



Databases. Java DataBase Connectivity

JDBC

Where to Find The Code and Materials?

<https://github.com/iproduct/java-fundamentals-2022>

Базы от данни



Съдържание

1. Базис от данни (БД). Видове БД.
2. Системи за управление на бази от данни (СУБД)
3. Релационен и обектен модели
4. Релации и релационни схеми
5. Базови и производни релации. Домейни. Ограничения
6. Ключове.
7. Връзки между таблици и кардиналност.
8. Операции над релационни бази от данни
9. Нормализация на бази от данни
10. Транзакции и конкурентност
11. Новостите в JDBC™ 4.1 (Java 7): try-with-resources и RowSets

Бази от данни (БД)

- Дефиниция (Wikipedia):

База данни (БД, още база от данни) представлява колекция от логически свързани данни в конкретна предметна област, които са структурирани по определен начин. В първоначалния смисъл на понятието, използван в компютърната индустрия, базата от данни се състои от записи, подредени систематично, така че компютърна програма да може да извлича информация по зададени критерии.

Видове БД – според структурата

- Йерархични бази от данни
 - директорийна структура, файлова система
 - IBM IMS – 1968 г.
- Мрежови модел на БД - Чарлс Бейчман
 - позволява представяне на връзки 1:N между различните нива на йерархията
 - CODASYL IDMS – 1971 г.
- Релационни БД – Едгар Код – 1970 г.
- Обектно-ориентирани бази от данни

Видове БД – според предназначението

- Оперативни БД
- Аналитични БД
- Data warehouse
- БД за крайни потребители
- Външни БД
- Хипермедийни БД
- Навигационни БД
- Документно-ориентирани БД
- БД работещи в реално време

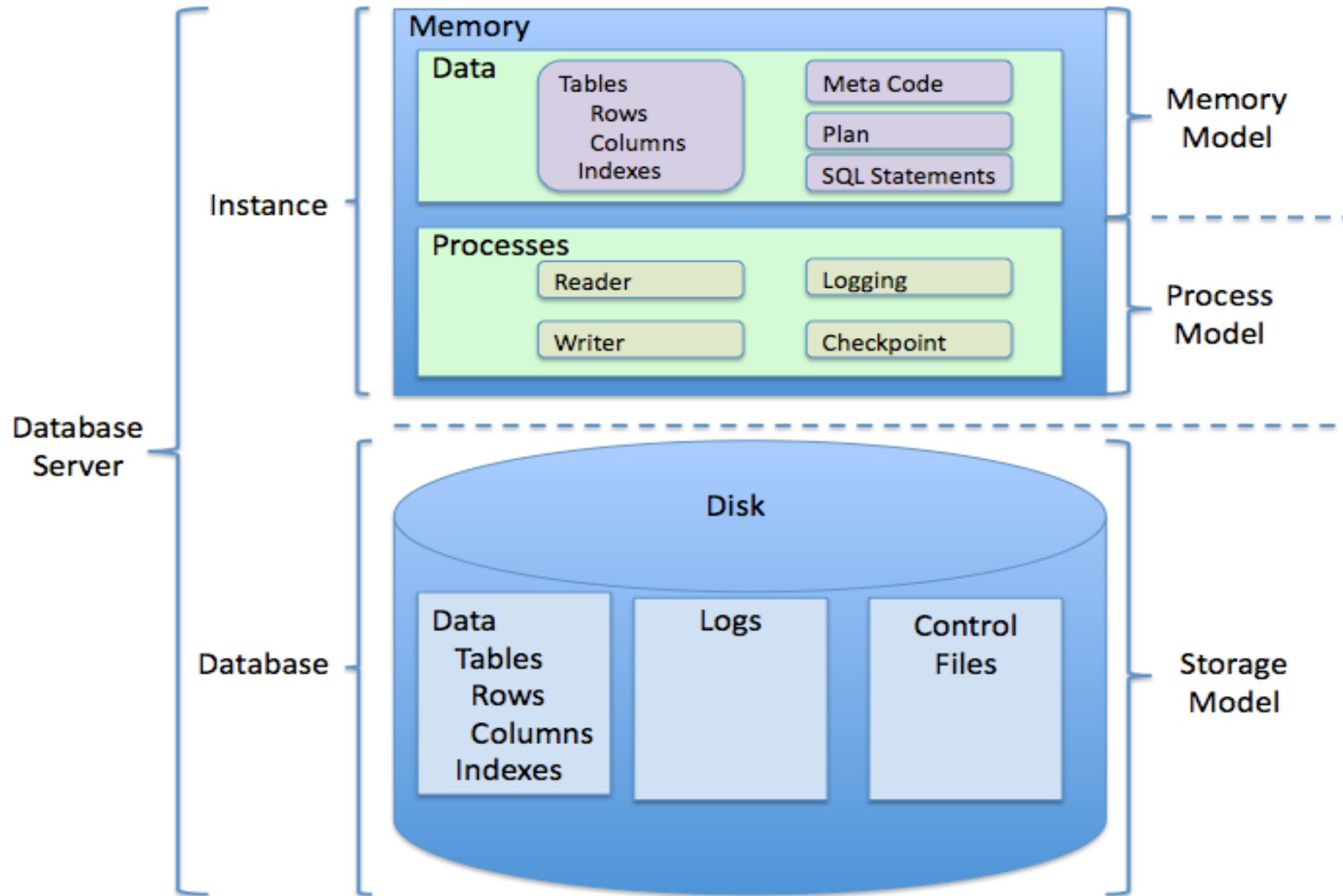
Системи за управление на бази от данни (СУБД)

- **Def:** съвкупност от компютърни програми, използвани за изграждане, поддръжка и използване на бази от данни
- **Примери:** MySQL, PostgreSQL, DB2, Microsoft SQL Server, Access, Oracle, Paradox, dBase, FoxPro, Cliper, Sybase, Informix
- **СУБД** създават, обработват и поддържат определени структури от данни. Най-популярен е релационният модел, при който данните се организират в таблици, между които се осъществяват връзки (т.н. релации). Таблиците се състоят от именувани редове и колони. Редовете се наричат записи, а колоните - полета.

ОСНОВНИ КОМПОНЕНТИ НА СУБД

- Релационни СУБД - RDBMS
 - Интерфейсни драйвери
 - SQL engine
 - Transaction engine
 - Relational engine
 - Storage engine
- Обектно-ориентирани БД – ODBMS
 - Езикови драйвери – C++, Java, .Net, Ruby
 - ОО език за заявки – JPAQL, LINQ, ...
 - Transaction & Storage engines

ОСНОВНИ КОМПОНЕНТИ НА СУБД

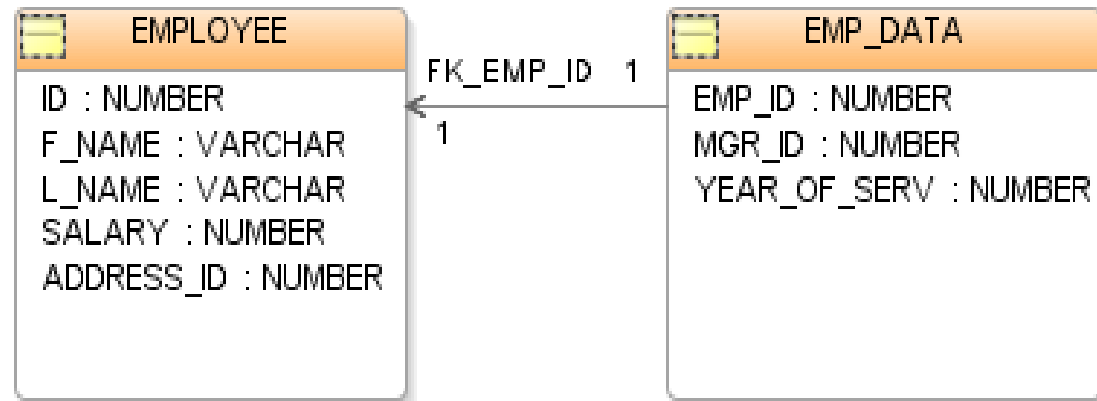
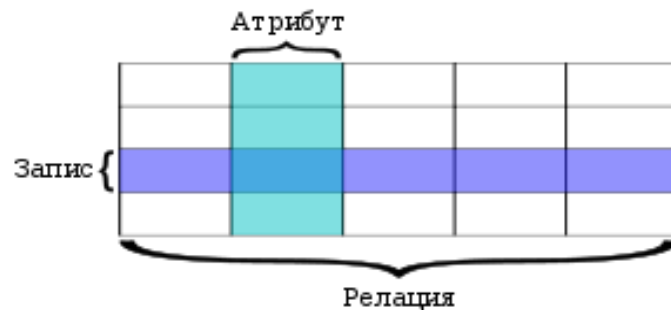


Релационна база от данни

- **Def:** Релационна база данни е тип база данни, която съхранява множество данни във вид на релации, съставени от записи и атрибути (полета) и възприемани от потребителите като таблици.
- Релационните бази данни понастоящем преобладават при избора на модел за съхранение на финансови, производствени, лични и други видове данни.
- Терминът „релационна база данни“ за първи път е предложен през 1970 година от Едгар Код, учен в IBM.

Релационен модел

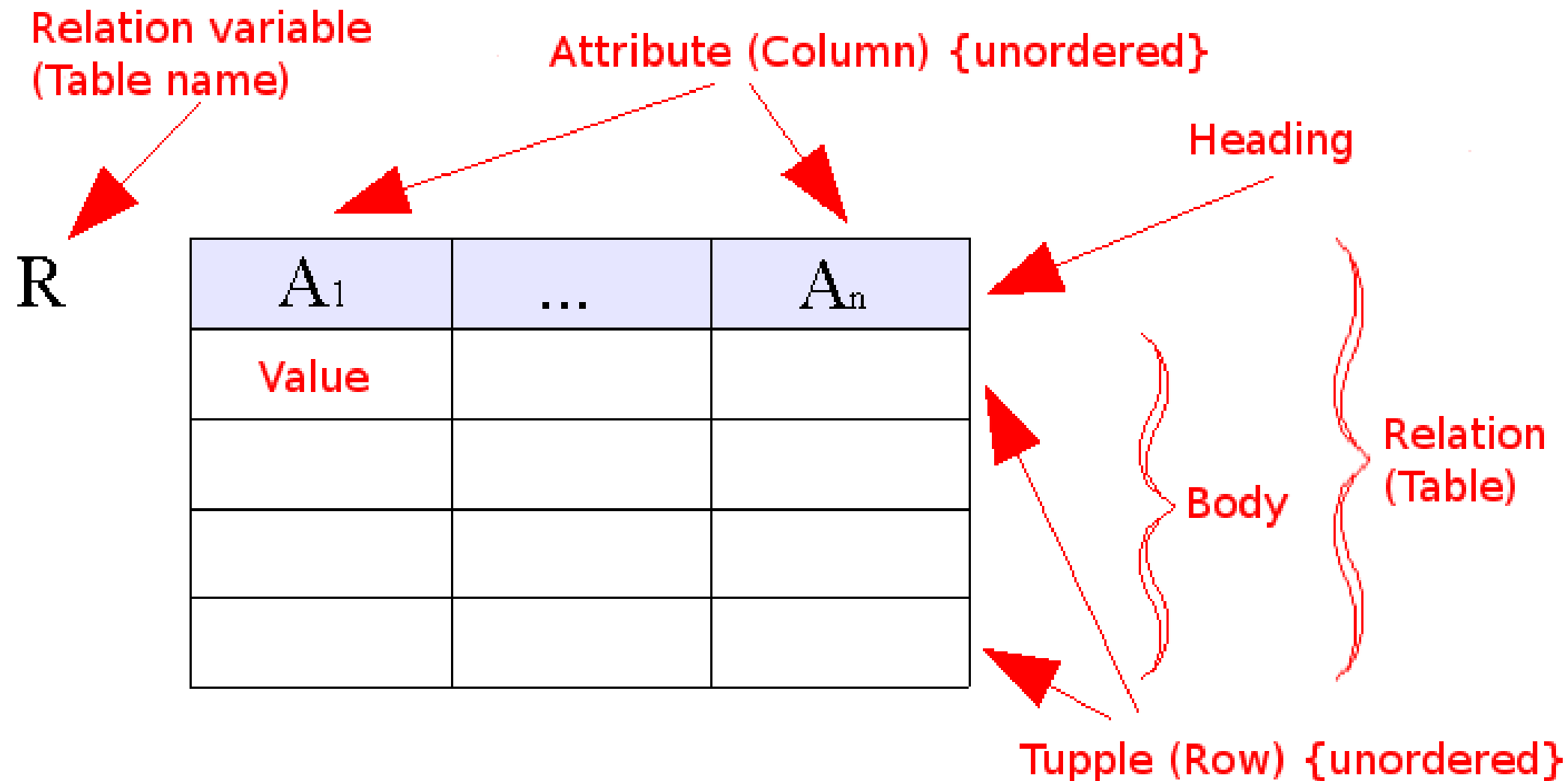
- релация, релационна схема (relation) \leftrightarrow таблица (table),
- запис, кортеж (tuple) \leftrightarrow ред (row)
- атрибут, поле (attribute) \leftrightarrow стълб, колона (column)



Релационен модел – таблица Customer

Customer ID	Tax ID	Name	Address	[More fields...]
1234567890	555-5512222	Ramesh	323 Southern Avenue	...
2223344556	555-5523232	Adam	1200 Main Street	...
3334445563	555-5533323	Shweta	871 Rani Jhansi Road	...
4232342432	555-5325523	Sarfaraz	123 Maulana Azad Sarani	...

Релационен модел



Релации и релационни схеми

- **Def:** Релацията (relation) се дефинира като множество от записи, които имат едни и същи атрибути. Записът обикновено представя обект и информация за обекта, който обичайно е физически обект или понятие. Релацията обикновено се оформя като таблица, организирана по редове и колони. Всички данни, които се съдържат в даден атрибут, принадлежат на едно и също множество от допустими стойности, наречено домейн, и съблюдават едни и същи ограничения.
- Заглавието (heading) на таблицата се нарича релационна схема, а множеството от всички релационни схеми в БД – схема на БД (database schema)

Базови и производни релации. Домейни. Ограничения

- **Def:** . Релациите, които съхраняват данните, се наричат **базови релации (base relations)** или таблици (**tables**). Други релации обаче не съхраняват данни, а се изчисляват чрез прилагането на операции над други релации. Наричат се **производни релации (отношения)**, а в приложенията за бази данни се наричат заявка (**query**) и изглед (**view**).
- **Def: Домейн** в базите данни означава множеството от допустимите стойности на даден атрибут на релация, т.е. представлява известно ограничение върху стойностите на атрибута.
- **Def: Ограничения (constraints)** позволяват в още по-голяма степен да се специфицират стойностите, които атрибутите от даден домейн могат да приемат – например от 1 до 10.

Ключове

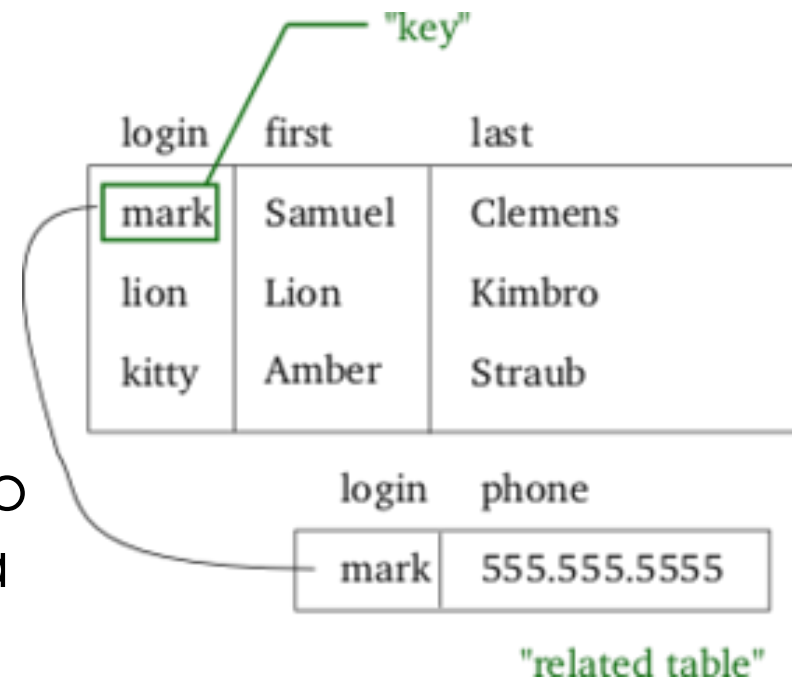
- **Ключ (key)** се наричат един или повече атрибута, такива че:

1) релацията няма две различни записа с едни и същи стойности за тези атрибути

2) няма строго подмножество на тези атрибути с горното свойство

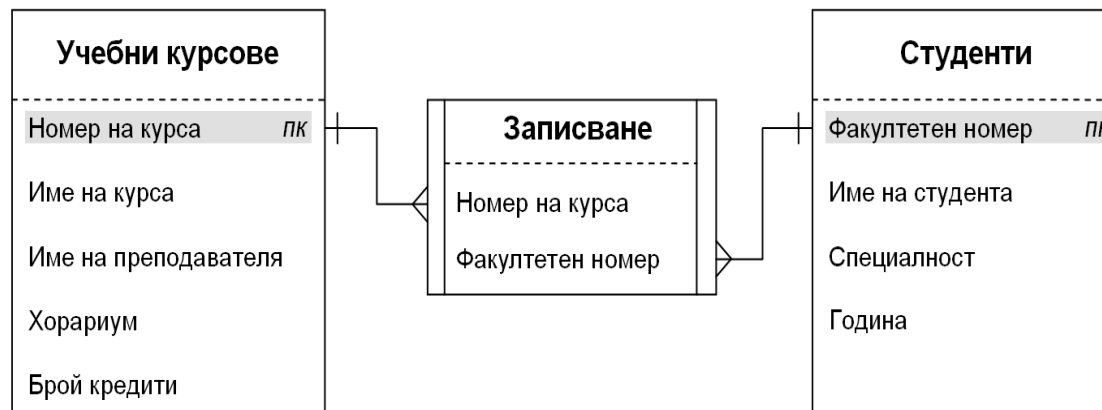
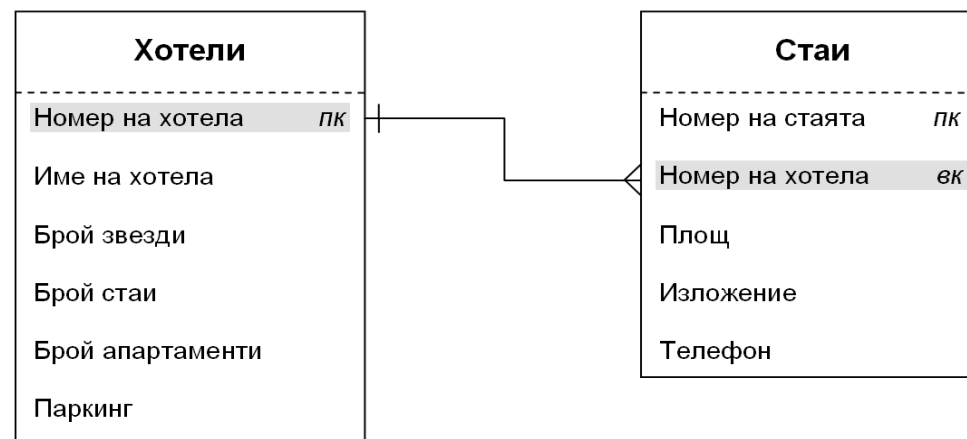
- **Първичен ключ (primary key)** е атрибут (по-рядко група атрибути), който служи да идентифицира по уникален начин всеки запис (екземпляр) на релацията

- **Външният ключ (foreign key)** е необходим, когато налице е отношение между две таблици (релации).

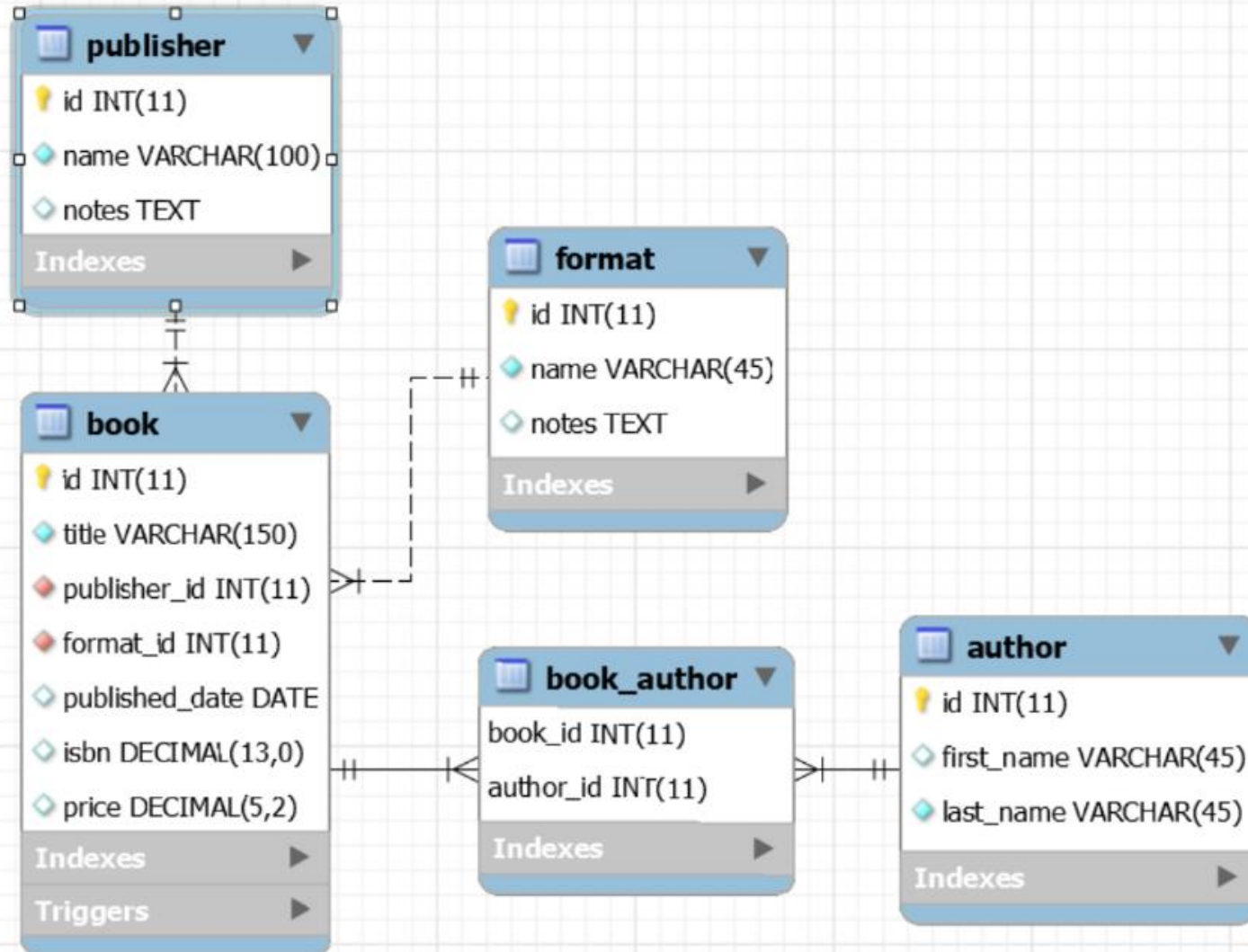


Връзки между таблици и кардиналност

- **Отношение (relationship)** се нарича зависимост, съществуваща между две таблици, когато записи от първата таблица могат да се свържат по някакъв начин със записи от втората таблица.
- Кардиналност:
 - едно към едно (1:1),
 - едно към много (1:N),
 - много към много (M:N).



ER Diagram



Операции над релационни бази от данни

- **Оператор обединение (union)** комбинира записи от две релации и премахва от резултата всички евентуални повтарящи се записи. Релационният оператор обединение е еквивалентен на SQL-оператора UNION.
- **Оператор сечение (intersection)** извежда множеството от записи, които са общи за двете релации. Сечението в SQL е реализирано чрез оператора INTERSECT.
- **Оператор разлика (difference)** се прилага над две релации и в резултат връща множеството от записите от първата релация, които не съществуват във втората релация. В SQL разликата е имплементирана посредством оператора EXCEPT или MINUS.

Операции над релационни бази от данни

- **Оператор декартово произведение** или само произведение (Cartesian product, cross join, cross product) на две релации представлява съединение (join), при което всеки запис от първата релация се конкатенира с всеки запис от втората релация. В SQL операторът е реализиран под името CROSS JOIN.
- **Операцията селекция (selection, restriction)** връща само онези записи от дадена релация, които отговарят на избрани критерии, т.е. подмножество в термините на теорията на множествата. Еквивалентът на селекцията в SQL е заявка SELECT с клауза WHERE.

Операции над релационни бази от данни

- **Операцията проекция (projection)** е по своя смисъл селекция, при която повтарящите се записи се отстраняват от резултата. В SQL е реализирана с клаузата GROUP BY или чрез ключовата дума DISTINCT, внедрена в някои диалекти на SQL.
- **Операцията съединение (join)**, дефинирана за релационни бази данни, често се нарича и естествено съединение (natural join). При този вид съединение две релации са свързани посредством общите им атрибути. В SQL тази операция е реализирана приблизително чрез оператора за съединение INNER JOIN. Други видове съединение са лявото и дясното външни съединения, внедрени в SQL като LEFT JOIN и RIGHT JOIN, съответно.

Операции над релационни бази от данни

- **Операцията деление (division)** е малко по-сложна операция, при която записи от една релация в ролята на делител се използват, за да се раздели втора релация в ролята на делимо. По смисъла си, тази операция е обратна на операцията (декартово) произведение.

Нормализация на бази от данни

- **Нормализацията**, т.е. привеждането в нормална форма включва набор от практики по отстраняването на повторения сред данните, което от една страна води до икономия на памет и повишено бързодействие, а от друга страна предпазва от аномалии при манипулирането с данните (вмъкване, актуализиране и изтриване) и от загуба на тяхната цялост. В процеса на нормализация се осигурява оптимална структура на базата от данни, основаваща се на взаимозависимостта между данните. Структурата на таблиците се трансформира, с цел да се оптимизират функционалните зависимости на съставните им атрибути.
- **Нормални форми** – https://en.wikipedia.org/wiki/Database_normalization

Транзакции и конкурентност

- Транзакция = бизнес събитие
- ACID правила:
 - **Атомарност (Atomicity)**: или се изпълнява цялата транзакция – всички задачи, или не се изпълнява никоя от тях (rolled back).
 - **Съгласуваност (Consistency)**: транзакцията трябва да запазва integrity constraints.
 - **Изоляция (Isolation)**: две едновременно транзакции не могат да си взаимодействат.
 - **Постоянство (Durability)**: успешно завършените транзакции не могат да се отменят.

Transaction Isolation Levels

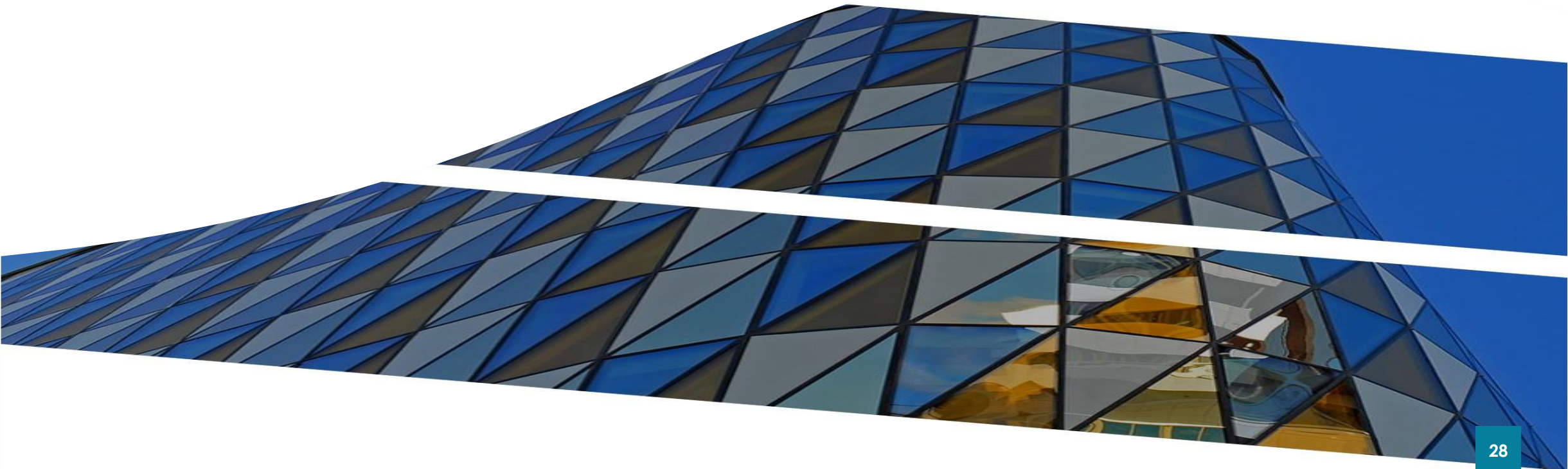
- **DEFAULT** - use the default isolation level of the underlying datastore
- **READ_UNCOMMITTED** – dirty reads, non-repeatable reads and phantom reads can occur
- **READ_COMMITTED** – prevents dirty reads; non-repeatable reads and phantom reads can occur
- **REPEATABLE_READ** – prevents dirty reads and non-repeatable reads; phantom reads can occur
- **SERIALIZABLE** – prevents dirty reads, non-repeatable reads and phantom reads

Types of Transactions

- **Global transactions** – enable you to work with multiple transactional resources, typically relational databases and message queues (JTA UserTransaction, JNDI lookup).
- **Local transactions** – resource-specific, such as a transaction associated with a JDBC connection, but cannot work across multiple transactional resources.
- **Spring Framework's transactions** – consistent programming model in any environment, write code once, and it can use different transaction management strategies in different environments – both **declarative and programmatic transaction management** (Spring Framework transaction abstraction) - **@Transactional**.

Introduction to SQL

Practical Exercises – <https://www.w3schools.com/sql/>



<https://www.mysqltutorial.org/>



MySQL 101 /dev/random

Practical Exercises:

<https://blogs.sakienvirotech.com/index.php/random/2011/09/05/mysql-101-creating-your-first>



Java DataBase Connectivity (JDBC)

Practical Exercises



Java Database Connectivity (JDBC)

- **Java Database Connectivity (JDBC)** е **application programming interface (API)** на езика **Java**, който дефинира как клиентите могат да достъпват, извличат и модифицират данни в една релационна база от данни.
- **JDBC-to-ODBC bridge** позволява връзки към всякакви ODBC-достъпни източници на данни в **Java virtual machine (JVM)** среда.

Java Database Connectivity (JDBC) – Example (1)

```
Scanner sc = new Scanner(System.in);
Properties props = new Properties();

System.out.println("Enter username (default root): ");
String user = sc.nextLine().trim();
user = user.length() > 0 ? user : "root";
props.setProperty("user", user);

String password = sc.nextLine().trim();
password = password.length() > 0 ? password : "root";
props.setProperty("password", password);
```

Java Database Connectivity (JDBC) – Example (2)

// 1. Load jdbc driver (optional)

```
try {  
    Class.forName("com.mysql.cj.jdbc.Driver");  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
    System.exit(0);  
}  
System.out.println("Driver loaded successfully.");
```

// 2. Connect to DB

```
Connection connection =  
    DriverManager.getConnection(  
        "jdbc:mysql://localhost:3306/employees?useSSL=false", props);  
  
System.out.println("Connected successfully.");
```

Java Database Connectivity (JDBC) – Example (3)

// 3. Execute query

```
PreparedStatement stmt =  
    connection.prepareStatement("SELECT * FROM employees JOIN salaries ON  
employees.emp_no=salaries.emp_no WHERE salaries.salary > ?");
```

```
System.out.println("Enter minimal salary (default 20000): ");  
String salaryStr = sc.nextLine().trim();
```

```
double salary = Double.parseDouble(salaryStr);
```

```
stmt.setDouble(1, salary);  
ResultSet rs = stmt.executeQuery();
```

Java Database Connectivity (JDBC) – Example (4)

// 4. Process results

```
while (rs.next()) {  
    System.out.printf("| %-15.15s | %-15.15s | %10.2f |\n",  
        rs.getString(2),  
        rs.getString("last_name"),  
        rs.getDouble("salary")  
    );  
}
```

// 5. Close connection and statement
connection.close();

Novelties in JDBC™ 4.1 (Java 7): try-with-resources

- `java.sql.Connection`, `java.sql.Statement` и `java.sql.ResultSet` имплементируют интерфейса **AutoCloseable**:

```
Class.forName("com.mysql.jdbc.Driver");           //Load MySQL DB driver

try (Connection c = DriverManager.getConnection(dbUrl, user, password);
     Statement s = c.createStatement() ) {

    c.setAutoCommit(false);

    int records = s.executeUpdate("INSERT INTO product " //Insert new product
                                + "VALUES ('CP-00002', 'Lenovo', 790.0, 'br', 'Laptop')");

    System.out.println("Successfully inserted "+ records + " records.");

    records = s.executeUpdate("UPDATE product "         //Update product price
                                + "SET price=470, description='Classic laptop' "
                                + "WHERE code='CP-00001'");

    System.out.println("Successfully updated "+ records + " records.");

    c.commit();                                       //Finish transaction
}
```

Novelties in JDBC™ 4.1 (Java 7): RowSets (1)

<https://docs.oracle.com/javase/tutorial/jdbc/basics/rowset.html>

- **RowSet** – дава възможност да работим с данните от таблиците в базата от данни като с нормални JavaBeans™ компоненти – да достъпваме и променяме стойностите като свойства (properties), да закачаме слушатели на събития свързани с промяна на данните (event listeners), както и да скролираме (scroll) и променяме (update) данните в заредените в RowSet-а редове
- Свързан (connected) RowSet
 - **JdbcRowSet** – обвиващ клас около стандартния JDBC ResultSet
- Несвързан (disconnected) RowSet
 - **CachedRowSet** - кешира данните в паметта, подходящ за изпращане
 - **WebRowSet** - подходящ за изпращане на данните през HTTP (XML)
 - **JoinRowSet** - позволява извършване на JOIN без свързване към БД
 - **FilteredRowSet** - позволява локално филтриране на данните (R/W)

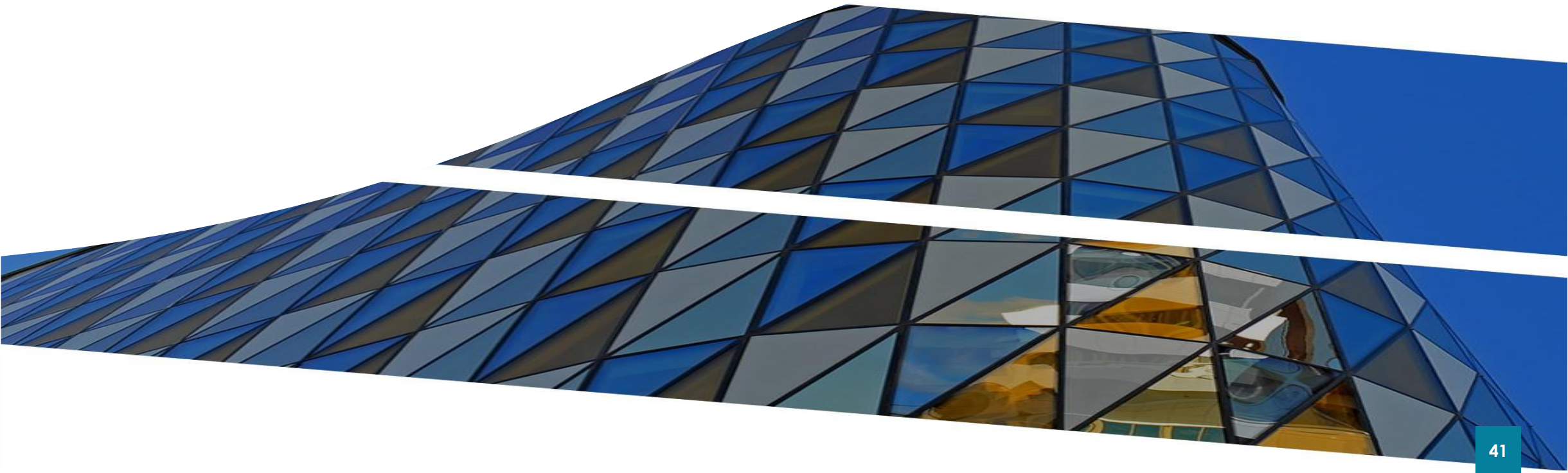
Novelties in JDBC™ 4.1 (Java 7): RowSets (2)

```
try {  
    RowSetFactory rsFactory = RowSetProvider.newFactory();  
    JdbcRowSet rowSet = rsFactory.createJdbcRowSet();  
    rowSet.setUrl("jdbc:mysql://localhost:3306/java21");  
    rowSet.setUsername(username);  
    rowSet.setPassword(password);  
    rowSet.setCommand("SELECT * FROM product");  
    rowSet.execute();  
    rowSet.absolute(3);           // Позиционира на третия запис  
    rowSet.updateFloat("price", 18.70f); //Променя цената на 18.70  
    rowSet.updateRow();           // Активира промените в RowSet-а  
}
```

Novelties in JDBC™ 4.1 (Java 7): RowSets (3)

```
try {  
    RowSetFactory rsFactory = RowSetProvider.newFactory();  
    CachedRowSet rowSet = rsFactory.createCachedRowSet();  
    rowSet.setUrl("jdbc:mysql://localhost:3306/java21");  
    rowSet.setUsername(username);  
    rowSet.setPassword(password);  
    rowSet.setCommand("SELECT * FROM product");  
    int [] keyColumns = {1}; rowSet.setKeyColumns(keyColumns);  
    rowSet.execute();  
    rowSet.absolute(3); rowSet.updateFloat("price", 18.70f);  
    rowSet.updateRow();  
    rowSet.acceptChanges(con); // Синхронизация с БД  
}
```


Spring JDBC



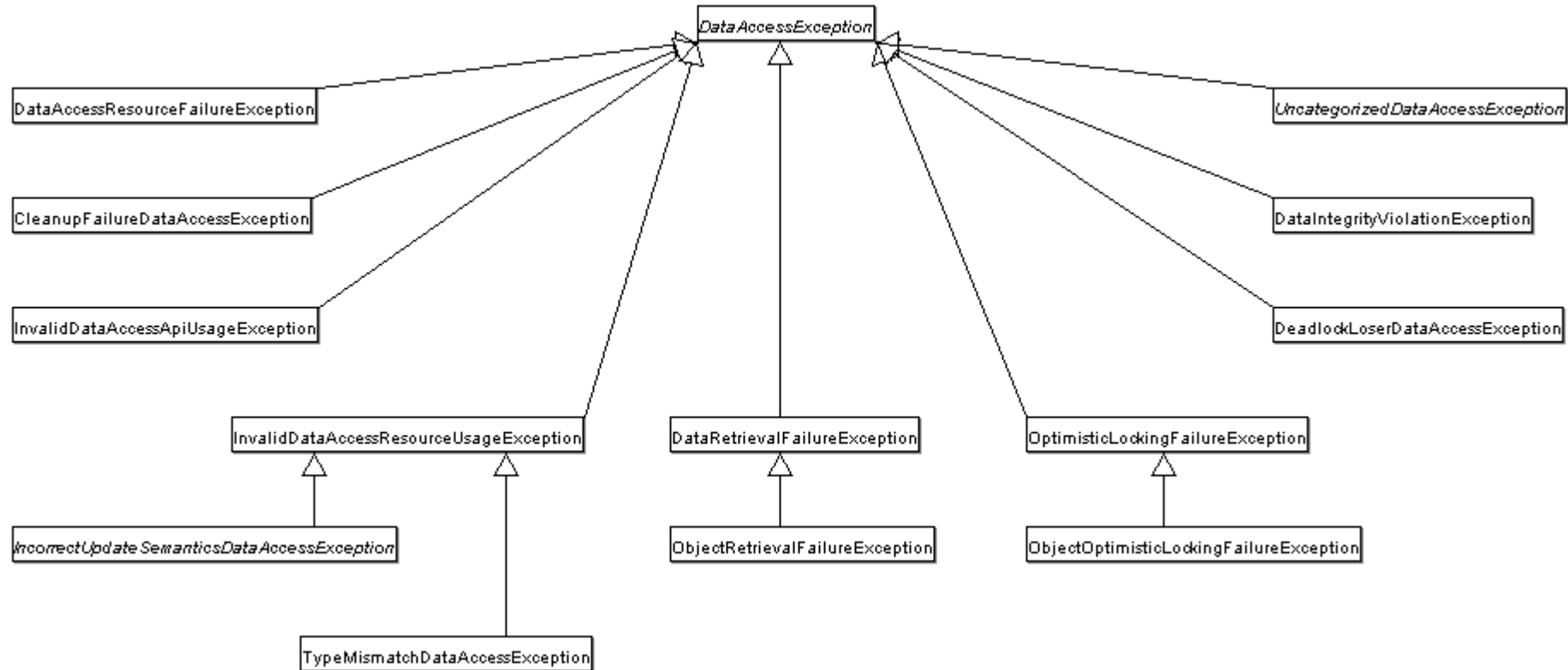
Spring Data Access Objects (DAO)

- ❖ **Data Access Object (DAO)** – simplifies work with different data access technologies like **JDBC**, **Hibernate** or **JPA** in a consistent way.
- ❖ Consistent exception hierarchy - **RuntimeExceptions**
- ❖ Annotations used for configuring **DAO** or **Repository** classes – with automatic exception translation:

- ```
import org.springframework.stereotype.Repository;

@Repository
public class SomeMovieFinder implements MovieFinder {
 // ...
}
```

# DAO Exception Hierarchy



# DAO Repository - JDBC

- `import javax.sql.DataSource;`

```
@Repository
public class JdbcMovieFinder implements MovieFinder {

 private JdbcTemplate jdbcTemplate;

 @Autowired
 public void init(DataSource dataSource) {
 this.jdbcTemplate = new JdbcTemplate(dataSource);
 }

 // ...

}
```

# DAO Repository - Hibernate

```
• import org.hibernate.SessionFactory;
 import
•
 org.springframework.beans.factory.annotation.Autowired;
 import org.springframework.stereotype.Repository;

 @Repository
 public class HibernateMovieFinder implements MovieFinder
 {

 private SessionFactory sessionFactory;

 @Autowired
 public void setSessionFactory(SessionFactory
 sessionFactory) {
 this.sessionFactory = sessionFactory;
 }

 // ...
 }
```

# DAO Repository - JPA

- ```
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Repository
public class JpaMovieFinder implements MovieFinder {

    @PersistenceContext
    private EntityManager entityManager;

    // ...

}
```

Spring JDBC

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

JDBC DB Access Alternatives

- ❖ **JdbcTemplate** - the “classic” Spring JDBC approach and the most popular - "lowest level", all others use a JdbcTemplate
- ❖ **NamedParameterJdbcTemplate** – wraps a JdbcTemplate to provide named parameters instead of the "?" placeholders
- ❖ **SimpleJdbcInsert** and **SimpleJdbcCall** uses DB metadata, you only need to provide the name of the table or procedure and provide a map of parameters matching column names.
- ❖ **RDBMS Objects** – include **MappingSqlQuery**, **SqlUpdate** and **StoredProcedure**, you create reusable and thread-safe objects during initialization, like JDO Query, wherein you define your query string, declare parameters, and compile the query. Then you can execute methods multiple times.

JDBC Repository Methods - I

- `@Override`

```
public Collection<Article> findAll() {  
    List<Article> articles = this.jdbcTemplate  
        .query("select * from articles", new  
ArticleMapper());  
    log.info("Articles loaded: {}", articles.size());  
    return articles;  
}
```

- `@Override`

```
public Article find(long id) {  
    Article article = this.jdbcTemplate.queryForObject(  
        "select * from articles where id = ?",  
        new Object[]{id}, new ArticleMapper());  
    log.info("Article found: {}", article);  
    return article;  
}
```

JDBC Repository Methods - II

- `@Override`

```
public Article create(Article article) {
    KeyHolder keyHolder = new GeneratedKeyHolder();
    jdbcTemplate.update(new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement
            (Connection connection) throws SQLException {
            PreparedStatement ps = connection
            .prepareStatement(INSERT_SQL, new String[] {"id"});
            ps.setString(1, article.getTitle());
            ps.setString(2, article.getContent());
            ps.setTimestamp(3, new Timestamp(
                article.getCreatedDate().getTime()));
            ps.setString(4, article.getPictureUrl());
            return ps;
        }
    }, keyHolder);
    article.setId(keyHolder.getKey().longValue());
    log.info("Article created: {}", article);
    return article;
}
```

JDBC Repository Methods - III

- `@Override`

```
public Article update(Article article) {  
    int count = this.jdbcTemplate.update(  
        "update articles set (title, content, created_date,  
        picture_url)  
        VALUES (?, ?, ?, ?) where id = ?",  
        article.getTitle(), article.getContent(),  
        article.getCreatedDate(),  
        article.getPictureUrl(), article.getId());  
    log.info("Article updated: {}", article);  
    return article;  
}
```
- `@Override`

```
public boolean remove(long articleId) {  
    int count = this.jdbcTemplate.update(  
        "delete from articles where id = ?",  
        Long.valueOf(articleId));  
    return count > 0;  
}
```

JDBC DataSource - I

- `@Configuration`
`@ComponentScan({"org.iproduct.spring.webmvc.service",`
`"org.iproduct.spring.webmvc.dao"})`
`@PropertySource("classpath:jdbc.properties")`
`public class SpringRootConfig {`

`@Value("${jdbc.driverClassName:org.postgresql.Driver}")`
`private String driverClassname;`

`@Value("${jdbc.url:jdbc:postgresql://localhost/articles}")`
`private String url;`

`@Value("${jdbc.username:postgres}")`
`private String username;`

`@Value("${jdbc.password:postgres}")`
`private String password;`
`(- continues on next slide -)`

JDBC DataSource - II

- **@Bean**
DataSource getDataSource() {
 DriverManagerDataSource dataSource =
 new DriverManagerDataSource();
 //PostgreSQL database we are using
 dataSource.setDriverClassName(driverClassname);
 dataSource.setUrl(url); *//change url*
 dataSource.setUsername(username); *//change username*
 dataSource.setPassword(password); *//change pwd*

 //H2 database
 /*
 dataSource.setDriverClassName("org.h2.Driver");
 dataSource.setUrl("jdbc:h2:tcp://localhost/~/test");
 dataSource.setUsername("sa");
 dataSource.setPassword(""); */
 return dataSource;
 }
}

Transactions



Transactions and Concurrency

- ❖ **Transaction** = Commits as **Business Event**
- ❖ **ACID rules:**
 - ❖ **Atomicity** – the whole transaction is completed (commit) or no part is completed at all (rollback).
 - ❖ **Consistency** – transaction should preserve existing integrity constraints
 - ❖ **Isolation** – two uncompleted transactions can not interact
 - ❖ **Durability** – successfully completed transactions can not be rolled back

Advantages of Spring Transactions

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

Spring Transaction Management

- ❖ **Global transactions** – enable you to work with multiple transactional resources, typically relational databases and message queues (JTA UserTransaction, JNDI lookup).
- ❖ **Local transactions** – resource-specific, such as a transaction associated with a JDBC connection, but cannot work across multiple transactional resources.
- ❖ **Spring Framework's transactions** – consistent programming model in any environment, write code once, and it can use different transaction management strategies in different environments – both **declarative and programmatic transaction management** (Spring Framework transaction abstraction).

Spring Transaction Abstraction

```
public interface PlatformTransactionManager {  
  
    TransactionStatus getTransaction(TransactionDefinition definition)  
        throws TransactionException;  
  
    void commit(TransactionStatus status) throws TransactionException;  
  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```

- **TransactionDefinition:**
 - **Propagation** – what to do when a transactional method is executed when a transaction context already exists)
 - **Isolation** – degree to which this transaction is isolated from the work of other transactions (e.g. can this transaction see uncommitted writes from other transactions?)
 - **Timeout** – how long run before timing out and being rolled back
 - **Read-only status:** used when you read but not modify data

Transaction Isolation Levels

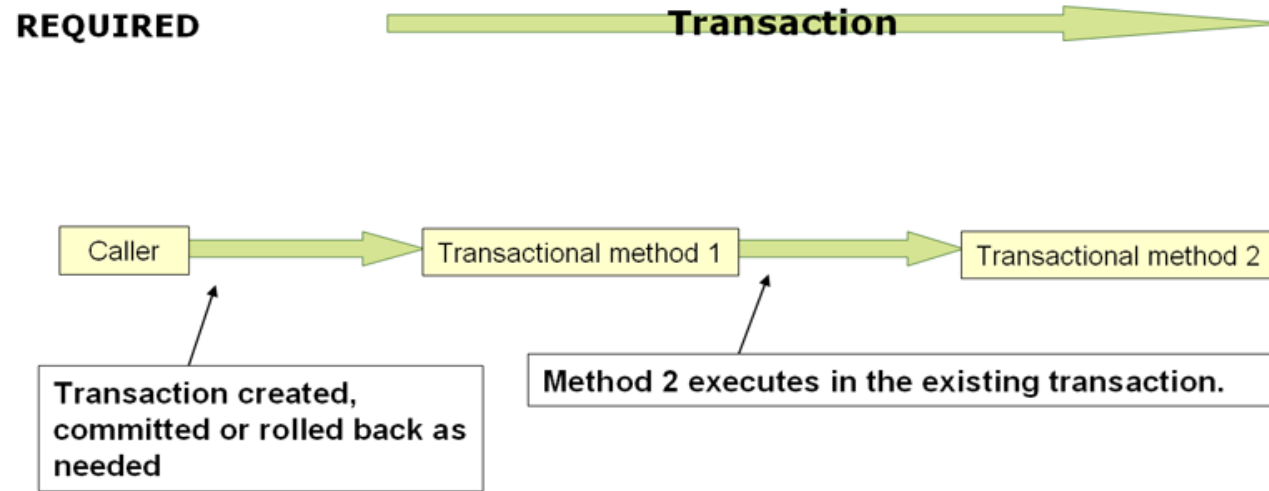
- ❖ **DEFAULT** - use the default isolation level of the underlying datastore
- ❖ **READ_UNCOMMITTED** – dirty reads, non-repeatable reads and phantom reads can occur
- ❖ **READ_COMMITTED** – prevents dirty reads; non-repeatable reads and phantom reads can occur
- ❖ **REPEATABLE_READ** – prevents dirty reads and non-repeatable reads; phantom reads can occur
- ❖ **SERIALIZABLE** – prevents dirty reads, non-repeatable reads and phantom reads

Transactions Propagation

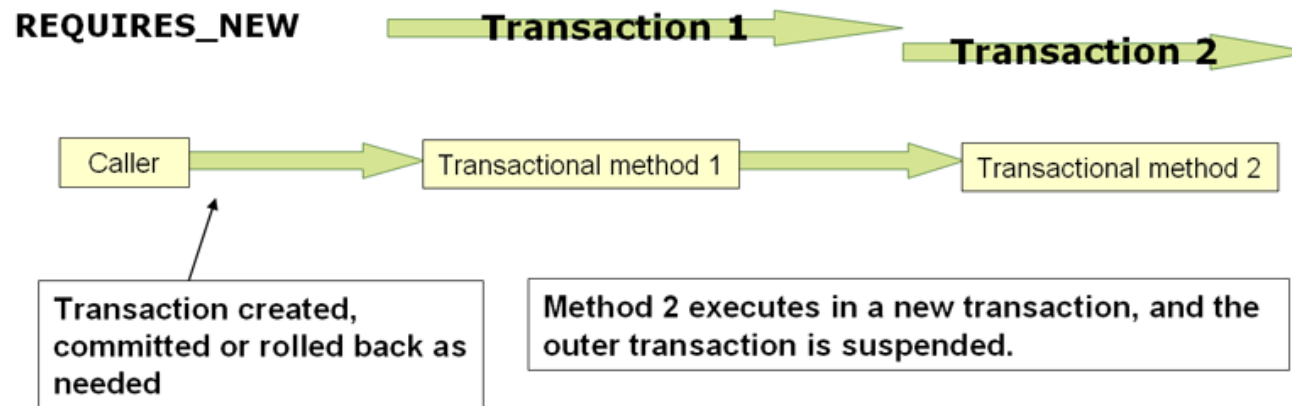
- ❖ **SUPPORTS** – supports transaction if existing, executes non-transactionally if not
- ❖ **REQUIRED** – supports transaction if existing, creates new if not
- ❖ **REQUIRES_NEW** – always create a new transaction, and suspend the current transaction if one exists
- ❖ **MANDATORY** – supports the current transaction, throws an exception if none exists
- ❖ **NEVER** – execute non-transactionally, throw an exception if a transaction exists
- ❖ **NOT_SUPPORTED** - execute non-transactionally, suspend the current transaction if one exists
- ❖ **NESTED** – executes within a nested transaction if current transaction exists, else does like **PROPAGATION_REQUIRED**

Transactions Propagation

REQUIRED



REQUIRES_NEW



TransactionStatus

```
public interface TransactionStatus extends SavepointManager {  
  
    boolean isNewTransaction();  
  
    boolean hasSavepoint();  
  
    void setRollbackOnly();  
  
    boolean isRollbackOnly();  
  
    void flush();  
  
    boolean isCompleted();  
  
}
```

Transactions and Concurrency

- **DataSourceTransactionManager** – JDBC local transactions, allows thread bound connections, obtained

```
<bean id="txManager"  
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

- **JtaTransactionManager** – using global JTA transactions

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/articles"/>  
<bean id="txManager"  
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

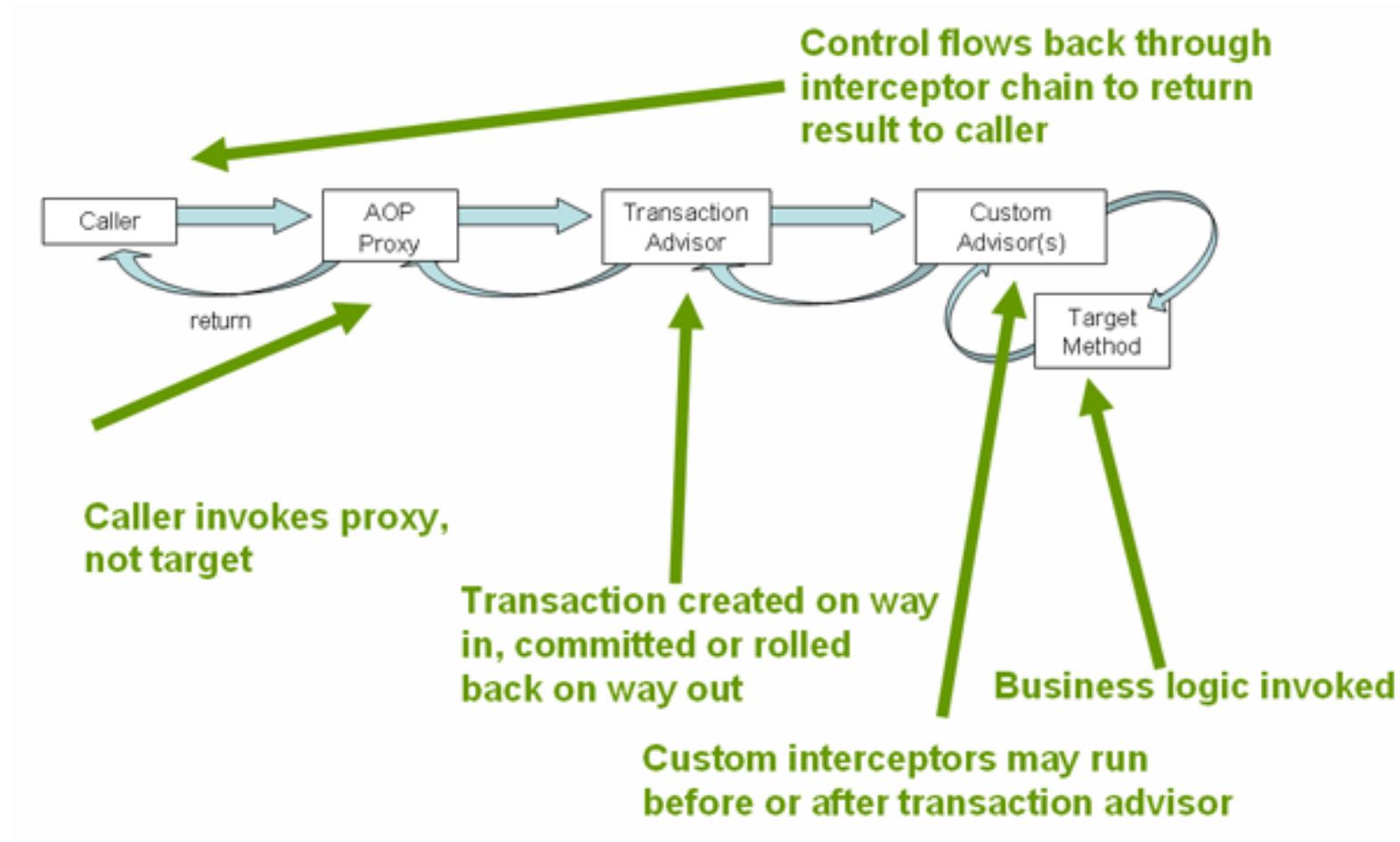
- **@Transactional** – declarative transactions
- **TransactionTemplate** or directly using **PlatformTransactionManager** – programmatic transactions

Declarative Transaction Demarcation

- Enabling declarative transactions:
 - **@EnableTransactionManagement**
 - **<tx:annotation-driven/>**
- **@Transactional** attributes: **value** (optional qualifier specifying the transaction manager to be used), **propagation**, **isolation**, **readOnly**, **timeout** (in seconds), **rollbackFor** (optional array of exception classes that must cause rollback), **rollbackForClassName**, **noRollbackFor** (optional array of exception classes that must not cause rollback), **noRollbackForClassName**

```
@Transactional(propagation = Propagation.REQUIRED)
public List<Article> createArticlesBatch(List<Article>
articles) {
    List<Article> created = articles.stream()
        .map(article -> addArticle(article))
        .collect(Collectors.toList());
    return created;
}
```


Transactions via AOP Proxies



Customizing Transactions using AOP

```
<aop:config>
  <aop:pointcut id="entryPointMethod"
    expression="execution(* x.y..*Service.*(..))"/>

  <aop:advisor advice-ref="txAdvice" pointcut-ref="entryPointMethod"
order="2"/>

  <aop:aspect id="profilingAspect" ref="profiler">
    <aop:pointcut id="methodWithReturn"
      expression="execution(!void x.y..*Service.*(..))"/>
    <aop:around method="profile" pointcut-ref="methodWithReturn"/>
  </aop:aspect>
</aop:config>

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
```

Programmatic Transactions - I

```
public List<Article> createArticlesBatch(List<Article> articles)
{
    return transactionTemplate.execute(
        new TransactionCallback<List<Article>>() {
            public List<Article> doInTransaction(
                TransactionStatus status)
            {
                List<Article> created = articles.stream()
                    .map(article -> {
                        try {
                            return addArticle(article);
                        } catch (ConstraintViolationException ex) {
                            log.error("Error:{}", ex.getMessage());
                            status.setRollbackOnly();
                            return null;
                        }
                    })
                    .collect(Collectors.toList());
                return created;
            }
        });
}
```

Programmatic Transactions - II

```
public List<Article> createArticlesBatch(List<Article> articles) {
    DefaultTransactionDefinition def = new DefaultTransactionDefinition();
    def.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);
    def.setTimeout(5);
    TransactionStatus status = transactionManager.getTransaction(def);
    List<Article> created = articles.stream()
        .map(article -> {
            try {
                Article resultArticle = addArticle(article);
                applicationEventPublisher.publishEvent(
                    new ArticleCreationEvent(resultArticle));
                return resultArticle;
            } catch (ConstraintViolationException ex) {
                log.error("Error: {}", ex.getMessage());
                transactionManager.rollback(status); // ROLLBACK
                throw ex;
            }
        })
        .collect(Collectors.toList());
    transactionManager.commit(status); // COMMIT
    return created;
}
```

@TransactionalEventListener

```
@TransactionalEventListener
public void
handleArticleCreatedTransactionCommit(ArticleCreationEvent
creationEvent) {
    log.info(">>> Transaction COMMIT for article: {}",
            creationEvent.getArticle());
}
```

```
@TransactionalEventListener(phase = TransactionPhase.AFTER_ROLLBACK)
public void
handleArticleCreatedTransactionRollback(ArticleCreationEvent
creationEvent) {
    log.info(">>> Transaction ROLLBACK for article: {}",
            creationEvent.getArticle());
}
```

Resources

- Wikipedia – Free Online Encyclopedia –
http://wikipedia.org/https://en.wikipedia.org/wiki/Relational_model
- Oracle® Java™ Technologies webpage –
<http://www.oracle.com/technetwork/java/>
- Oracle®: The Java Tutorials: Lesson: JDBC Basics –
<http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>
- Oracle®: Новости в JDBC™ 4.1 –
http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html
- Joshua Bloch: Automatic Resource Management (V.2) –
https://docs.google.com/View?id=ddv8ts74_3fs7483dp

Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>