Laboratório de Banco de Dados

SQL Avançado

Acesso via Linguagens de Programação de Propósito Geral

UFMT

Junho de 2025

Agenda

1. **SQL via Linguagem de Programação de Propósito Geral.**
2. Java e a API JDBC.
   - Configuração JDBC.
   - Estabelecimento de conexão via JDBC.
   - Executando comandos SQL via JDBC.
   - Programas exemplo.
   - Execução de consultas.
   - Conjuntos de resultados roláveis e atualizáveis.
   - Row sets.

SQL via Ling. de Programação de Propósito Geral

- Um programador de aplicações deve ter acesso a linguagens de programação de propósito geral com capacidade de processar SQL por pelo menos dois motivos:

1. SQL não fornece o mesmo poder de expressividade de uma linguagem de propósito geral. Por exemplo, existem operações com dados que podem ser expressas em linguagens como C, Java e Python, mas que não podem ser expressas em SQL.

2. Ações não declarativas como imprimir um relatório, interagir com um usuário, ou o envio de resultados de consultas para uma interface gráfica não podem ser feitas com SQL.

SQL via Ling. de Programação de Propósito Geral

- Há duas abordagens principais para processar SQL via linguagens de programação de propósito geral:
  1. **SQL dinâmico**. Consiste em um conjunto de funções (linguagens procedurais) ou métodos (linguagens orientadas a objetos) usados para submeter requisições SQL ao SGBD. As requisições SQL são formuladas como strings de caracteres construídas em tempo de execução. O resultado retornado pelo SGBD é armazenado em estruturas de dados da linguagem/API para processamento posterior.
  2. **SQL embutido**. Consiste em requisições SQL identificadas em tempo de compilação por um pré-processador que traduz os comandos em chamadas de função. Em tempo de execução, essas funções são usadas para conectar com o SGB e realizar as operações desejadas com o banco de dados.

Incompatibilidades de Linguagens

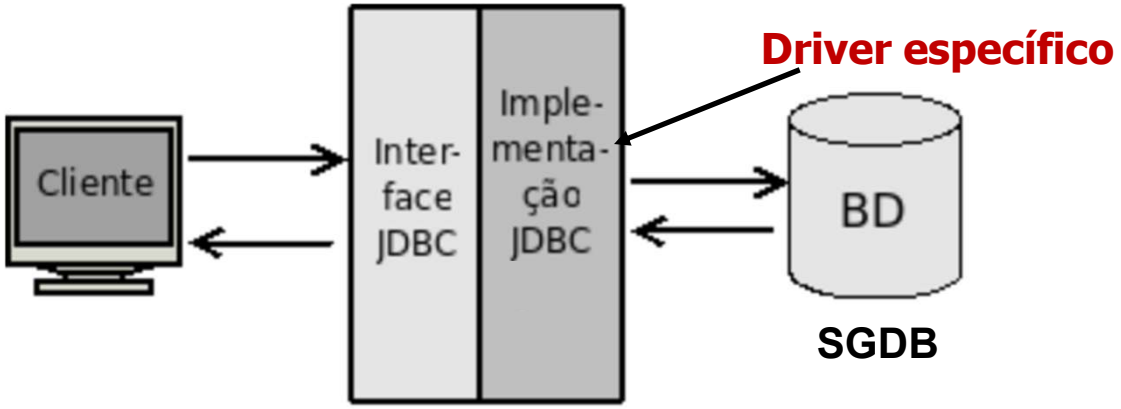
- O grande desafio em se misturar SQL com linguagens de programação de propósito geral está na incompatibilidade na forma como essas linguagens representam e manipulam dados.
- Em SQL, o principal tipo de dado é a relação (tabela). Comandos SQL operam com relações e retornam relações. Por outro lado, linguagens de programação normalmente operam uma variável por vez, correspondendo normalmente ao valor de um atributo de uma *tupla* de uma relação.
- Logo, a integração desses dois tipos de linguagens em uma única aplicação requer um mecanismo de **mapeamento do resultado de uma consulta** em uma **estrutura de dados** que o programa possa manipular.

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - Configuração JDBC.
   - Estabelecimento de conexão via JDBC.
   - Executando comandos SQL via JDBC.
   - Programas exemplo.
   - Execução de consultas.
   - Conjuntos de resultados roláveis e atualizáveis.
   - Row sets.

Visão Geral

- O padrão JDBC define uma API (*Application Program Interface*) que programas Java usam para submeter comandos SQL a um SGBD, bem como um gerenciador de drivers que permite acesso a diferentes SGBDs.
- Os desenvolvedores de SGBDs fornecem drivers próprios que são registrados por um mecanismo simples com o gerenciador de drivers da API JDBC.



Tipos de drivers JDBC

- A maioria dos desenvolvedores de SGBDs fornecem um driver tipo 3 ou tipo 4 para acesso via API JDBC.

- Um **driver tipo 3** utiliza um middleware escrito em java puro, localizado no servidor, que traduz as chamadas JDBC para o protocolo específico do SGBD.

- Um **driver tipo 4** é uma biblioteca cliente Java pura que traduz solicitações JDBC diretamente para o protocolo específico do SGBD. Apresenta alta performance, pois elimina a necessidade de tradução ou middleware.

Passos Típicos de Interação com SGBD via JDBC

1. O programa primeiro **estabelece uma conexão** com o SGBD usando a API JDBC. Isso envolve especificar a **URL** de acesso ao SGBD (IP e porta), bem como **login** e **senha** de acesso ao banco de dados.
2. Uma vez estabelecida a conexão com o SGBD, o programa pode enviar consultas, atualizações e outros comandos SQL via API JDBC. A maioria dos comandos SQL pode ser incluído em um programa Java.
3. Os resultados são retornados pelo SGBD para a API JDBC que, por sua vez, os retorna ao programa na forma de estruturas de dados (e.g., `Resultset`).
4. Quando o programa não precisa mais acessar o banco de dados, ele fecha a conexão via API JDBC.

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - **Configuração JDBC.**
   - Estabelecimento de conexão via JDBC.
   - Executando comandos SQL via JDBC.
   - Programas exemplo.
   - Execução de consultas.
   - Conjuntos de resultados roláveis e atualizáveis.
   - Row sets.

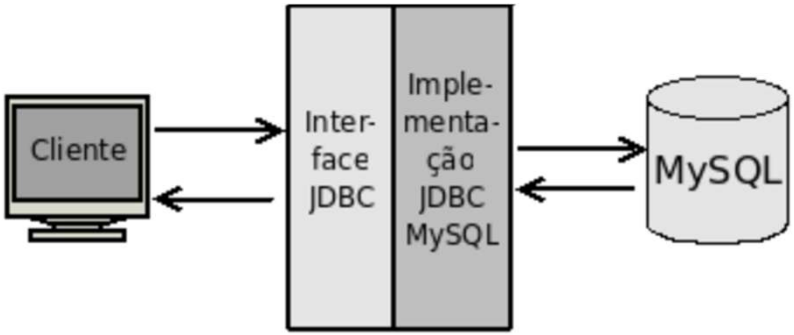
Escolhendo o SGBD

- Primeiro, precisamos de um SGBD para o qual um driver JDBC esteja disponível.

- Existem muitas opções excelentes, como IBM DB2, Microsoft SQL Server, MySQL, Oracle e PostgreSQL.

- Adotaremos o MySQL, pelos seguintes motivos:
  - Foi utilizado no curso anterior.
  - Gratuito.
  - Fácil de instalar e usar.

Driver JDBC MySQL



- Drivers JDBC são implementações das interfaces do pacote `java.sql`, as quais são normalmente disponibilizados na forma de um arquivo JAR (Java ARchive) pelos desenvolvedores de SGBD.
- MySQL implementa um driver JDBC tipo 4 que pode ser encontrado em: https://dev.mysql.com/downloads/connector/j/
- Detalhes sobre a instalação do driver dependem do ambiente de desenvolvimento, mas geralmente consiste em copiá-lo para o computador e adicionar o local ao `CLASSPATH` de java.

Registrando o Driver JDBC

- Muitos drivers JDBC registram-se automaticamente com o gerenciador de drivers usando SPI (*Service Provider Interface*). Para saber se o arquivo JAR faz o registro automático, faça a descompactação e verifique se ele contém o arquivo `META-INF/services/java.sql.Driver`.
  - O driver JDBC MySQL atual faz carregamento automático.
- Se o arquivo `jar` do driver **não fizer o registro automático**, então primeiro precisamos descobrir o nome da classe do driver JDBC.
- A forma mais simples de carregá-la é usando o método `Class.forname()`, tal como detalhado no slide a seguir.

Registro de Driver JDBC

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.SQLException;
4.
5. public class LoadDriver {
6.     public static void main(String[] args) {
7.         try {
8.            Class.forName("com.mysql.cj.jdbc.Driver").newInstance();
9.         } catch (Exception ex) {
10.           // tratamento do erro
11.        }
12.       ...
13.    }
14.}
```

Nota: no caso do MySQL não é necessário pois o registro é automático!

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - Configuração JDBC.
   - **Estabelecimento de conexão via JDBC.**
   - Executando comandos SQL via JDBC.
   - Programas exemplo.
   - Execução de consultas.
   - Conjuntos de resultados roláveis e atualizáveis.
   - Row sets.

Parâmetros da Conexão

- Um programa Java estabelece uma conexão via JDBC usando as seguintes informações:

1. **URL**: Consiste no endereço do host, número de porta e nome do banco de dados. URL MySQL típica: **"jdbc:mysql://localhost:3306/nome_bd"**
   - Para mais detalhes sobre URLS JDBC/MySQL acesse a documentação clicando aqui!
2. **User**: Nome de usuário para acesso ao banco de dados (login).
3. **Password**: Senha de acesso ao banco de dados.

Estabelecimento da Conexão

- Para estabelecer uma conexão usamos a classe `DriverManager`, que gerencia o conjunto de drivers carregados:

1. O cliente (programa Java) chama o método `getConnection()` de `DriverManager` passando a **URL**, **usuário** e **senha**.
2. `DriverManager` itera pelos drivers registrados a fim de encontrar um driver que use o subprotocolo especificado na URL.
3. Se tudo ocorrer bem, `getConnection()` retorna um objeto da classe `Connection`, representando a conexão.

- Com a conexão estabelecida, o programa Java passa a poder executar comandos SQL.

Estabelecimento de Conexão

```
1.  import java.sql.Connection;
2.  import java.sql.DriverManager;
3.  import java.sql.SQLException;
4.
5.  Connection conn = null;
6.  ...
7.  try {
8.      conn = DriverManager.getConnection(url, username, password);
9.      //faça algo com a conexão
10.     ...
11.} catch (SQLException ex) {
12.     System.out.println("SQLException: " + ex.getMessage());
13.     System.out.println("SQLState: " + ex.getSQLState());
14.     System.out.println("VendorError: " + ex.getErrorCode());
15.}
```

Exercício

- Crie um banco de dados MySQL chamado Lab1, inicialmente vazio.

```
create database Lab1;
```

- Estabeleça uma conexão com o banco de dados usando a API JDBC, tal como descrito nos slides anteriores. Faça isso de duas formas:
  1. Usando `jshell`.
     - `jshell --class-path /path/to/mysql-connector-java-x.x.xx.jar`
  2. Com um programa Java completo escrito na IDE de sua preferência.

- Verifique se a conexão está ativa usando o método `isValid()` do objeto `Connection`.

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - Configuração JDBC.
   - Estabelecimento de conexão via JDBC.
   - **Executando comandos SQL via JDBC.**
   - Programas exemplo.
   - Execução de consultas.
   - Conjuntos de resultados roláveis e atualizáveis.
   - Row sets.

Objetos `Statement`

- Objetos da classe `Statement` são usados para executar consultas SQL estáticas, bem como obter os resultados reportados pelo SGBD sobre essas consultas.

- São disponibilizados três métodos principais para executar consultas SQL:
  - `executeQuery(String sql)`
  - `executeUpdate(String sql)`
  - `execute(String sql)`

Objetos `Statement`

- `executeUpdate()` executa comandos SQL que modificam o banco de dados (INSERT, UPDATE, DELETE), bem como comandos DDL, tais como CREATE TABLE e DROP TABLE.
  - Retorna a contagem das tuplas afetadas pelo comando SQL, ou zero para comandos que não retornam a contagem de tuplas.
- `executeQuery()` executa uma consulta SQL (SELECT e suas variações) e retorna um objeto da classe `ResultSet` que permite percorrer tupla por tupla o resultado da consulta.
  - `ResultSet rs = stat.executeQuery("SELECT * FROM Books");`
- `execute()` permite executar comandos SQL arbitrários. Normalmente usado em aplicações interativas, onde o usuário fornece os comandos SQL.

Usando objetos `Statement`

1. Crie um objeto `Statement` usando o objeto de conexão (`Connection`):
   - `Statement stat = conn.createStatement();`

2. Descreva a requisição SQL usando uma `String`:
   - `String command = “select * from livro”;`
   - `String command = “Update livro Set num_pagina = 530” +“WHERE livro_id = 2”;`

3. Use o objeto `Statement` para chamar o método `executeQuery()`, se for uma consulta, ou `executeUpdate()`, para modificações do BD, passando como parâmetro a `String` que define o comando SQL.

Percorrendo um `ResultSet`

- Loop básico para percorrer um objeto `ResultSet`:

```
1. while (rs.next())
2. {
3.   //a cada iteração rs aponta para a
4.   //próxima tupla do conjunto
5. }
```

- Como veremos a seguir, há um grande número de métodos de acesso para inspecionar os atributos de cada tupla recuperada.

Acesso a Atributos

- Os métodos de acesso a atributos de tuplas de um `ResultSet` tem o seguinte formato padrão:
  - `Xxx getXxx(int col_num)` ou
  - `Xxx getXxx(String col_label)`
- `Xxx` se refere a um tipo, tal como `int`, `double`, `String` e `Date`. Por exemplo:
  - `String isbn = rs.getString(1); //primeiro atributo da tupla`
  - `double preco = rs.getDouble("preco"); //atributo chamado preco`
- O método `int findColumn(String col_name)` também costuma ser útil. Ele retorna o índice da coluna com nome col_name.

Gerenciando `Connections`, `Statements` e `ResultSets`

- Um objeto `Connection` pode criar um ou mais objetos `Statement`. Um objeto `Statement` pode ser usado em várias consultas SQL não relacionadas. Porém, um `Statement` tem **no máximo** um `ResultSet` aberto.

- Certifique-se de concluir o processamento de qualquer `ResultSet` antes de emitir uma nova consulta SQL pelo mesmo objeto `Statement`, pois os `ResultSets` de consultas anteriores **são fechados** automaticamente.

- Quando terminar de usar um `ResultSet`, `Statement` ou `Connection`, chame o método `close()`. Esses objetos usam grandes estruturas de dados que sugam os recursos finitos do servidor de banco de dados.

Gerenciando `Connections`, `Statements` e `ResultSets`

- É recomendado criar a conexões usando `try-with-resources.`

```
1.try (ResourceType resource = new ResourceType()) {
2.    // Utilize o recurso
3.} catch (ExceptionType e) {
4.    // Tratamento de exceção
5.}
```

- **try-with-resources** é uma estrutura de controle de Java que facilita o gerenciamento automático de recursos, tais como conexões com banco de dados, que precisam ser fechados após o uso. Com ele, vazamentos de recursos são evitados e o código fica mais limpo e menos propenso a erros.

Gerenciando `Connections`, `Statements` e `ResultSets`

- O trecho de código a seguir ilustra como usar `try-with-resources` para criar um objeto de conexão com o banco de dados.

```
1.try (Connection conn = DriverManager.getConnection(...))
2.{
3.    String queryString = “Select ...”;
4.    Statement stat = conn.createStatement();
5.    ResultSet result = stat.executeQuery(queryString);
6.    //processa o resultado da consulta
7.}
```

- Ao finalizar o bloco, a conexão é **automaticamente** fechada!

Analisando Exceções SQL

- A classe `SQLException` agora implementa a interface `Iterable<Throwable>`.
- O método `iterator()` produz um `Iterator<Throwable>` que possibilita iterar pelas cadeias de exceções, começando pela cadeia de causa da primeira `SQLException`, seguindo para a próxima `SQLException`, e assim por diante.
- Essa funcionalidade facilita inspecionar a cadeia de exceções `SQLException` de maneira conveniente e clara.
- A seguir veremos um exemplo.

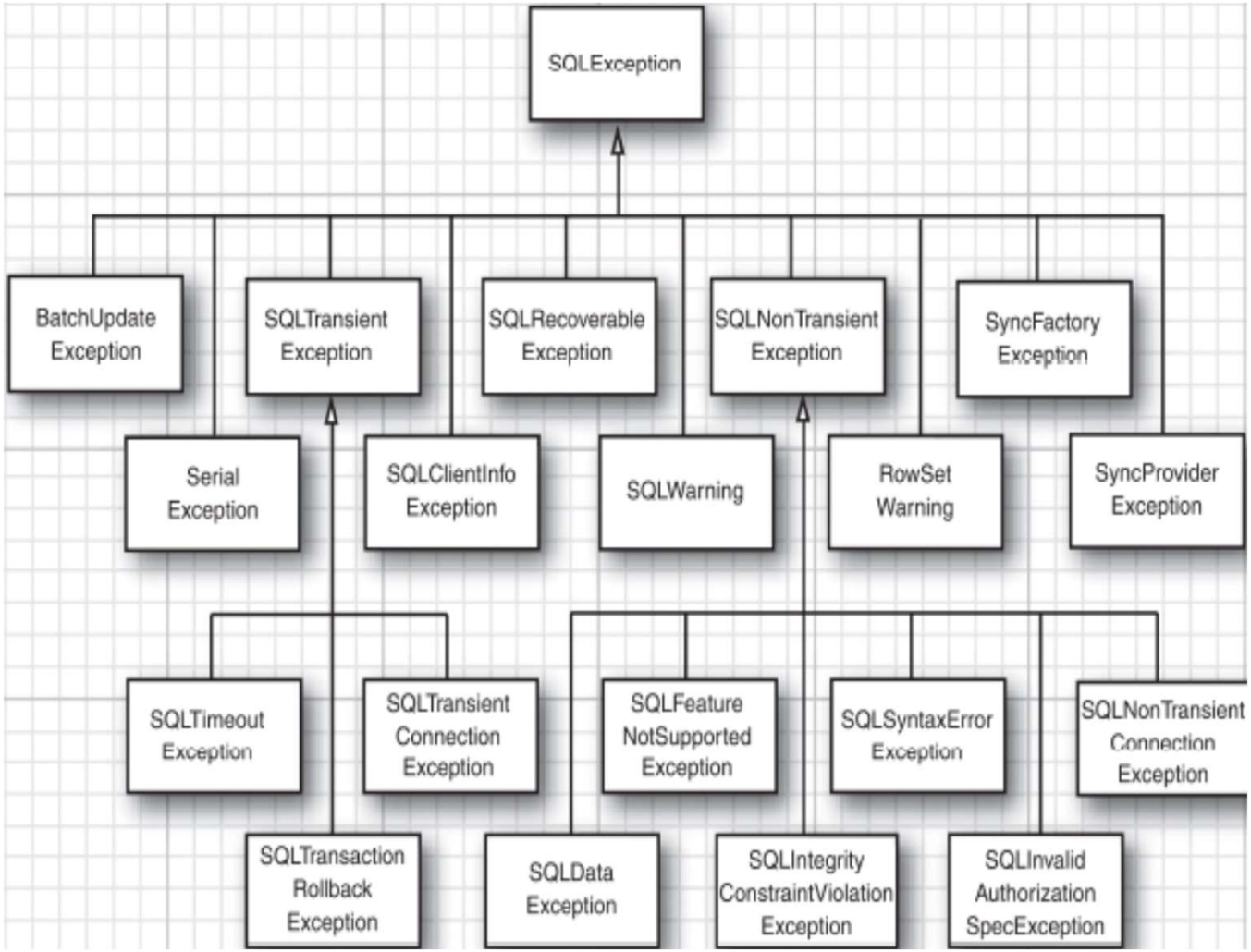
Inpecionando Exceções SQL

- Para percorrer as cadeias de exceções, usamos um loop `for`:

```
for (Throwable exception : SQLExceptionObj)
{
   //faça algo com t
}
```

- Para analisar uma `SQLException`, chame `getSQLState()` e `getErrorCode()`. O primeiro método produz uma string padronizada por X/Open ou SQL:2003 e o segundo é um código de erro específico do desenvolvedor do SGBD.

Exceções SQL são organizadas em uma hierarquia de herança, possibilitando detectar erros específicos de maneira independente de fornecedores de SGBD.



Warnings SQL

- O driver JDBC do SQGBD pode reportar condições não fatais como warnings (avisos). É possível recuperar warnings de Connections, Statements e ResultSets.

- A classe `SQLWarning` é uma subclasse de `SQLException`, mas SQLWarnings não são lançados como exceções.

- Chame `getSQLState()` e `getErrorCode()` para obter detalhes sobre os warnings. O slide a seguir apresenta um exemplo.

SQL Warnings

- Semelhante às exceções SQL, os warnings são encadeados. Para recuperá-los, use um loop:

```
1. Statement stat = ...
2. ...
3. SQLWarning w = stat.getWarning();
4. while (w != null)
5. {
6.    //faça algo com w
7.    w.nextWarning();
8. }
```

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - Configuração JDBC.
   - Estabelecimento de conexão via JDBC.
   - Executando comandos SQL via JDBC.
   - **Programas exemplo.**
   - Execução de consultas.
   - Conjuntos de resultados roláveis e atualizáveis.
   - Row sets.

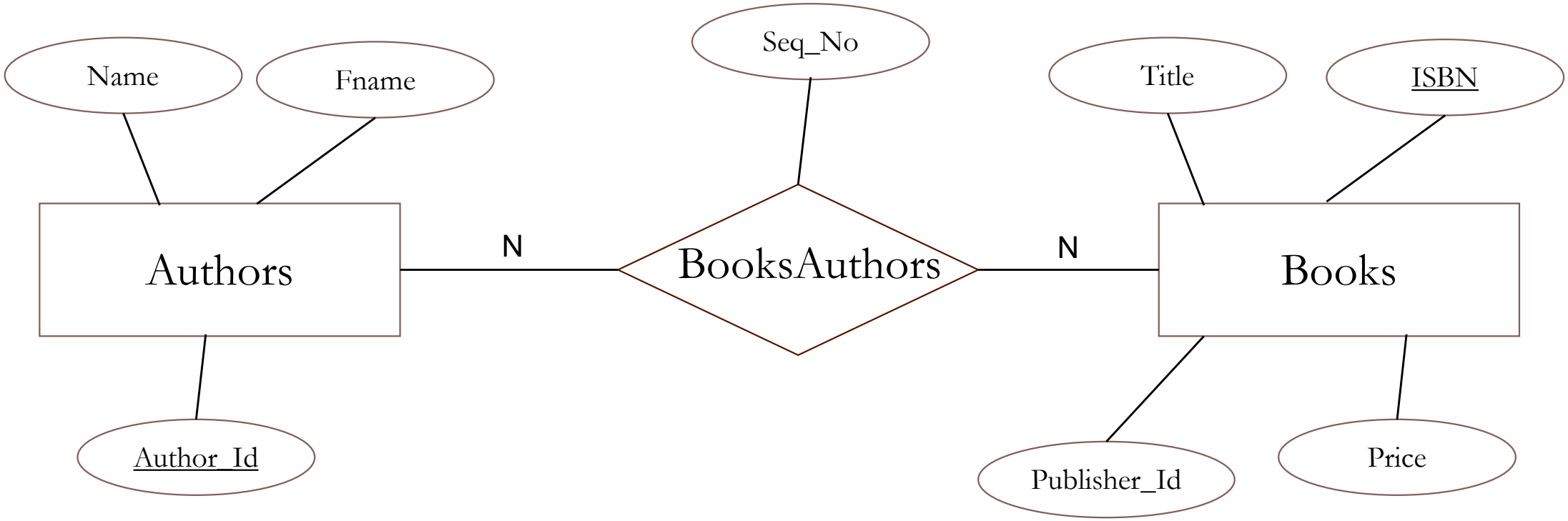
Exemplo 1: Localizado pasta TestaBD no AVA

- O programa `TestaBD.java` cria uma conexão com o SGBD MySQL e faz as seguintes operações:
  - **Cria** uma tabela chamada **Saudacao** com um único atributo chamado **Mensagem**.
  - **Insere** uma tupla na tabela **Saudacao**.
  - **Consulta** a tupla inserida e mostra no terminal.
  - **Deleta** a tabela **Saudacao**.

- O projeto (vs code) está localizado na pasta `TestaBD`.

- A configuração, explicação do código e execução do programa serão apresentados **em aula**.

Exemplo 2: MER do BD Author_Book



Mapeamento

- Como exercício, faça o mapeamento do MER do BD Author_Book para o modelo relacional.

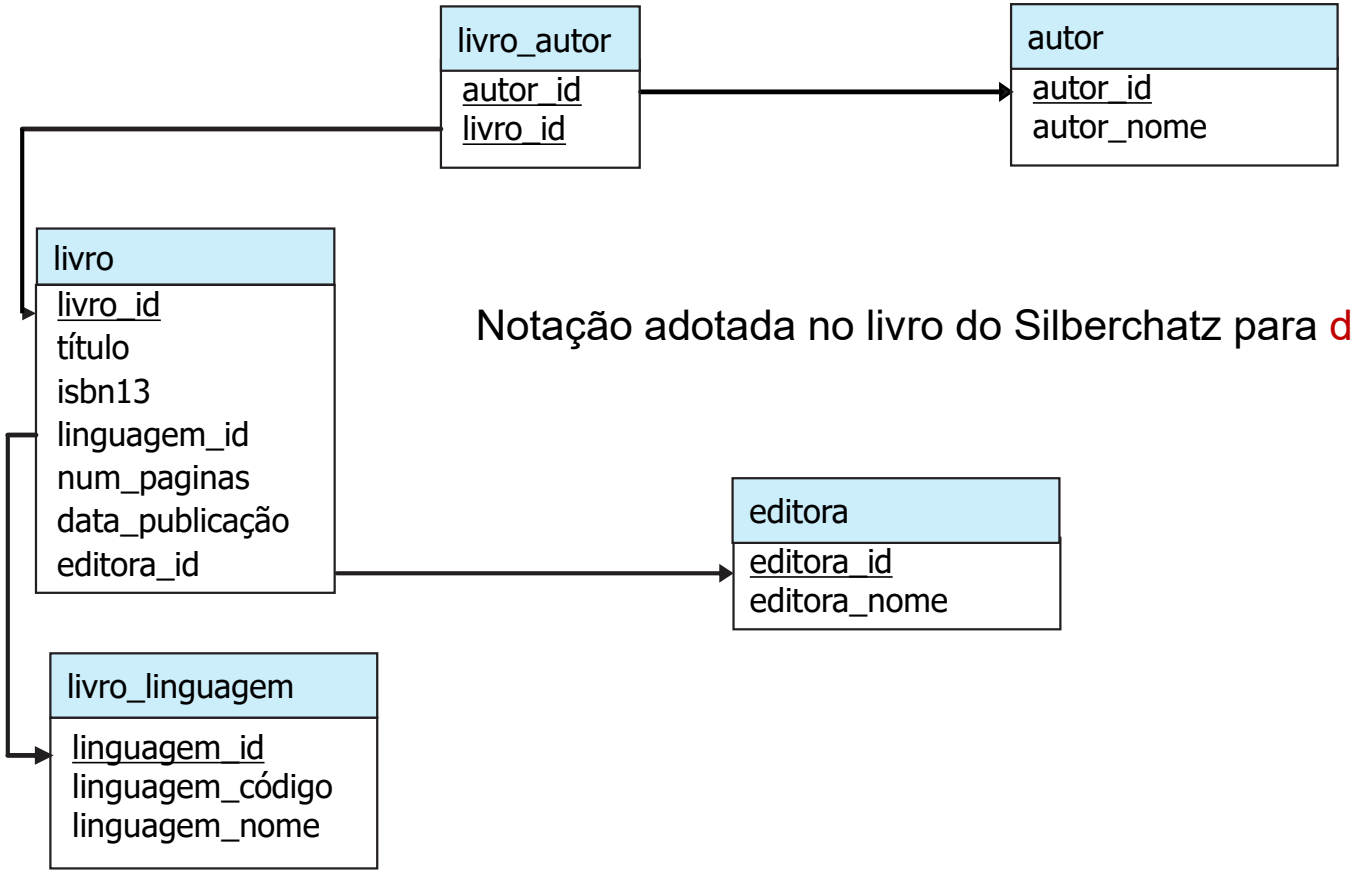
- O modelo relacional desse banco de dados será implementando e povoado a usando um programa Java.

- A solução será apresentada em aula.

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - Configuração JDBC.
   - Estabelecimento de conexão via JDBC.
   - Executando comandos SQL via JDBC.
   - Programas exemplo.
   - **Execução de consultas.**
   - Conjuntos de resultados roláveis e atualizáveis.
   - Row sets.

Banco de Dados Exemplo: `Catalogo_Livros`



Notação adotada no livro do Silberchatz para diagrama relacional!

Banco de Dados Exemplo: `Catalogo_Livros`

- O banco de dados Catalogo_Livros deve ser criado com permissões apropriadas para **consultar**, **atualizar**, **criar** e **deletar** tabelas.
- No site da disciplina, na pasta **bd_catalogo**, está disponível um conjunto de scripts para a criação e povoamento do banco de dados:
  - O arquivo `create_tables.sql` cria as tabelas.
  - O arquivo `drop_tables.sql` deleta as tabelas.
  - Os arquivos `povoa_autor.sql`, `povoa_livro.sql`, `povoa_editora.sql`, `povoa_livro_autor.sql` e `povoa_livro_linguagem.sql` inserem informações nas respectivas tabelas.
- Crie o banco de dados `Catalogo_Livros`!

Exercícios

- Usando o **jshell --class-path**, estabeleça uma conexão com o BD Catalogo_Livros e realize as seguintes consultas usando objetos `Statement`:

1. Consulte as informações sobre os livros publicados pelo author "Garrison Keillor".

2. Consulte as informações sobre os livros publicados pela editora "Addison Wesley".

Exercícios

1. Consulte as informações sobre os livros publicados pelo author “Garrison Keillor”.

```
1. SELECT L.titulo, L.isbn13, L.num_paginas
2. FROM autor as A NATURAL JOIN livro_autor as LA NATURAL JOIN livro as L
3. WHERE A.autor_nome = ‘Garrison Keillor’;
```

Exercícios

1. Consulte as informações sobre os livros publicados pela editora Addison Wesley.

```
1.  SELECT L.titulo, L.isbn13, L.num_paginas
2.  FROM editora as E NATURAL JOIN livro as L
3.  WHERE E.editora_nome = ‘Addison Wesley’;
```

Objeto `PreparedStatement`

- Já vimos como executar consultas usando um objeto `Statement`. Há uma outra forma de submeter consultas, usando comandos pré-processados usando objetos `PreparedStatement`.
- Principais diferenças entre **`Statement`** e **`PreparedStatement`**:
  - `Statement`:
    - Usado para executar consultas SQL estáticas, que não mudam.
    - Ideal para executar consultas simples que são executadas uma única vez ou ocasionalmente.
  - `PreparedStatement`:
    - Usado para consultas SQL dinâmicas que aceitam parâmetros.
    - Ideal para consultas que serão executadas repetidamente, especialmente em situações onde os valores dos parâmetros podem mudar.

Objeto `PreparadeStatement`

- Considere a consulta no banco de dados `Catalogo_Livros` que recupera todos os livros de uma editora específica, independentemente do autor:

```
1. SELECT L.titulo, L.data_publicao
2. FROM livro as L, editora as E
3. WHERE L.editora_id = E.editora_id AND E.nome = ?;
```

- Em vez de criar um comando de consulta separado toda vez que o usuário fizer essa consulta, podemos preparar a consulta com um objeto `PreparedStatement` e usá-la várias vezes, cada vez passando uma string diferente.

Usando `PreparadeStatement`

- Consultar o BD usando `PreparedStatement` melhora o desempenho, pois sempre que o SGBD executa uma consulta ele primeiro faz um planejamento de como executá-la de forma eficiente. Quando a consulta é preparada e reutilizada, a etapa de planejamento é realizada apenas uma vez.

- Consultas preparadas usam variáveis host. Cada variável host é indicada com uma ?.

- Se houver mais de uma variável host, as posições das ? devem ser conhecidas para a vinculação dos valores. A seguir um exemplo mais detalhado.

Usando `PreparadeStatement`

- Antes de executar o comando, é preciso usar um método `set()` para vincular as variáveis host aos valores reais. Existem diferentes métodos `set()` para vários tipos de dados. Aqui, queremos definir uma String para o nome de uma editora:

```
1. String cmd = "SELECT L.titulo, L.data_publicao"
2.              + "FROM livro as L, editora as E"
3.              + "WHERE L.editora_id = E.editora_id AND"
4.              + "E.editora_nome = ?";
5. PreparedStatement stat = conn.prepareStatement(cmd);
```

```
stat.setString(1, nome_da_editora);
```

- Com as variáveis vinculadas a valores, podemos executar o comando preparado:

```
ResultSet rs = stat.executeQuery();
```

Usando `PreparadeStatement` (Cont.)

- Abaixo um outro exemplo que supõe a existência da tabela `instructor`. Observe que chamamos `executeUpdate()`, e não executeQuery(), porque INSERT não retorna um ResultSet. O valor de retorno do executeUpdate() é a contagem de linhas alteradas.

```
PreparedStatement pStmt = conn.prepareStatement(
                    "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

Gerenciando `PreparadeStatement`

- Um objeto `PreparedStatement` torna-se inválido depois que o objeto `Connection` associado é fechado. No entanto, muitos bancos de dados armazenam em cache os comandos pré-preparados.

- Por isso, se a mesma consulta preparada for reutilizada, o banco de dados pode simplesmente reutilizar a estratégia de consulta.

- Portanto, não se preocupe com a sobrecarga de chamar `PrepareStatement`.

Exemplo 3: pasta `TestaConsultas` no AVA

- O programa Testaconsulta.java, de acordo com uma seleção feita pelo usuário, executa consultas e atualizações no banco de dados Catalogo_Livros:
  - Consulta todos os livros.
  - Consulta livros de um autor especificado.
  - Consulta livros de um autor especificado publicados por uma editora especificada.
  - Consulta todos os livros publicados por uma editora especificada.

- Para funcionar o banco de dados Catalogo_Livros deve estar ativo e povoado.
- A explicação do código e execução do programa são apresentados **em aula**.

Lendo LOBs

- Além de números, strings e datas, muitos bancos de dados podem armazenar objetos grandes (LOBs), como imagens ou outros dados.
- No SQL, objetos binários grandes são chamados de BLOBs e objetos grandes de caracteres são chamados de CLOBs.
- Para ler um LOB, execute o comando SELECT e chame o método `getBlob()` ou `getClob()` do `ResultSet`. Isso te dará um objeto do tipo `Blob` ou `Clob`.
  - Para obter os dados binários de um Blob, chame o método `getBytes()` ou `getBinaryStream()`.
  - Da mesma forma, se você recuperar um objeto `Clob`, use o método `getSubString()` ou `getCharacterStream()`.

Exemplo: Lendo um Blob

- Supondo existência de uma tabela `capa_livro(livro_id, capa)` que armazene imagens da capa dos livro do banco de dados `Catalogo_Livros`.

```
1. String query = "SELECT capa FROM capa_livro WHERE livro_id=?”;
2. PreparedStatement stat = conn.prepareStatement(query);
3. stat.set(1, isbn);
4. try (ResultSet result = stat.executeQuery())
5. {
6.  if (result.next())
7.  {
8.   Blob capaBlob = result.getBlob(1);
9.   Image coverImage = ImageIO.read(capaBlob.getBinaryStream());
10. }
11.}
```

Escrevendo LOBs

- Para colocar um LOB em um banco de dados, primeiro chame `createBlob()` ou `createClob()` do objeto `Connection`. Isso te dará uma stream de saída para o LOB. Daí é só gravá-lo. O exemplo abaixo mostra como armazenar um Blob:

```
1. Blob capaBlob = conn.createBlob();
2. int offset = 0;
3. OutputStream out = capaBlob.setBinaryStream(offset);
4. ImageIO.write(capaImage, "PNG", out);
5. String sqlInsert = ("INSERT INTO capa_livro VALUES (?,   ?)”);
6. PreparedStatement stat = conn.prepareStatement(sqlInsert);
7. stat.set(1, livro_id);
8. stat.set(2, capaBlob);
9. stat.executeUpdate();
```

Exercício

- Crie a tabela `capa_livro(livro_id, capa)` no banco de dados Catalogo_Livros. O atributo `livro_id` deve ser chave estrangeira para a tabela livro.
- Povoe a tabela `capa_livro` com algumas tuplas (pegue imagens de capas de livro da internet e associe a alguns livros).

- Consulte as capas com base no `livro_id` e apresente ao usuário.

SQL Scapes

- É tarefa do driver JDBC traduzir a sintaxe de escape comum para a sintaxe de um banco de dados específico.

- Escapes são fornecidos para os seguintes recursos:
  - Literais de data e hora
  - Chamadas de funções
  - Chamadas de procedimentos armazenados
  - Outer Joins
  - Caractere de escape em cláusulas LIKE

Veremos mais tarde!

SQL Scapes: Literais de data e hora

- Os literais de data e hora variam amplamente entre os bancos de dados. Para incorporar uma data ou hora literal, especifique o valor no formato ISO 8601 (www.cl.cam.ac.uk/~mgk25/iso-time.html).
- O driver irá traduzi-lo para o formato nativo. Use d, t, ts para valores DATE, TIME ou TIMESTAMP:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

SQL Scapes: Outer Joins

- Um `Outer Join` de duas tabelas não requer correspondência de tuplas de acordo com a condição de junção.

- A sintaxe de escape {oj ...} é necessária porque nem todos os bancos de dados usam uma notação padrão para essas junções.

```
SELECT *
FROM {oj livro LEFT OUTER JOIN editora ON
      livro.editora_id = editora.editora_id}
```

SQL Scapes em cláusulas LIKE

- Os caracteres `_` e `%` são usados corresponder a um único caractere ou a uma sequência de caracteres em cláusulas `LIKE`.
- Se for necessário buscar por literais `%` ou `_` na string, pode-se usar a cláusula `ESCAPE` no SQL, embora isso não seja sempre suportado diretamente via JDBC. Uma maneira de fazer isso é manualmente escapando esses caracteres no Java antes de definir os parâmetros.

```
1. String pattern = "100%";
2. pattern = pattern.replace("%", "\\%");
3. String sql = "SELECT * FROM livro WHERE titulo LIKE ? ESCAPE '\\'";
4. stmt = conn.prepareStatement(sql);
5. stmt.setString(1, pattern);
```

Recuperando Chaves Geradas Automaticamente

- A maioria dos bancos de dados oferece suporte a algum mecanismo para auto numeração de tuplas quando inserções são feitas no banco de dados.

- Esses números automáticos são frequentemente usados como chaves primárias, mas infelizmente os mecanismos de especificação diferem amplamente entre SGBDs.

- Embora o JDBC não ofereça uma solução independente de fornecedor para geração de chaves, no próximo slide é apresentada uma maneira eficiente de recuperá-las.

Recuperando Chaves Geradas Automaticamente

- Quando uma nova tupla é inserida em uma tabela e uma chave é gerada automaticamente, é possível recuperá-la com o seguinte código:

```
1. stat.executeUpdate(insertSql,Statement.RETURN_GENERATED_KEYS);
2. ResultSet rs = stat.getGeneratedKeys();
3. if (rs.next())
4. {
5.   int key = rs.getInt(1);
6.   ...
7. }
```

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - Configuração JDBC.
   - Estabelecimento de conexão via JDBC.
   - Executando comandos SQL via JDBC.
   - Programas exemplo.
   - Execução de consultas.
   - **Conjuntos de resultados roláveis e atualizáveis.**
   - Row sets.

Conjuntos de Resultados Roláveis e Atualizáveis

- Com o método `next()` da interface `ResultSet` podemos iterar pelas tuplas resultantes de uma consulta em um sentido.
- No entanto, considere a exibição dos resultados de uma consulta usando uma tabela com barra de rolagem. Normalmente, o usuário deseja avançar e retroceder no conjunto de resultados.
- Existem variações de ResultSets, os quais são roláveis (scrollable) e/ou atualizáveis. Ou seja, permitem avançar e retroceder nas tuplas e até mesmo pular para uma posição específica, bem como atualizar, excluir e inserir tuplas no conjunto de resultados.

Conjuntos de Resultados Roláveis e Atualizáveis

- Para realizar uma consulta com um conjunto de resultados rolável e/ou atualizável, é necessário obter um objeto `Statement` ou `PreparedStatement` apropriado:

```
Statement stat = conn.createStatement(type, concurrency);
```

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

- Os possíveis valores para `type` e `concurrency` estão listados nas tabelas apresentadas no próximo slide.

Conjuntos de Resultados Roláveis e Atualizáveis

`type`

Value	Explanation
`TYPE_FORWARD_ONLY`	The result set is not scrollable (default).
`TYPE_SCROLL_INSENSITIVE`	The result set is scrollable but not sensitive to database changes.
`TYPE_SCROLL_SENSITIVE`	The result set is scrollable and sensitive to database changes.

`concurrency`

Value	Explanation
`CONCUR_READ_ONLY`	The result set cannot be used to update the database (default).
`CONCUR_UPDATABLE`	The result set can be used to update the database.

Conjuntos de resultados roláveis e atualizáveis (Cont.)

- Com um conjunto de resultados rolável é possível navegar para frente com o método next() e no sentido contrário usando o método `previous()` ambos do objeto `ResultSet`.
- O tipo `CONCUR_UPDATABLE` permite atualizar, excluir e inserir `tuplas` no `ResultSet`. Depois de executar uma operação `UPDATE` ou `INSERT` em um conjunto de resultados, as alterações do BD são propagadas em uma etapa separada que pode ser ignorada, se desejado, para cancelar as alterações.
- Uma operação DELETE em um `ResultSet`, no entanto, é imediatamente executada (mas não necessariamente comitada) no banco de dados.

Prog. teste 4: Localizado pasta TestaRsRolavel

- O programa TestaRsRolavel.java cria um `Statement` rolável e percorre a tabela `livro` do banco de dados `catalogo_livro` em dois sentidos:
  - No sentido convencional usando o método `next()` do objeto `ResultSet`.
  - No sentido contrário usando o método `previous()` do objeto `ResultSet`.

- O arquivo TestaRsRolavel.java está localizado na pasta TestaRsRolavel.
- A explicação do código e execução do programa serão apresentados em aula.

Prog. teste 5: Localizado pasta TestaRsAtualizavel

- O programa TestaRsAtualizavel.java cria um `Statement` atualizável que modifica a tabela Empregado(emp_nu, emp_nome) do banco de dados Emp.
  - Primeiro ele insere 3 tuplas via `ResultSet`.
  - Em seguida ele deleta as 3 tuplas via `ResultSet`.
- O arquivo TestaRsAtualizavel.java está localizado na pasta TestaRsAtualizavel.
- A explicação do código e execução do programa serão apresentados em aula.

Agenda

1. Acesso a SQL via Linguagem de Programação.
2. **Java e a API JDBC.**
   - Configuração JDBC.
   - Estabelecimento de conexão via JDBC.
   - Executando comandos SQL via JDBC.
   - Programas exemplo.
   - Execução de consultas.
   - Conjuntos de resultados roláveis e atualizáveis.
   - **Row sets.**

Row Sets

- Os ResultSets roláveis e atualizáveis são poderosos, mas a conexão com o banco de dados precisa estar aberta durante toda a interação do usuário. Isso não é bom se o usuário se afastar do computador por muito tempo.
- Nessa situação, use um `RowSet`. A interface `RowSet` estende a interface `ResultSet`, mas um `RowSet` não precisa estar vinculado a uma conexão de banco de dados.
- Os RowSets também são adequados se você precisar mover um resultado de consulta para uma camada diferente de uma aplicação complexa ou para outro dispositivo, como um telefone celular. Nunca é bom fazer isso com um ResultSet, porque a estrutura de dados pode ser enorme, além de ser associada à conexão do banco de dados.

Construindo Row Sets

- O pacote `javax.sql.rowset` fornece interfaces que estendem `RowSet`:
  - `CachedRowSet permite a operação desconectada.`
  - `WebRowSet` é um `RowSet` em cache que pode ser salvo em um arquivo `XML`. O arquivo XML pode ser movido para outra camada de uma *app Web* onde é aberto por outro objeto `WebRowSet`.
  - `FilteredRowSet` e `JoinRowSet` suportam operações leves equivalentes às operações SQL SELECT e JOIN. Elas podem ser realizadas nos dados armazenados no `RowSet`, sem uma conexão com o BD.
  - Um `JdbcRowSet` é um wrapper em torno de um `ResultSet`. Ele adiciona métodos úteis da interface RowSet.

Construindo Row Sets (Cont.)

- Para obter um `RowSet` em cache, faça:

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

- Existem métodos semelhantes para obter os outros tipos de `RowSets`.

- No próximo slide detalharemos um pouco sobre `CachedRowSet`.

CachedRowSets

- Como CachedRowSet é uma subinterface da interface `ResultSet`, você pode usá-lo exatamente como usaria um `ResultSet`.
- Um `CachedRowSet` confere um benefício importante: você pode fechar a conexão e ainda usá-lo. Como veremos em nosso último programa exemplo, isso simplifica bastante a implementação de aplicações interativas. Cada comando de usuário simplesmente abre a conexão com o banco de dados, faz uma consulta, coloca o resultado em um `CachedRowSet` e fecha a conexão com o banco de dados.
- É possível, inclusive, modificar os dados em um `CachedRowSet`. Obviamente, as modificações não são refletidas imediatamente no BD; é necessário fazer uma solicitação explícita para aceitar as alterações acumuladas. Nesse caso, o `CachedRowSet` se reconecta ao BD e emite comandos SQL para gravar as alterações acumuladas.

CachedRowSets (Cont.)

- Podemos povoar um CachedRowSet com um `ResultSet`:

```
ResultSet result = . . .;
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
crs.populate(result);
conn.close(); //OK agora a conexão pode ser fechada
```

CachedRowSets (Cont.)

- Alternativamente, você pode permitir que o objeto `CachedRowSet` estabeleça uma conexão automaticamente. Para isso, primeiro configure os parâmetros do banco de dados:

```
crs.setURL("jdbc:mysql://localhost:3306/bd");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

CachedRowSets (Cont.)

- Em seguida, defina o comando de consulta e quaisquer parâmetros:

```
crs.setCommand("SELECT * FROM Books WHERE Publisher_ID = ?");
crs.setString(1, publisherId);
Crs.execute();
```

- Essa chamada estabelece uma conexão com o banco de dados, emite a consulta, preenche o `CachedRowSet` e desconecta.

CachedRowSets (Cont.)

- Se o resultado da consulta for muito grande, você provavelmente não quer sua totalidade, pois os usuários provavelmente só olharão para algumas linhas. Nesse caso, especifique um tamanho de página:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20); ...
crs.execute();
```

- Agora você terá apenas 20 tuplas. Para obter o próximo lote de tuplas, chame `crs.nextPage()`.

CachedRowSets (Cont.)

- Você pode inspecionar e modificar um `CachedRowSet` com os mesmos métodos usados para um `ResultSet`. Se você modificá-lo, deverá escrevê-lo de volta no BD chamando `crs.acceptChanges(conn)` ou `crs.acceptChanges()`.

- A segunda chamada funciona apenas se você configurou o `CachedRowSet` com as informações necessárias para se conectar a um banco de dados (como URL, nome de usuário e senha).

- Um `CachedRowSet` que contém o resultado de uma consulta complexa **não conseguirá** gravar suas alterações de volta no banco de dados. Por outro lado, é seguro se seu `CachedRowSet` contiver dados de uma única tabela.

Agenda

1. Revisão sobre Modelagem de Dados.
2. Características arquitetônicas de aplicações de Banco de Dados.
3. Acesso a Banco de Dados Relacional via Linguagens de Programação.
4. Java e a API JDBC.
    a. Configuração JDBC.
    b. Estabelecimento de conexão via JDBC.
    c. Executando comandos SQL via JDBC.
    d. Programas de teste.
    e. Banco de Dados para testes de consultas.
    f. Execução de consultas.
    g. Conjuntos de resultados roláveis e atualizáveis.
    h. Row sets.
    i. Metadados.

Metadados

- Já vimos como preencher, consultar e atualizar tabelas de banco de dados. No entanto, o JDBC pode fornecer informações adicionais sobre a estrutura de um banco de dados e suas tabelas. Por exemplo, podemos obter a lista de tabelas de um determinado banco de dados ou os nomes e tipos de coluna de uma tabela.
- Essas informações não são úteis quando você está implementando uma aplicação de negócios com um banco de dados predefinido. Porém, são extremamente úteis para programadores que escrevem ferramentas que funcionam com qualquer banco de dados.
- No SQL, os dados que descrevem o banco de dados ou uma de suas partes são chamados de metadados (para distingui-los dos dados reais armazenados no banco de dados). Podemos obter três tipos de metadados: sobre um BD, sobre um `ResultSet` e sobre parâmetros de instruções pré-preparadas.

JDBC: recuperação de metadados

- Para recuperar metadados sobre o banco de dados, crie um objeto do tipo `DatabaseMetaData` a partir do objeto de conexão com o banco de dados:

```
DatabaseMetaData meta = conn.getMetaData();
```

- Com isso estamos prontos para obter alguns metadados. Por exemplo:

```
ResultSet mrs = meta.getTables(String catalog, String schemaPattern,
        String tableNamePattern, new String[] { "TABLE" });
```

- Esse comando retorna um conjunto de resultados que contém informações sobre todas as tabelas do banco de dados. Cada linha no conjunto de resultados contém informações sobre uma tabela no BD. A terceira coluna é o nome da tabela. Consulte a documentação da API para outros parâmetros deste método.

JDBC: recuperação de metadados

- No exemplo abaixo o método `getMetaData()` retorna um objeto `ResultSetMetaData` que contém metadados da tabela do `ResultSet rs`.

```
ResultSet rs = stat.executeQuery("SELECT * FROM " + nome_tabela);
ResultSetMetaData meta = rs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
  String nomeColuna = meta.getColumnLabel(i);
  String tipo = meta.getColumnTypeName(i);
  int    larguraColuna = meta.getColumnDisplaySize(i);
  ...
}
```