing transaction neither does it create its own transaction. It run without transaction**.
2. If the calling service method does not have a transaction then the method **will not create a new transaction and run without transaction**.

So in the case of **NOT_SUPPORTED** the insertEmployee() method never run in transaction.

TRANSACTION PROPAGATION- REQUIRES_NEW


Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.*REQUIRES_NEW)*
EmployeeService insertEmployee()
End Transaction T1


Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.*REQUIRED)*
OrganizationService joinOrganization()
End Transaction T1
Executed in Transaction T2
Begin Transaction T2
@Transactional(propagation=Propagation. *REQUIRES_NEW)*
EmployeeService insertEmployee()
End Transaction T2

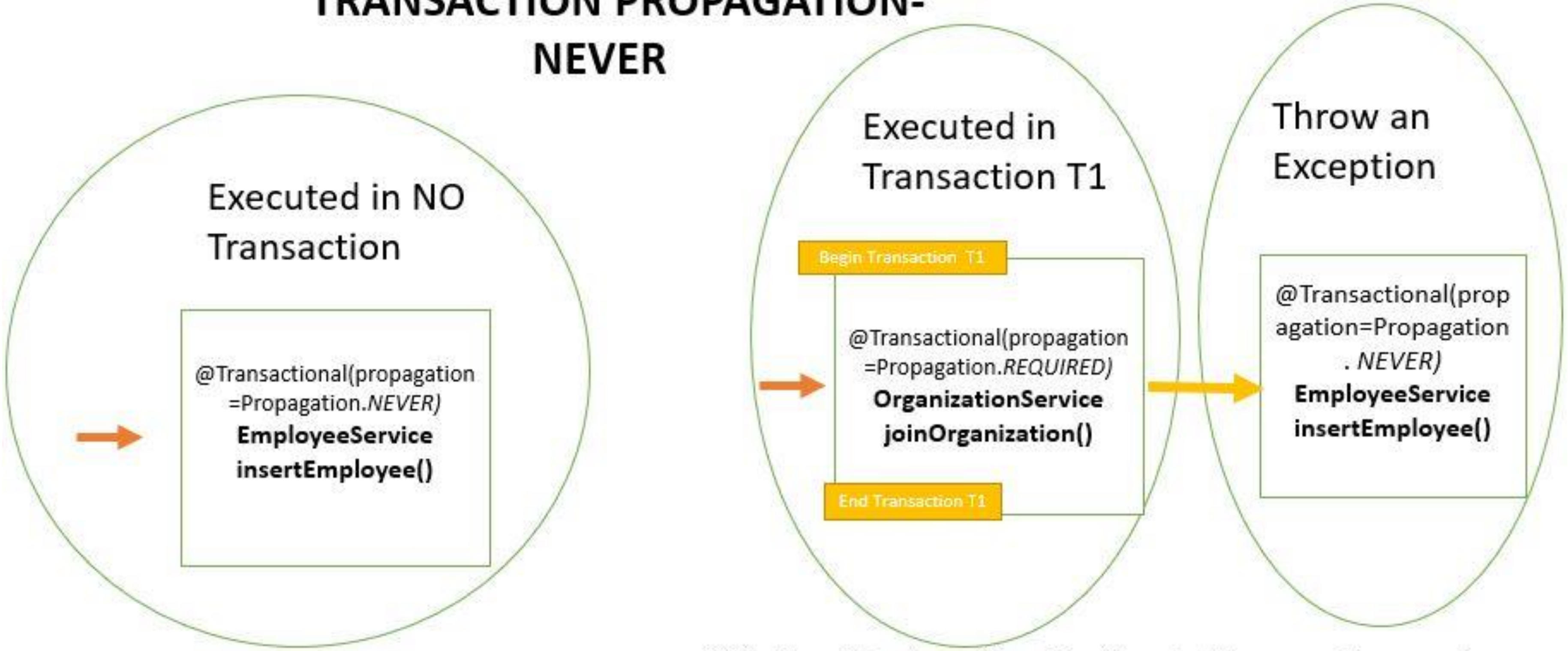
If the insertEmployee() method **is called directly** it creates its **own new transaction**

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **does not make use of the existing transaction but creates its own transaction**.
2. If the calling service method does not have a transaction the method **will create a new transaction**

So in the case of **REQUIRES_NEW** the insertEmployee() method always creates a new Transaction

TRANSACTION PROPAGATION- NEVER


Executed in NO Transaction
@Transactional(propagation =Propagation.*NEVER)*
EmployeeService insertEmployee()
Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.*REQUIRED)*
OrganizationService joinOrganization()
End Transaction T1
Throw an Exception
@Transactional(propagation=Propagation. *NEVER)*
EmployeeService insertEmployee()

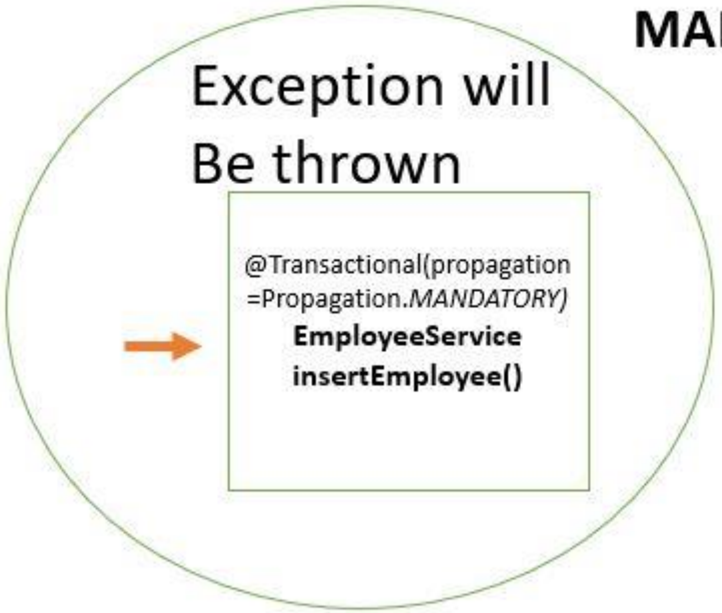
If insertEmployee() method **is called directly** it creates it does not create a new transaction

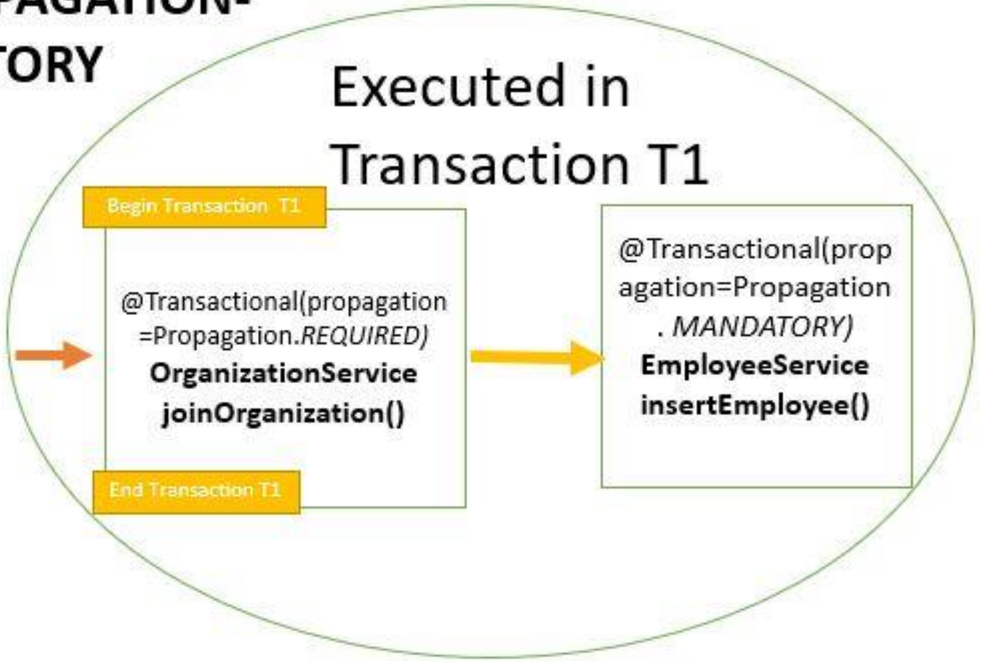
If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **throws an exception.**
2. If the calling service method does not have a transaction the method **will not create a new one and run without transaction.**

So in the case of **NEVER** the insertEmployee() method never uses Transaction

TRANSACTION PROPAGATION-MANDATORY


Exception will Be thrown
@Transactional(propagation =Propagation.*MANDATORY)*
EmployeeService insertEmployee()


Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.*REQUIRED)*
OrganizationService joinOrganization()
End Transaction T1
@Transactional(propagation=Propagation. *MANDATORY)*
EmployeeService insertEmployee()

If insertEmployee() method is **called directly** it will **throw an exception**.

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **makes use of the existing transaction**.
2. If the calling service method does not have a transaction the method **will throw an exception**

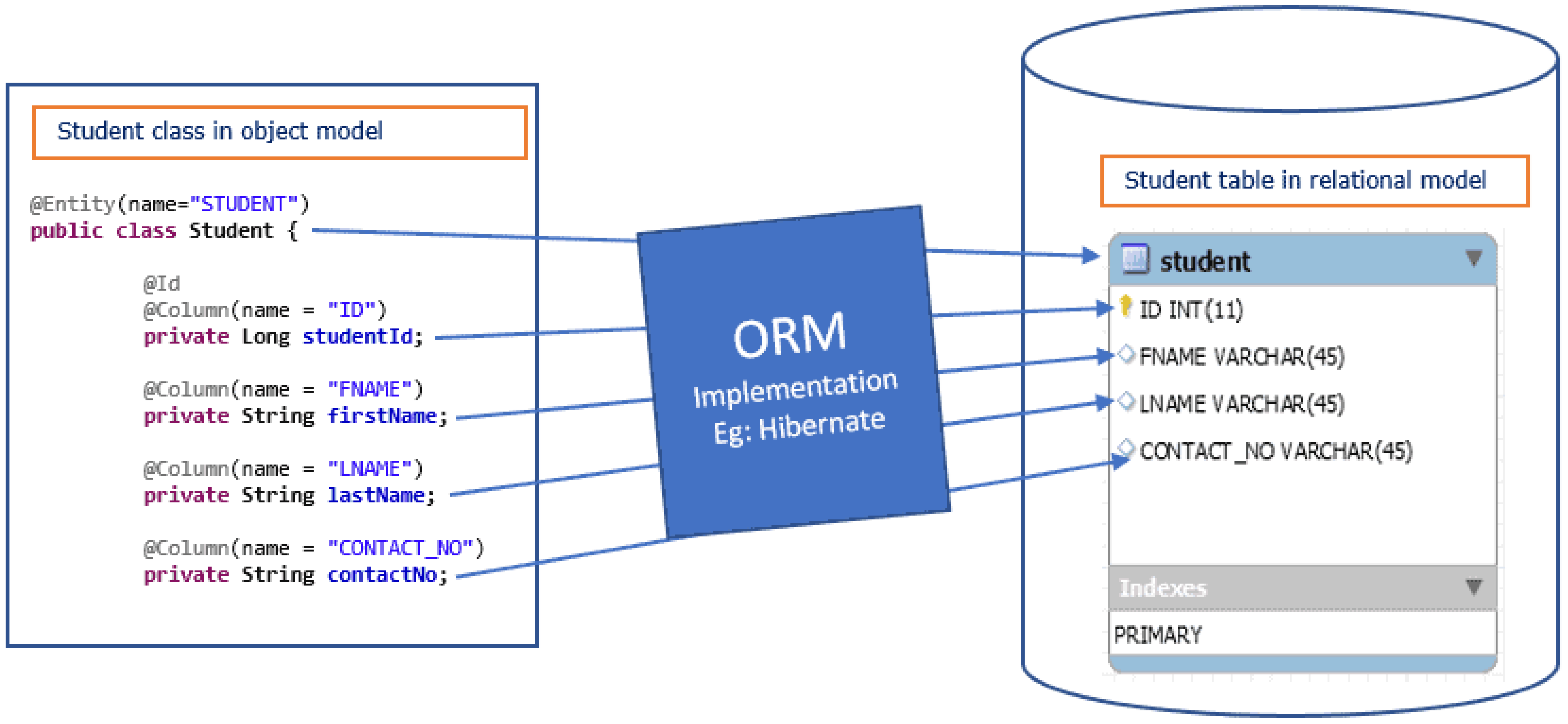
So in the case of **MANDATORY** the insertEmployee() method always needs the calling service to have a transaction else exception is thrown

Data Access Layer

What is Object Relational Mapping (ORM)?

ORM or Object Relational Mapping is a system that implements the responsibility of mapping the Object to Relational Model.

That means it is responsible to store Object Model data into Relational Model and further read the data from Relational Model into Object Model.



```
@Entity(name="STUDENT")
public class Student {

        @Id
        @Column(name = "ID")
        private Long studentId;

        @Column(name = "FNAME")
        private String firstName;

        @Column(name = "LNAME")
        private String lastName;

        @Column(name = "CONTACT_NO")
        private String contactNo;
```

```
student
ID INT(11)
FNAME VARCHAR(45)
LNAME VARCHAR(45)
CONTACT_NO VARCHAR(45)
Indexes
PRIMARY
```

Student class in object model
ORM
Implementation
Eg: Hibernate
Student table in relational model

ORM implements responsibility of mapping the Object to Relational Model.

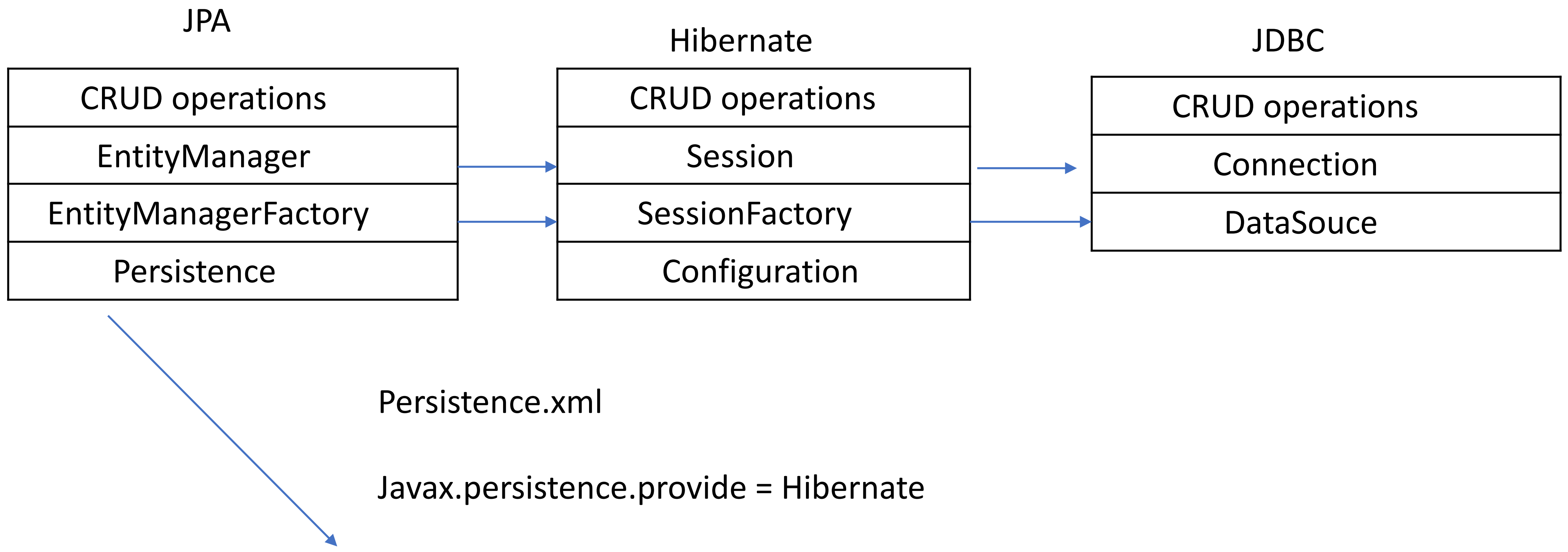
Popular ORM tools/frameworks in Java:

Hibernate – Open Source
Top Link – By Oracle
Eclipse Link – Eclipse Persistence Platform
Open JPA – By Apache
MyBatis – Open Source – Formerly known as iBATIS

Java Persistence API

- JPA is an API
  - Implemented by a persistence provider
  - Like JDBC is an API implemented by
- Some persistence providers
  - Hibernate from JBoss
  - TopLink from Oracle

JPA / Hibernate / JDBC



Persistence.xml

Javax.persistence.provide = Hibernate

+

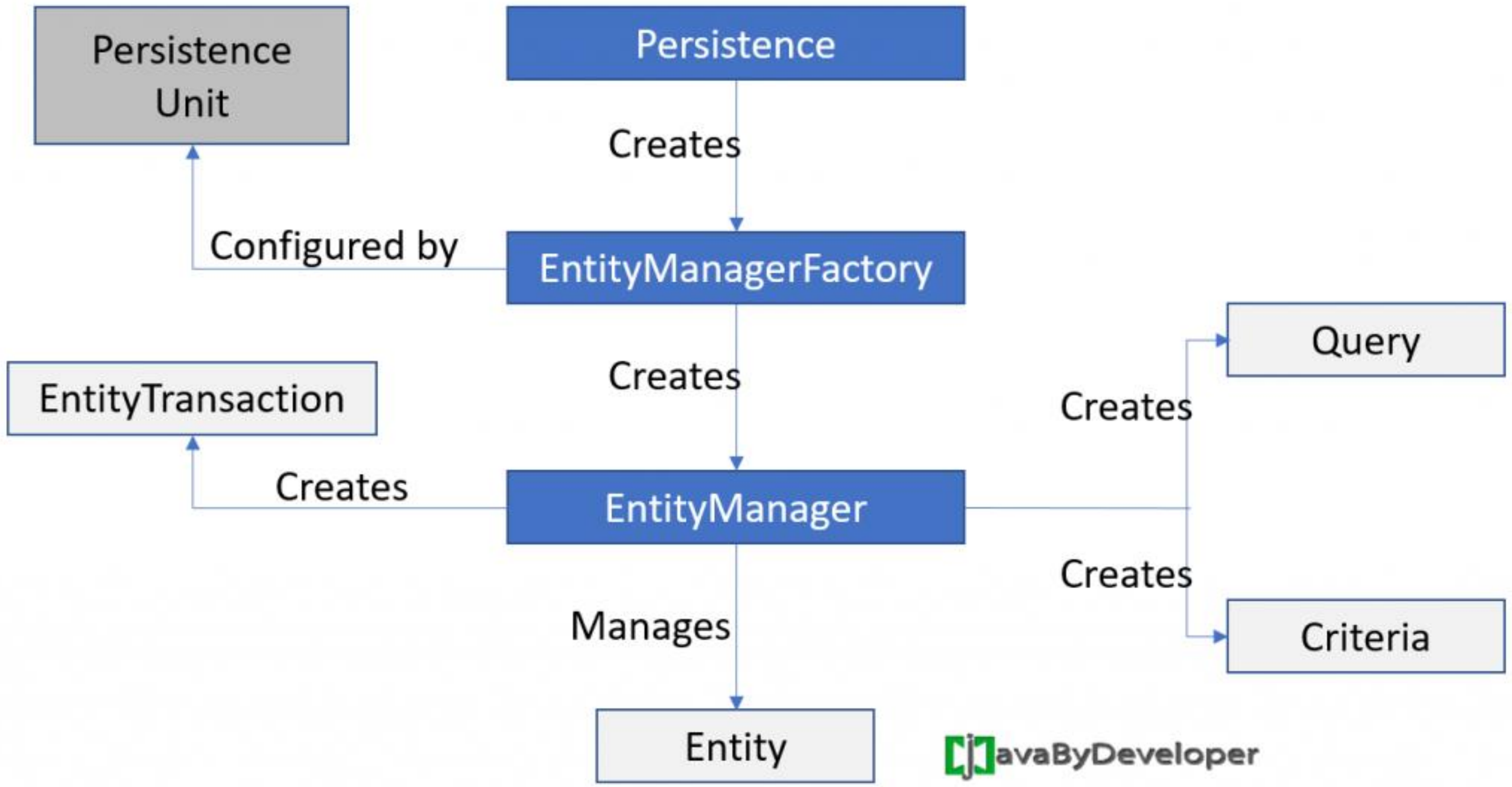
Hibernate-jpa.jar

```
@Entity
Class User
{
Long id;
String name;
....
getName(){}

setName(){}
}
```

```
interface  IUserDao
{

List<User>  getUsers();

User getUser(long id);

void updateUser(User user);

void deleteUser(User user);

}
```

```
@Repository
Class  UserDao   implements  IUserDao
{

@Autowired
EntityManager  manager;


List<User>  getUsers()
{   return manager.createQuery("from User"); }

User getUser(long id)
{ return  manager.find(User.class, id); }

void updateUser(User user)  { ... }

void deleteUser(User user)   { ... }

}
```

JPA Architecture:



Entities

An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented through either persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Requirements for Entity Classes

- ✓ An entity class must follow these requirements.
- ✓ The class must be annotated with the javax.persistence.Entity annotation.
- ✓ The class must have a public or protected, no-argument constructor. The class may have other constructors.
- ✓ The class must not be declared final. No methods or persistent instance variables must be declared final.
- ✓ If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the Serializable interface.
- ✓ Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- ✓ Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

```
@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet();
    ...
}
```

Direction in Entity Relationships

The direction of a relationship can be either bidirectional or unidirectional.

A bidirectional relationship has both an owning side and an inverse side.

A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

Cascade Operations and Relationships

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade delete relationship.

Cascade Operation	Description
`ALL`	All cascade operations will be applied to the parent entity's related entity. `All` is equivalent to specifying `cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}`
`DETACH`	If the parent entity is detached from the persistence context, the related entity will also be detached.
`MERGE`	If the parent entity is merged into the persistence context, the related entity will also be merged.
`PERSIST`	If the parent entity is persisted into the persistence context, the related entity will also be persisted.
`REFRESH`	If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.
`REMOVE`	If the parent entity is removed from the current persistence context, the related entity will also be removed.

Managing Entities

Entities are managed by the entity manager, which is represented by javax.persistence.EntityManager instances.

Each EntityManager instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store.

A persistence context defines the scope under which particular entity instances are created, persisted, and removed. The EntityManager interface defines the methods that are used to interact with the persistence context.

Application-Managed Entity Managers

Applications create EntityManager instances in this case by using the createEntityManager method of javax.persistence.EntityManagerFactory.

To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the javax.persistence.PersistenceUnit annotation:

```
EntityManagerFactory emf =  EntityManagerFactory emf =
context.getBean(EntityManagerFactory.class);
```

Then obtain an EntityManager from the EntityManagerFactory instance:

```
EntityManager em = emf.createEntityManager()
```

Container-Managed Entity Managers

To obtain an EntityManager instance, inject the entity manager into the application component:

```
@PersistenceContext
EntityManager em;
```

The class EntityManager

- EntityManager is the most important class of JPA
  - Full name javax.persistence.EntityManager
- Some methods of EntityManager
  - T find(primaryKey)
  - Query createQuery(String jpql)
    - Creates a JPQL query
  - Query createNativeQuery(String sql)
    - Creates a SQL query
- Some methods of Query
  - List getResultList()
    - Executes a Query and returns a list of objects

Java Persistence Query Language (JPQL)

- Very much like ordinary SQL
  - But not specific to any DBMS
- JPA converts JPQL to ordinary SQL for the actual DBMS