JDBC

FAPESC – DESENVOLVEDORES PARA TECNOLOGIA DA INFORMAÇÃO

HERCULANO DE BIASI
herculano.debiasi@unoesc.edu.br

Tópicos

- Bibliografia
- JDBC
- *Drivers*
- Classes e *interfaces*
- Tipos de dados
- Programando com JDBC

BIBLIOGRAFIA

- TURINI, Rodrigo. **Explorando APIs e bibliotecas Java**: JDBC, IO, Threads, Java FX e mais. Casa do Código, São Paulo: 2017.

UNOESC

Apoiadores:

JDBC

- À medida que se tornava necessário que a aplicação acessasse bancos de dados diferentes, era preciso incluir as rotinas para cada um deles em sua base de código, o que gerava dois problemas
  - Uma base de código grande e difícil de manter
  - Necessidade de distribuir os *drivers* de cada banco de dados com a aplicação

- Para facilitar a vida dos programadores, foi criado para o Java, uma API (*Application Programming Interface*) chamada Java *Database Connectivity* (JDBC), que disponibiliza uma interface de programação padrão para acesso aos bancos de dados que é agnóstica quanto ao *driver* de banco de dados a ser utilizado, permitindo que os programadores consigam utilizar vários tipos de bancos de dados de maneira transparente

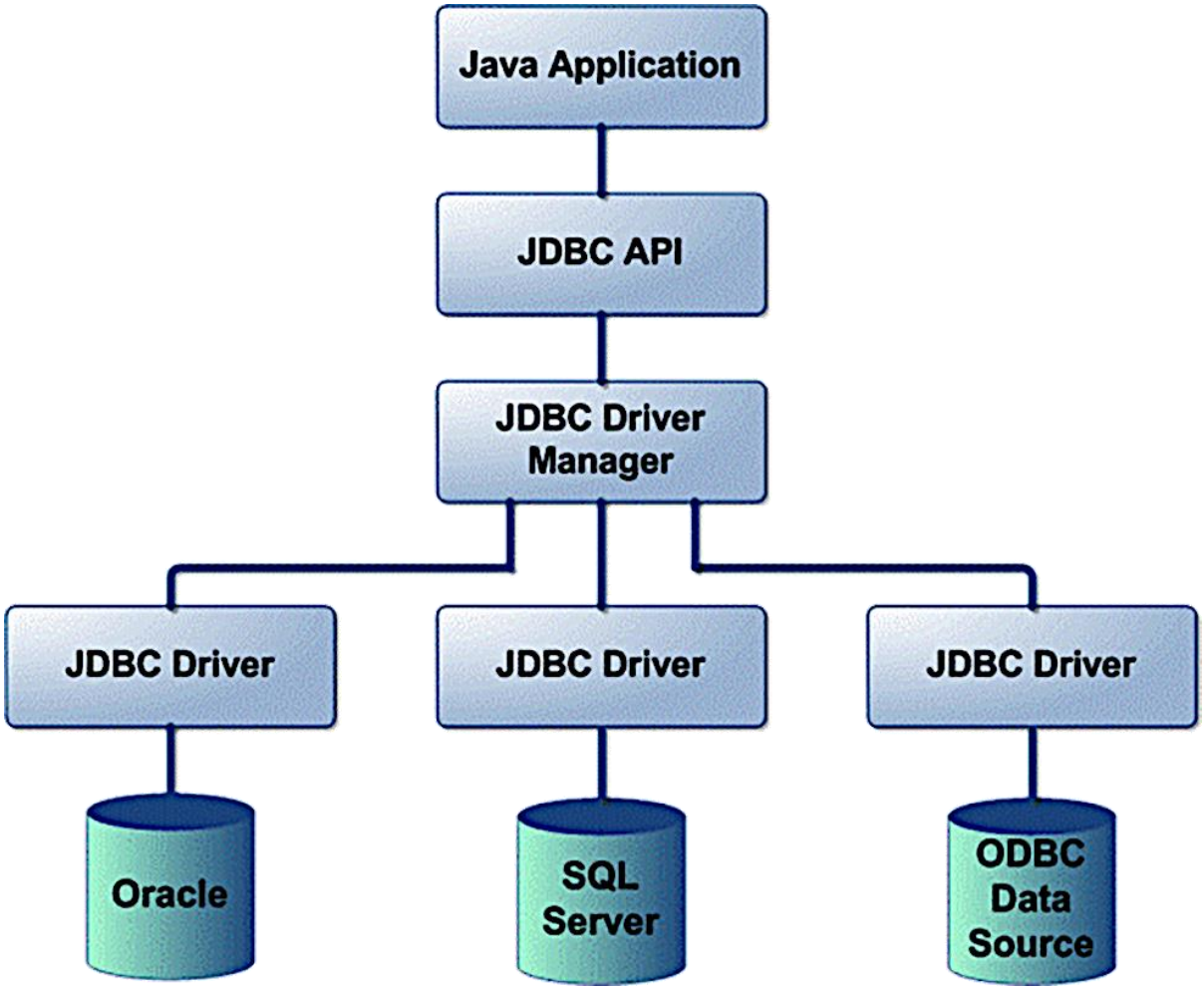
JDBC

- JDBC (*Java Database Connectivity*) é a API padrão do Java, mantido pela Oracle, para persistência em banco de dados

- Uma API padrão permite o acesso padronizado a bancos de dados diferentes, favorecendo a manutenibilidade, para bancos como
  - Microsoft SQL Server (MSSQL)
  - Oracle
  - MySQL
  - PostgreSQL

JDBC

- Esquema de conexão



UNOESC

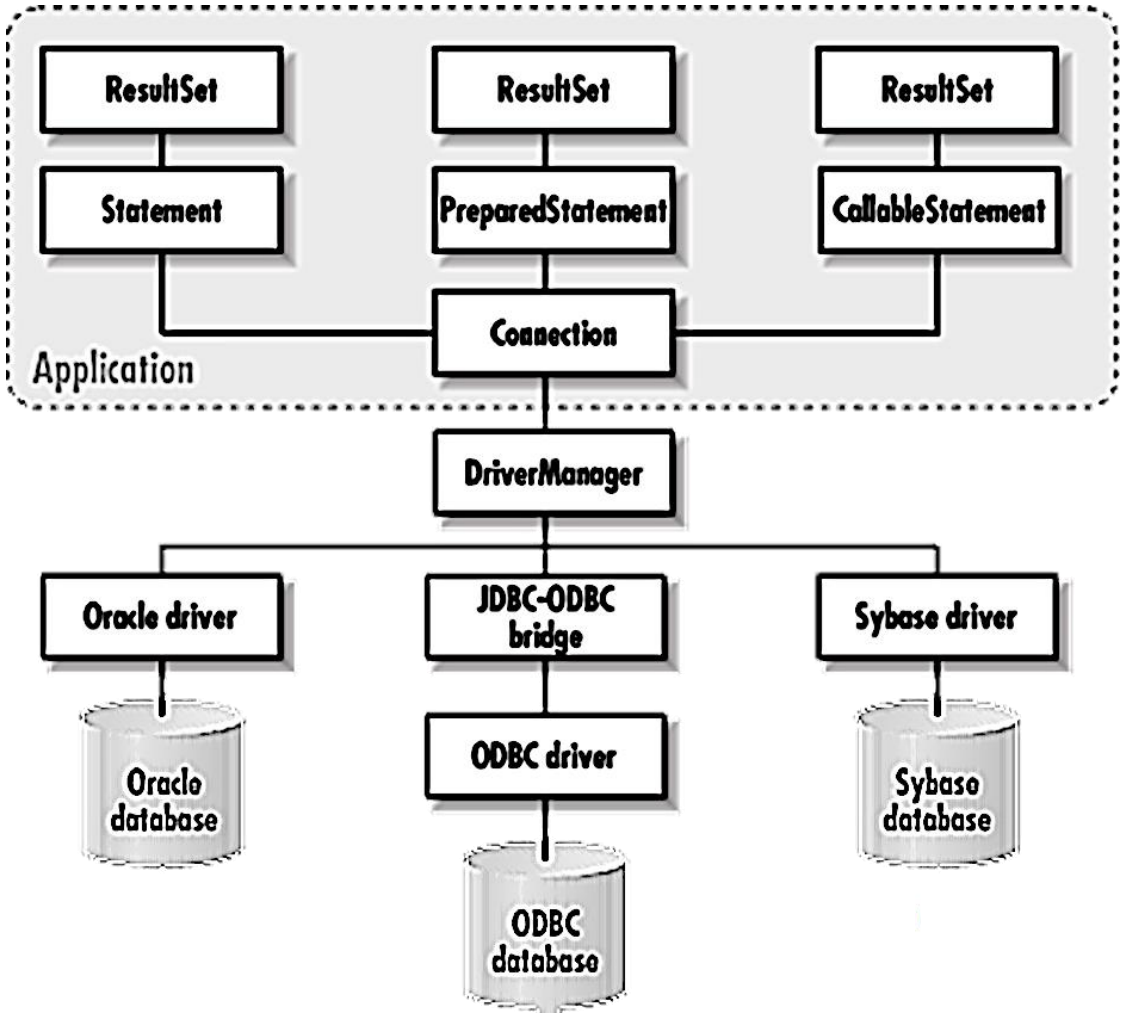
Drivers

- Para conectar um aplicativo Java a um banco de dados usando JDBC, é necessário usar um *driver* JDBC, que atua como um intermediário entre o aplicativo e o BD
  - *Drivers* JDBC são componentes de software que permitem que aplicações Java interajam com implementações específicas de banco de dados
  - O *driver* JDBC fornece a conexão ao banco de dados e implementa o protocolo para transferir a consulta e o resultado entre cliente e banco de dados
  - Existem vários tipos de *drivers* JDBC disponíveis, portanto, é preciso escolher aquele que melhor se adapte às circunstâncias
  - Deve-se estar ciente que nem todos os tipos de *drivers* são suportados por um SGBD específico, como por exemplo, a Oracle e, mesmo quando um tipo de *driver* é suportado pelo SGBD, pode não ser suportado por todas as suas versões

- *Drivers* podem ser baixados normalmente no site do fabricante do banco de dados
  - O *driver* do MySQL (MySQL *connector*) pode ser baixado no site http://www.mysql.org

CLASSES E *INTERFACES*

- Arquitetura da API JDBC



UNOESC

Classes e *Interfaces*

- `Connection`: Define métodos para executar uma *query* (como um `INSERT` e `SELECT`), comitar uma transação, fechar a conexão, entre outros

- Métodos
  - `close()`: Fecha uma conexão
  - `createStatement()`: Cria um objeto `Statement` que será usado para enviar expressões SQL para o banco
  - `isClosed()`:Verifica se a conexão está fechada
  - `isReadOnly()`:Verifica se a conexão é somente leitura
  - `prepareCall(String sql)`: Cria um objeto para execução de *stored procedures*
  - `prepareStatement(String sql)`: Cria um objeto semelhante ao criado por `createStatement()`, porém permite trabalhar com *queries* parametrizadas

CLASSES E *INTERFACES*

- `SQLException`: Classe de exceção do tipo *checked*, lançada por muitos dos métodos da API de JDBC
  - Em uma aplicação real devemos utilizar `try/catch` nos lugares que julgamos haver possibilidade de recuperar de uma falha com o banco de dados
  - É preciso tomar sempre cuidado para fechar todas as conexões que foram abertas

- Na aplicação, ao se fazer um `try/catch` em `SQLException` e relançando-a como uma `RuntimeException` desacopla o código que chama a fábrica de conexões da API de JDBC

- É uma boa prática de desenvolvimento, toda vez que for necessário lidar com uma `SQLException`, relançá-la como `RuntimeException`

Classes e *Interfaces*

- `DriverManager`: Responsável por se comunicar com todos os *drivers* disponíveis
- Método
  - `getConnection`: Método estático responsável por conectar ao banco
- *String* de conexão: `jdbc:tipo_banco://ip:porta/nome_do_banco`
  - `jdbc`: Protocolo
  - `tipo_banco`: Subprotocolo, podendo ser, entre outros, `mysql` ou `postgresql`
  - `ip`: Endereço do servidor (IP ou localhost)
  - `porta`: Informação obrigatória caso não esteja se usando a porta padrão do SGBD e opcional caso porta em uso seja a 3306 para o MySQL ou a 5432 para o PostgreSQL
  - `nome_do_banco`: Nome do banco de dados

CLASSES E *INTERFACES*

- `DriverManager`: Responsável por se comunicar com todos os *drivers* disponíveis

SGBD	URL de Conexão
Oracle	`jdbc:oracle:thin:<usuário>/<senha>@<servidor>:<porta>:<serviço>`
MySQL	`jdbc:mysql://<servidor>:<porta>/<banco>`
PostgreSQL	`jdbc:postgresql://<servidor>:<porta>/<banco>`
SQL Server	`jdbc:sqlserver://<servidor>:<porta>[;propriedade=valor][;propriedade=valor]`
DB2	`jdbc:db2://<servidor>:<porta>/<banco>`

Classes e *Interfaces*

- `DriverManager`: Responsável por se comunicar com todos os *drivers* disponíveis
  - Até a versão 3 do JDBC, antes de chamar o `DriverManager.getConnection()` era necessário registrar o driver JDBC que iria ser utilizado através do método `Class.forName("org.postgresql.Driver")`, no caso do PostgreSQL, que carregava essa classe, e essa se comunicava com o `DriverManager`
  - A partir do JDBC 4, que está presente no Java 6, esse passo não é mais necessário
  - Para usar o JDBC em projetos com Java 5 ou mais antigo, será preciso fazer o registro do *driver* JDBC, carregando a sua classe, que vai se registrar no `DriverManager`
  - Isso também pode ser necessário em alguns servidores de aplicação e servidores *web*, como no Tomcat 7 ou posterior

CLASSES E *INTERFACES*

- `Statement`: Executa instruções e realiza consultas

- `PreparedStatement`: Subclasse de `Statement` mas possui métodos para inserir parâmetros para as instruções
  - Os parâmetros de são indicados por sinais de interrogação dentro da *string* que contém o comando SQL
  - A identificação é feita pela posição dentro da *string* SQL, iniciando pela posição 1
  - A *string* contendo o comando SQL deve ser fornecida no momento de criação do comando e não no momento de execução do comando

- Vantagens de `PreparedStatement`
  - Melhor performance pois os comandos são précompilados e otimizados
  - Ganhos de segurança minimizando ataques de injeção de SQL
  - Código mais legível

CLASSES E *INTERFACES*

- Métodos de `PreparedStatement`
  - `close()`: Fecha a conexão
  - `execute()`: Pode ser utilizado com qualquer instrução SQL e retorna um valor booleano (`true` indica que um objeto `ResultSet` pode ser recuperado e `false` indicando que a consulta retornou um valor `int` ou `void`)
  - `executeQuery()`: Executa uma instrução SQL que retorna um objeto `ResultSet`
  - `executeUpdate()`: Recebe o SQL que será executado; deve ser usado para comandos como `INSERT`, `UPDATE` ou `DELETE`; retorna um valor inteiro representando o número de registros afetados
  - `getGeneratedKeys()`: Obtém o valor gerado em campos de geração automática como o `auto_increment` do MySQL
  - `setXYZ()`: Define valores em um `PreparedStatement`, em que `XYZ` pode ser `String`, `Int`, `Long`, `Date`, `Float`, `Double`

Classes e *Interfaces*

- Métodos `SetXYZ()` de `PreparedStatement`

Método	Tipo SQL
`setASCIIStream()`	`LONGVARCHAR`
`setBigDecimal()`	`NUMERIC/DECIMAL`
`setBinaryStream()`	`LONGVARBINARY`
`setBoolean()`	`BIT`
`setByte()`	`TINYINT`
`setBytes()`	`VARBINARY/LONGVARBINARY`
`setDate()`	`DATE`
`setDouble()`	`DOUBLE`
`setFloat()`	`FLOAT`
`setInt()`	`INTEGER`
`setLong()`	`BIGINT`
`setObject()`	`Objeto`
`setNull`	`NULL`
`setShort()`	`SMALLINT`
`setString()`	`VARCHAR/LONGVARCHAR`
`setTime()`	`TIME`
`setTimestamp()`	`TIMESTAMP`

UNOESC

CLASSES E *INTERFACES*

- `ResultSet`: Responsável por manipular o conjunto de resultados de uma consulta
- Conceito de *cursor*, que é como um ponteiro para as linhas da tabela

Número das linhas

Cursor está na linha 0

0	id	nome	endereco	data_nascimento	salario
1	1	Fulano	Rua x.	1981-01-01	123.45
2	2	Beltrano	Rua y.	1982-02-02	234.56
3	3	Sicrano	Rua z.	1983-03-03	345.67

`ResultSet`

CLASSES E *INTERFACES*

- `ResultSet`: Responsável por manipular o conjunto de resultados de uma consulta

- Métodos
  - `absolute(posição)`: Move o cursor para uma posição (linha) específica
  - `close()`: Fecha o `ResultSet`, liberando a memória
  - `deleteRow()`: Remove a linha corrente do `ResultSet`
  - `beforeFirst()`: Move o cursor para antes do primeiro registro do `ResultSet`
  - `first()`: Move o cursor para a primeira linha do `ResultSet`
  - `last()`: Move o cursor para a última linha do `ResultSet`
  - `next()`: Move o cursor para o próximo registro do `ResultSet`
  - `previous()`: Move o cursor para a linha anterior do `ResultSet`
  - `getXYZ(coluna)`: Recupera valores da coluna, em que `XYZ` pode ser `String`, `Int`, `Long`, `Date`, `Float`, `Double`

Classes e *Interfaces*

- Métodos `getXYZ()` de `ResultSet`

Método	Tipo Java Retornado
`getASCIIStream()`	`java.io.InputStream`
`getBigDecimal()`	`java.math.BigDecimal`
`getBinaryStream()`	`java.io.InputStream`
`getBoolean()`	`boolean`
`getByte()`	`Byte`
`getBytes()`	`byte[]`
`getDate()`	`java.sql.Date`
`getDouble()`	`double`
`getFloat()`	`float`
`getInt()`	`int`
`getLong()`	`long`
`getObject()`	`Object`
`getShort()`	`short`
`getString()`	`java.lang.String`
`getTime()`	`java.sql.Time`
`getTimestamp()`	`java.sql.Timestamp`
`getUnicodeStream()`	`java.io.InputStream`

UNOESC

Tipos de Dados

- Conversão de dados entre SQL e o Java

SQL	Java (Tipo)	Java (Método)
CHAR/VARCHAR	String	setString()
INTEGER	Integer	setInt()
TINYINT	Boolean Integer	setBoolean()
SMALLINT	Integer	setInt()
BIGINT	Long	setLong()
FLOAT/REAL	Float	setFloat()
DOUBLE	Double	setDouble()
NUMERIC/DECIMAL	java.math.BigDecimal	new BigDecimal("valor")
CLOB/BLOB	java.sql.Clob java.sql.Blob byte[]	setClob()/setBlob()
DATE	java.sql.Date	setDate()
TIME	java.sql.Time	setTime()
TIMESTAMP	java.sql.Timestamp	setTimestamp()
RAW/LONGRAW BINARY/VARBINARY/LONGVARBINARY	byte[]	setBytes()

Programando com JDBC

- Fluxograma de atividades