

CAPÍTULO 2

Bancos de dados e JDBC

"O medo é o pai da moralidade"
— Friedrich Wilhelm Nietzsche

Ao término desse capítulo, você será capaz de:

- conectar-se a um banco de dados qualquer através da API JDBC;
- criar uma fábrica de conexões usando o design pattern Factory;
- pesquisar dados através de queries;
- encapsular suas operações com bancos de dados através de DAO – Data Access Object.

2.1 – POR QUE USAR UM BANCO DE DADOS?

Muitos sistemas precisam manter as informações com as quais eles trabalham, seja para permitir consultas futuras, geração de relatórios ou possíveis alterações nas informações. Para que esses dados sejam mantidos para sempre, esses sistemas geralmente guardam essas informações em um banco de dados, que as mantém de forma organizada e prontas para consultas.

A maioria dos bancos de dados comerciais são os chamados relacionais, que é uma forma de trabalhar e pensar diferente ao paradigma orientado a objetos.

O **MySQL** é o banco de dados que usaremos durante o curso. É um dos mais importantes bancos de dados relacionais, e é gratuito, além de ter uma instalação fácil para todos os sistemas operacionais. Depois de instalado, para acessá-lo via terminal, fazemos da seguinte forma:

```
mysql -u root
```

Banco de dados

Para aqueles que não conhecem um banco de dados, é recomendado ler um pouco sobre o assunto e também ter uma base de SQL para começar a usar a API JDBC.

O processo de armazenamento de dados é também chamado de **persistência**. A biblioteca de persistência em banco de dados relacionais do Java é chamada JDBC, e também existem diversas ferramentas do tipo **ORM** (*Object Relational Mapping*) que facilitam bastante o uso do JDBC. Neste momento, focaremos nos conceitos e no uso do JDBC. Veremos um pouco da ferramenta de ORM Hibernate ao final deste mesmo curso e, no curso FJ-25, com muitos detalhes, recursos e tópicos avançados.

2.2 – PERSISTINDO ATRAVÉS DE SOCKETS?

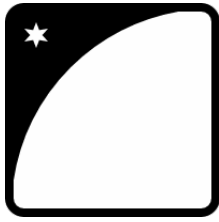
Para conectar-se a um banco de dados poderíamos abrir *sockets* diretamente com o servidor que o hospeda, por exemplo um Oracle ou MySQL e nos comunicarmos com ele através de seu protocolo proprietário.

Mas você conhece o protocolo proprietário de algum banco de dados? Conhecer um protocolo complexo em profundidade é difícil, trabalhar com ele é muito trabalhoso.

Uma segunda ideia seria utilizar uma API específica para cada banco de dados. Antigamente, no PHP, por exemplo, a única maneira de acessar o Oracle era através de funções como `oracle_connect`, `oracle_result`, e assim por diante. O MySQL tinha suas funções análogas, como `mysql_connect`. Essa abordagem facilita muito nosso trabalho por não precisarmos entender o protocolo de cada banco, mas faz com que tenhamos de conhecer uma API um pouco diferente para cada tipo de banco. Além disso, caso precisemos trocar de banco de dados um dia, precisaremos trocar todo o nosso código para refletir a função correta de acordo com o novo banco de dados que estamos utilizando.

Você pode também fazer o curso FJ-21 dessa apostila na Caelum

Querendo aprender ainda mais sobre Java na Web e Hibernate? Esclarecer



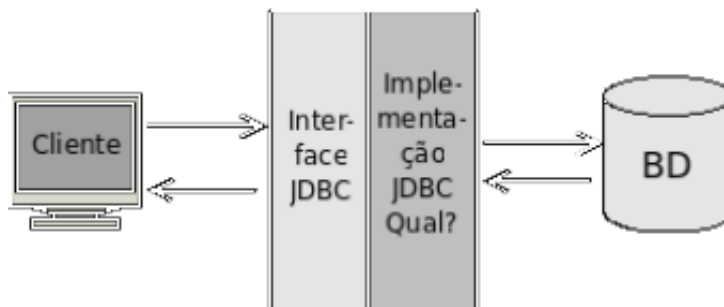
dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso FJ-21** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

[Consulte as vantagens do curso *Java para Desenvolvimento Web*.](#)

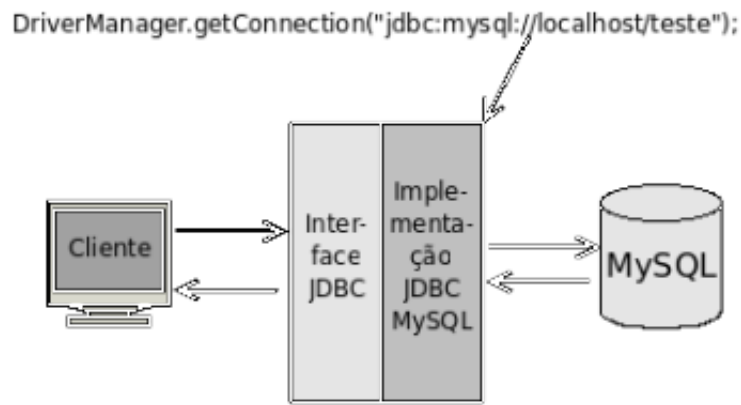
2.3 – A CONEXÃO EM JAVA

Conectar-se a um banco de dados com Java é feito de maneira elegante. Para evitar que cada banco tenha a sua própria API e conjunto de classes e métodos, temos um único conjunto de interfaces muito bem definidas que devem ser implementadas. Esse conjunto de interfaces fica dentro do pacote `java.sql` e nos referiremos a ela como **JDBC**.



Entre as diversas interfaces deste pacote, existe a interface `Connection` que define métodos para executar uma query (como um `insert` e `select`), comitar transação e fechar a conexão, entre outros. Caso queiramos trabalhar com o MySQL, precisamos de classes concretas que implementem essas interfaces do pacote `java.sql`.

Esse conjunto de classes concretas é quem fará a ponte entre o código cliente que usa a API JDBC e o banco de dados. São essas classes que sabem se comunicar através do protocolo proprietário do banco de dados. Esse conjunto de classes recebe o nome de **driver**. Todos os principais bancos de dados do mercado possuem **drivers JDBC** para que você possa utilizá-los com Java. O nome driver é análogo ao que usamos para impressoras: como é impossível que um sistema operacional saiba conversar com todo tipo de impressora existente, precisamos de um driver que faça o papel de "tradutor" dessa conversa.



Para abrir uma conexão com um banco de dados, precisamos utilizar sempre um driver. A classe `DriverManager` é a responsável por se comunicar com todos os drivers que você deixou disponível. Para isso, invocamos o método estático `getConnection` com uma `String` que indica a qual banco desejamos nos conectar.

Essa `String` – chamada de **String de conexão JDBC** – que utilizaremos para acessar o MySQL tem sempre a seguinte forma:

```
jdbc:mysql://ip/nome_do_banco
```

Devemos substituir `ip` pelo IP da máquina do servidor e `nome_do_banco` pelo nome do banco de dados a ser utilizado.

Seguindo o exemplo da linha acima e tudo que foi dito até agora, seria possível rodar o exemplo abaixo e receber uma conexão para um banco MySQL, caso ele esteja rodando na mesma máquina:

```
public class JDBCExemplo {
    public static void main(String[] args) throws SQLException {
        Connection conexao = DriverManager.getConnection(
            "jdbc:mysql://localhost/fj21");
        System.out.println("Conectado!");
        conexao.close();
    }
}
```

Repare que estamos deixando passar a `SQLException`, que é uma exception *checked*, lançada por muitos dos métodos da API de JDBC. Numa aplicação real devemos utilizar `try/catch` nos lugares que julgamos haver possibilidade de recuperar de uma falha com o banco de dados. Também precisamos tomar sempre cuidado para fechar todas as conexões que foram abertas.

Ao testar o código acima, recebemos uma exception. A conexão não pôde ser aberta. Recebemos a mensagem:

```
java.sql.SQLException: No suitable driver found for
jdbc:mysql://localhost/fj21
```

Por que?

O sistema ainda não achou uma implementação de **driver JDBC** que pode ser usada para abrir a conexão indicada pela URL `jdbc:mysql://localhost/fj21`.

O que precisamos fazer é adicionar o driver do MySQL ao *classpath*, o arquivo **.jar** contendo a implementação JDBC do MySQL (*mysql connector*) precisa ser colocado em um lugar visível pelo seu projeto ou adicionado à variável de ambiente `CLASSPATH`. Como usaremos o Eclipse, fazemos isso através de um clique da direita em nosso projeto, *Properties/Java Build Path* e em *Libraries* adicionamos o jar do driver JDBC do MySQL. Veremos isto passo a passo nos exercícios.

E o `Class.forName`?

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("com.mysql.jdbc.Driver")`, no caso do MySQL, 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. Mas lembre-se: caso você utilize JDBC em um projeto com Java 5 ou anterior, 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 web, como no Tomcat 7 ou posterior, por proteção para possíveis vazamentos de memória:

<http://bit.ly/18BpDfG>

Alterando o banco de dados

Teoricamente, basta alterar as duas `strings` que escrevemos para mudar de um banco para outro. Porém, não é tudo tão simples assim! O código SQL que veremos a seguir pode funcionar em um banco e não em outros. Depende de quão aderente ao padrão ANSI SQL é seu banco de dados.

Isso só causa dor de cabeça e existem projetos que resolvem isso, como é o caso do Hibernate (www.hibernate.org) e da especificação JPA (Java Persistence API).

Veremos um pouco do Hibernate ao final desse curso e bastante sobre ele no FJ-25.

Drivers de outros bancos de dados

Os drivers podem ser baixados normalmente no site do fabricante do banco de dados.

Alguns casos, como no Microsoft SQL Server, existem outros grupos que desenvolvem o driver em <http://jtds.sourceforge.net>. Enquanto isso, você pode achar o driver do MYSQL (chamado de mysql connector) no site <http://www.mysql.org>.

2.4 – FÁBRICA DE CONEXÕES

Em determinado momento de nossa aplicação, gostaríamos de ter o controle sobre a construção dos objetos da nossa classe. Muito pode ser feito através do construtor, como saber quantos objetos foram instanciados ou fazer o log sobre essas instâncias.

Às vezes, também queremos controlar um processo muito repetitivo e trabalhoso, como abrir uma conexão com o banco de dados. Tomemos como exemplo a classe a seguir que seria responsável por abrir uma conexão com o banco:

```
public class ConnectionFactory {  
    public Connection getConnection() {  
        try {  
            return DriverManager.getConnection(  
                "jdbc:mysql://localhost/fj21", "root", "");  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Poderíamos colocar um aviso na nossa aplicação, notificando todos os programadores a adquirir uma conexão:

```
Connection con = new ConnectionFactory().getConnection();
```

Note que o método `getConnection()` é uma fábrica de conexões, isto é, ele cria novas conexões para nós. Basta invocar o método e recebemos uma conexão pronta para uso, não importando de onde elas vieram e eventuais detalhes de

criação. Portanto, vamos chamar a classe de `ConnectionFactory` e o método de `getConnection`.

Encapsulando dessa forma, podemos mais tarde mudar a obtenção de conexões, para, por exemplo, usar um mecanismo de *pooling*, que é fortemente recomendável em uma aplicação real.

Tratamento de Exceções

Repare que estamos fazendo um try/catch em `SQLException` e relançando-a como uma `RuntimeException`. Fazemos isso para que o seu código que chamará a fábrica de conexões não fique acoplado com a API de JDBC. Toda vez que tivermos que lidar com uma `SQLException`, vamos relançá-las como `RuntimeException`.

Poderíamos ainda criar nossa própria exceção que indicaria que ocorreu um erro dentro da nossa Factory, algo como uma `ConnectionFactoryException`.

2.5 – DESIGN PATTERNS

Orientação a objetos resolve as grandes dores de cabeça que tínhamos na programação procedural, restringindo e centralizando responsabilidades.

Mas alguns problemas não podemos simplesmente resolver com orientação a objetos, pois não existe palavra chave para uma funcionalidade tão específica.

Alguns desses pequenos problemas aparecem com tanta frequência que as pessoas desenvolvem uma solução "padrão" para ele. Com isso, ao nos defrontarmos com um desses problemas clássicos, podemos rapidamente implementar essa solução genérica com uma ou outra modificação, de acordo com nossa necessidade. Essa solução padrão tem o nome de **Design Pattern** (**Padrão de Projeto**).

A melhor maneira para aprender o que é um **Design Pattern** é vendo como surgiu sua necessidade.

A nossa `ConnectionFactory` implementa o design pattern Factory que prega o encapsulamento da construção (fabricação) de objetos complicados.

A bíblia dos Design Patterns

O livro mais conhecido de Design Patterns foi escrito em 1995 e tem trechos de código em C++ e Smalltalk. Mas o que realmente importa são os conceitos e os diagramas que fazem desse livro independente de qualquer linguagem. Além de tudo, o livro é de leitura agradável.

Design Patterns, Erich Gamma *et al.*

Tire suas dúvidas no novo GUJ Respostas



O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

[Faça sua pergunta.](#)

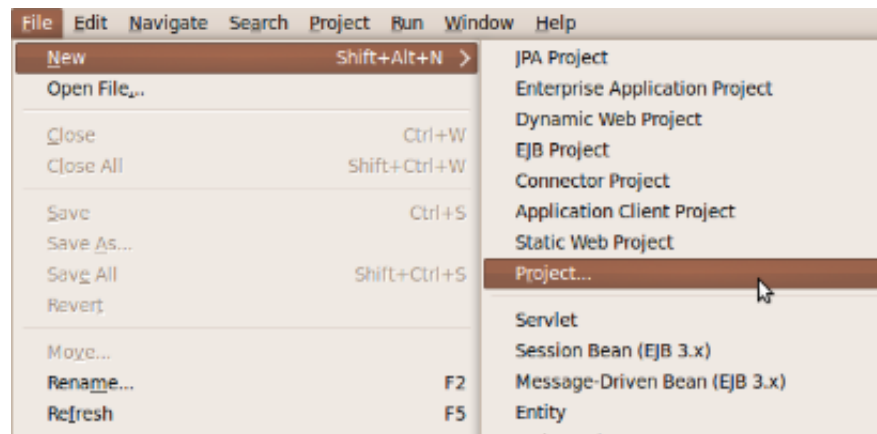
2.6 – EXERCÍCIOS: CONNECTIONFACTORY

1. Nos computadores da Caelum, clique no ícone do Eclipse no Desktop;

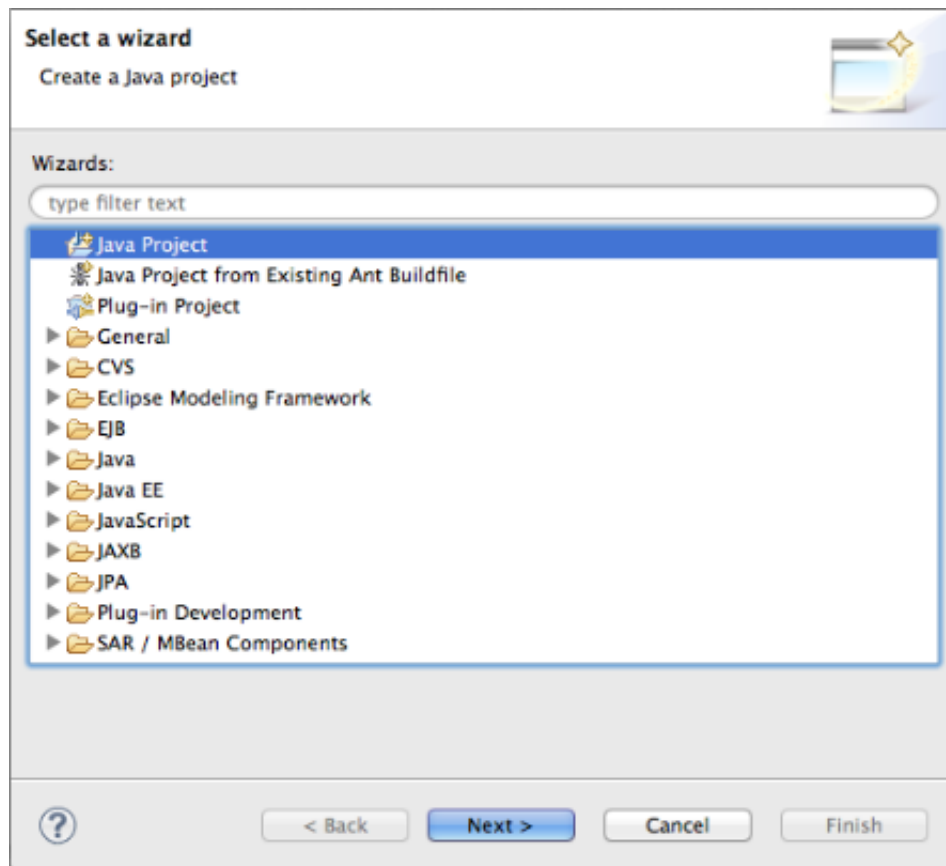
Baixando o Eclipse em casa

Estamos usando o Eclipse for Java EE Developers. Você pode obtê-lo direto no site do Eclipse em www.eclipse.org

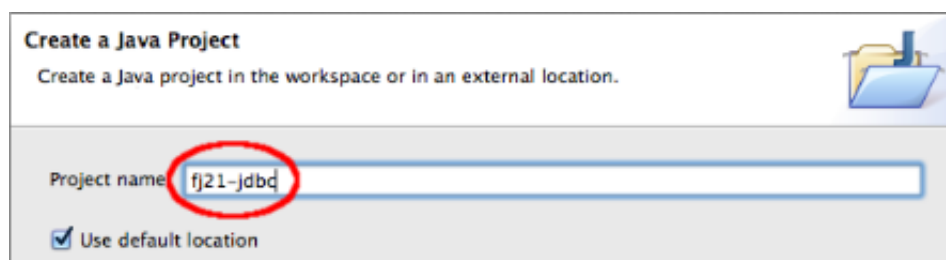
- a. Feche a tela de *Welcome* caso ela apareça
- b. Vamos criar um projeto no Eclipse chamado `fj21-jdbc`.
- c. Vá em **File -> New -> Project:**



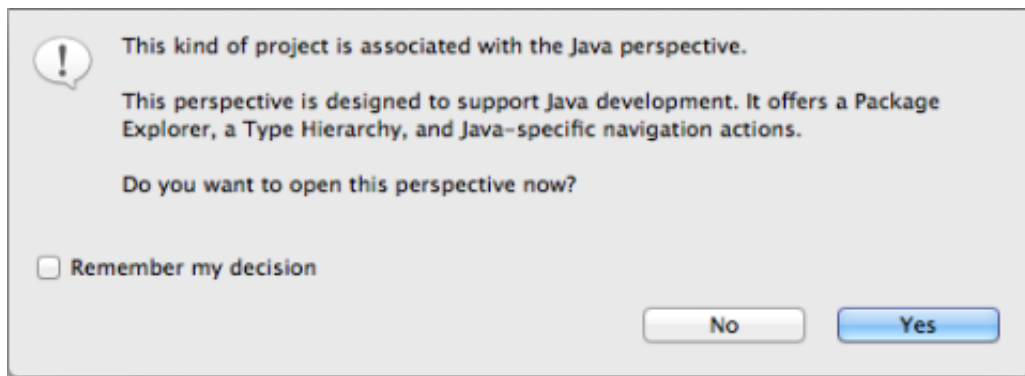
d. Selecione **Java Project** e clique em **Next**:



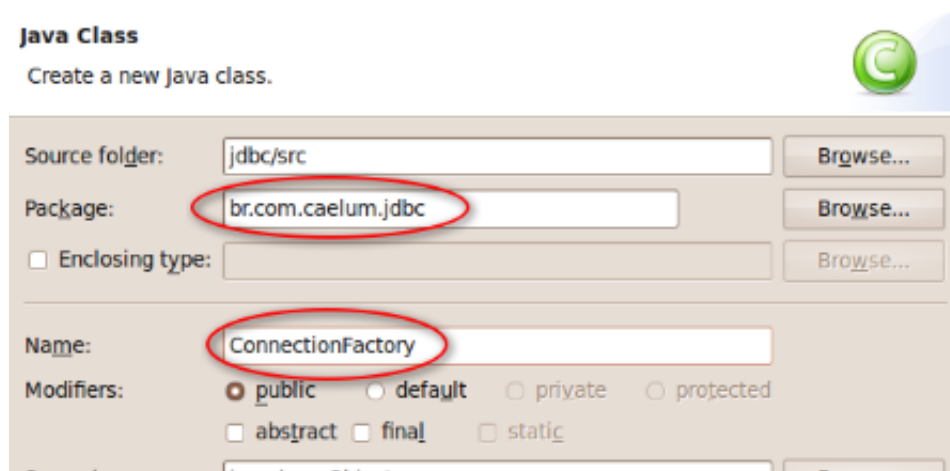
e. Coloque o nome do projeto como **fj21-jdbc** e clique em **Finish**:



f. Aceite a mudança de perspectiva:



2. Copie o driver do MySQL para o seu projeto.
 - a. no seu Desktop, clique na pasta **Caelum/21**;
 - b. clique da direita no driver do MySQL mais novo, escolha Copy;
 - c. vá para sua pasta principal (webXXX) na coluna da direita do *File Browser*;
 - d. entre no diretório workspace, fj21-jdbc;
 - e. clique da direita e escolha Paste: você acaba de colocar o arquivo ".jar" no seu projeto.
3. Vamos criar a classe que fabrica conexões:
 - a. Clique em **File -> New -> Class**.
 - b. Crie-a no pacote `br.com.caelum.jdbc` e nomeie-a como `ConnectionFactory`.



- c. Clique em **Finish**
 - d. No código, crie o método `getConnection` que retorna uma nova conexão. Quando perguntado, importe as classes do pacote `java.sql` (cuidado para não importar NADA de `com.mysql`).

```
1 public Connection getConnection() {  
2     try {
```

```

3         return DriverManager.getConnection(
4     "jdbc:mysql://localhost/fj21", "root", "");
5     } catch (SQLException e) {
6         throw new RuntimeException(e);
7     }
8 }

```

4. Crie uma classe chamada TestaConexao no pacote `br.com.caelum.jdbc.teste`. Todas as nossas classes de teste deverão ficar nesse pacote.

- Crie um método `main` dentro da classe. Use o atalho do Eclipse para ajudar.
- Dentro do `main`, fabrique uma conexão usando a `ConnectionFactory` que criamos. Vamos apenas testar a abertura da conexão e depois fechá-la com o método `close`:

```

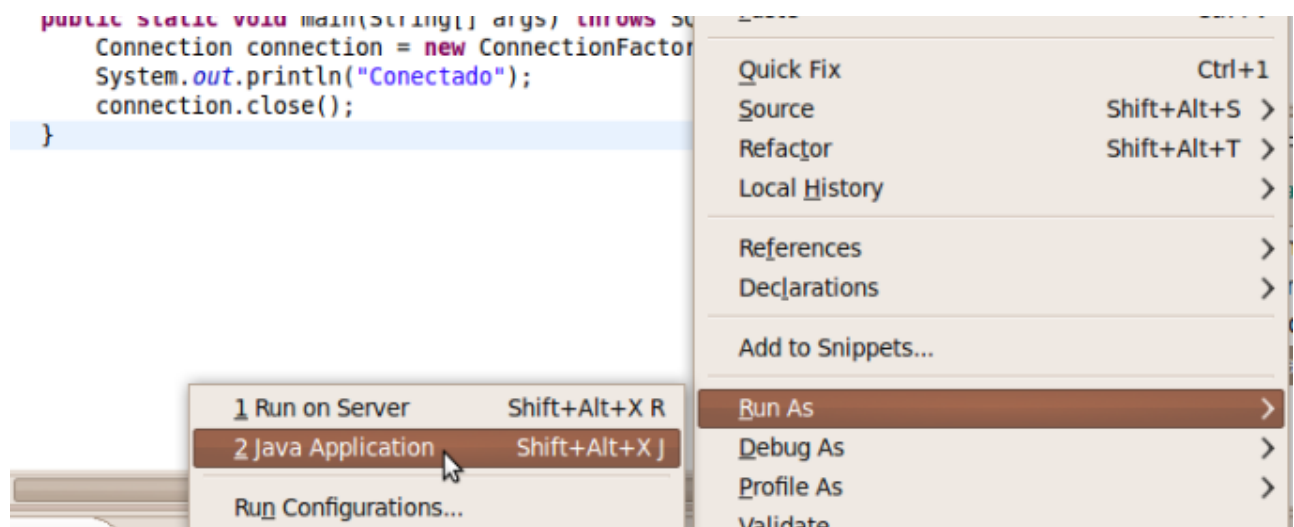
Connection connection = new ConnectionFactory().getConnection();
System.out.println("Conexão aberta!");
connection.close();

```

- Trate os erros com `throws`. (Use: `Ctrl + 1` e escolha "add throws declaration").

5. Rode a sua classe TestaConexao pelo Eclipse.

- Clique da direita na sua classe TestaConexao
- Escolha **Run as, Java Application** (caso prefira, aprenda a tecla de atalho para agilizar nas próximas execuções)



6. Parece que a aplicação não funciona pois o driver não foi encontrado? Esquecemos de colocar o JAR no classpath! (*Build Path* no Eclipse)

- Clique no seu projeto com o botão da direita e escolha *Refresh* (ou pressione `F5`).

b. Selecione o seu driver do MySQL, clique da direita e escolha **Build Path, Add to Build Path**:



c. Rode novamente sua aplicação TestaConexao agora que colocamos o driver no classpath.

2.7 – A TABELA CONTATO

Para criar uma tabela nova, primeiro devemos acessar o terminal e fazermos o comando para logarmos no mysql.

```
mysql -u root
```

Nos preparamos para usar o banco de dados **fj21**:

```
use fj21;
```

A seguinte tabela será usada nos exemplos desse capítulo:

```
create table contatos (  
  id BIGINT NOT NULL AUTO_INCREMENT,  
  nome VARCHAR(255),  
  email VARCHAR(255),  
  endereco VARCHAR(255),  
  dataNascimento DATE,  
  primary key (id)  
);
```

No banco de dados relacional, é comum representar um contato (entidade) em uma tabela de contatos.

2.8 – JAVABEANS

O que são Javabeans? A pergunta que não quer se calar pode ser respondida

muito facilmente uma vez que a uma das maiores confusões feitas aí fora é entre Javabeans e Enterprise Java Beans (EJB).

Javabeans são classes que possuem o construtor sem argumentos e com métodos de acesso do tipo `get` e `set`! Mais nada! Simples, não? Já os EJBs costumam ser javabeans com características mais avançadas e são o assunto principal do curso FJ-31 da Caelum.

Podemos usar *beans* por diversos motivos, normalmente as classes que representam nosso modelo de dados são usados dessa forma.

Utilizaremos:

- uma classe com métodos do tipo `get` e `set` para cada um de seus parâmetros, que representa algum objeto;
- uma classe com construtor sem argumentos que representa uma coleção de objetos.

A seguir, você vê um exemplo de uma classe `JavaBean` que seria equivalente ao nosso modelo de entidade do banco de dados:

```
package br.com.caelum.jdbc.modelo;

public class Contato {

    private Long id;
    private String nome;
    private String email;
    private String endereco;
    private Calendar dataNascimento;

    // métodos get e set para id, nome, email, endereço e dataNascimento

    public String getNome() {
        return this.nome;
    }
    public void setNome(String novo) {
        this.nome = novo;
    }

    public String getEmail() {
        return this.email;
    }
    public void setEmail(String novo) {
        this.email = novo;
    }

    public String getEndereco() {
        return this.endereco;
    }
```

```
}  
public void setEndereco(String novo) {  
    this.endereco = novo;  
}  
  
public Long getId() {  
    return this.id;  
}  
public void setId(Long novo) {  
    this.id = novo;  
}  
  
public Calendar getDataNascimento() {  
    return this.dataNascimento;  
}  
public void setDataNascimento(Calendar dataNascimento) {  
    this.dataNascimento = dataNascimento;  
}  
}
```

A especificação JavaBeans é muito grande e mais informações sobre essa vasta área que é a base dos componentes escritos em Java pode ser encontrada em:

<http://java.sun.com/products/javabeans>

Métodos getters e setters

Um erro muito comum cometido pelos desenvolvedores Java é a criação dos métodos getters e setters indiscriminadamente, sem ter a real necessidade da existência tais métodos.

Existe um artigo no blog da caelum que trata desse assunto:

<http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/>

Os cursos FJ-11 e FJ-22 também mostram isso claramente quando criam algumas entidades que não possuem apenas getters e setters.

Se você quer saber mais sobre Enterprise Java Beans (EJB), a Caelum oferece o curso FJ-31. Não os confunda com Java Beans!

Nova editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não



conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

2.9 – INSERINDO DADOS NO BANCO

Para inserir dados em uma tabela de um banco de dados entidade-relacional basta usar a cláusula **INSERT**. Precisamos especificar quais os campos que desejamos atualizar e os valores.

Primeiro o código SQL:

```
String sql = "insert into contatos " +  
    "(nome,email,endereco, dataNascimento)" +  
    " values ('" + nome + "', '" + email + "', '" +  
    endereco + "', '" + dataNascimento + "');"
```

O exemplo acima possui três pontos negativos que são importantíssimos. O primeiro é que o programador que não escreveu o código original não consegue bater o olho e entender o que está escrito. O que o código acima faz? Lendo rapidamente fica difícil. Mais difícil ainda é saber se faltou uma vírgula, um fecha parênteses talvez? E ainda assim, esse é um caso simples. Existem tabelas com 10, 20, 30 e até mais campos, tornando inviável entender o que está escrito no SQL misturado com as concatenações.

Outro problema é o clássico "preconceito contra Joana D'arc", formalmente chamado de **SQL Injection**. O que acontece quando o contato a ser adicionado possui no nome uma aspas simples? O código SQL se quebra todo e para de funcionar ou, pior ainda, o usuário final é capaz de alterar seu código sql para executar aquilo que ele deseja (SQL injection)... tudo isso porque escolhemos aquela linha de código e não fizemos o escape de caracteres especiais.

Mais um problema que enxergamos aí é na data. Ela precisa ser passada no formato que o banco de dados entenda e como uma `String`, portanto, se você possui um objeto `java.util.Calendar` que é o nosso caso, você precisará fazer a conversão desse objeto para a `String`.

Por esses três motivos não usaremos código SQL como mostrado anteriormente. Vamos imaginar algo mais genérico e um pouco mais interessante:

```
String sql = "insert into contatos " +  
            "(nome,email,endereco,dataNascimento) " +  
            "values (?, ?, ?, ?)";
```

Existe uma maneira em Java de escrever o código SQL como no primeiro exemplo dessa seção (com concatenações de strings). Essa maneira não será ensinada durante o curso pois é uma péssima prática que dificulta a manutenção do seu projeto.

Perceba que não colocamos os pontos de interrogação de brincadeira, mas sim porque realmente não sabemos o que desejamos inserir. Estamos interessados em executar aquele código mas não sabemos ainda quais são os **parâmetros** que utilizaremos nesse código SQL que será executado, chamado de statement.

As cláusulas são executadas em um banco de dados através da interface PreparedStatement. Para receber um PreparedStatement relativo à conexão, basta chamar o método prepareStatement, passando como argumento o comando SQL com os valores vindos de variáveis preenchidos com uma interrogação.

```
String sql = "insert into contatos " +  
            "(nome,email,endereco,dataNascimento) " +  
            "values (?, ?, ?, ?)";  
PreparedStatement stmt = connection.prepareStatement(sql);
```

Logo em seguida, chamamos o método setString do PreparedStatement para preencher os valores que são do tipo String, passando a posição (começando em 1) da interrogação no SQL e o valor que deve ser colocado:

```
// preenche os valores  
stmt.setString(1, "Caelum");  
stmt.setString(2, "contato@caelum.com.br");  
stmt.setString(3, "R. Vergueiro 3185 cj57");
```

Precisamos definir também a data de nascimento do nosso contato, para isso, precisaremos de um objeto do tipo java.sql.Date para passarmos para o nosso PreparedStatement. Nesse exemplo, vamos passar a data atual. Para isso, vamos passar um long que representa os milissegundos da data atual para dentro de um java.sql.Date que é o tipo suportado pela API JDBC. Vamos utilizar a classe Calendar para conseguirmos esses milissegundos:

```
java.sql.Date dataParaGravar = new java.sql.Date(  
    Calendar.getInstance().getTimeInMillis());
```



```
stmt.setDate(4, dataParaGravar);
```

Por fim, uma chamada a `execute()` executa o comando SQL:

```
stmt.execute();
```

Imagine todo esse processo sendo escrito toda vez que desejar inserir algo no banco? Ainda não consegue visualizar o quão destrutivo isso pode ser?

Veja o exemplo abaixo, que abre uma conexão e insere um contato no banco:

```
public class JDBCInsere {  
  
    public static void main(String[] args) throws SQLException {  
  
        // conectando  
        Connection con = new ConnectionFactory().getConnection();  
  
        // cria um preparedStatement  
        String sql = "insert into contatos" +  
            " (nome,email,endereco,dataNascimento)" +  
            " values (?, ?, ?, ?)";  
        PreparedStatement stmt = con.prepareStatement(sql);  
  
        // preenche os valores  
        stmt.setString(1, "Caelum");  
        stmt.setString(2, "contato@caelum.com.br");  
        stmt.setString(3, "R. Vergueiro 3185 cj57");  
        stmt.setDate(4, new java.sql.Date(  
            Calendar.getInstance().getTimeInMillis()));  
  
        // executa  
        stmt.execute();  
        stmt.close();  
  
        System.out.println("Gravado!");  
  
        con.close();  
    }  
}
```

Para saber mais: Fechando a conexão propriamente

Não é comum utilizar JDBC diretamente hoje em dia. O mais praticado é o uso de alguma API de ORM como a **JPA** ou o **Hibernate**. Tanto na JDBC quanto em bibliotecas ORM deve-se prestar atenção no momento de fechar a conexão.

O exemplo dado acima não a fecha caso algum erro ocorra no momento de inserir um dado no banco de dados. O comum é fechar a conexão em um bloco `finally`:

```

public class JDBCInsere {
    public static void main(String[] args) throws SQLException {
        Connection con = null;
        try {
            con = new ConnectionFactory().getConnection();

            // faz um monte de operações.
            // que podem lançar exceptions runtime e SQLException
        catch (SQLException e) {
            System.out.println(e);
        } finally {
            con.close();
        }
    }
}

```

Dessa forma, mesmo que o código dentro do try lance exception, o `con.close()` será executado. Garantimos que não deixaremos uma conexão pendurada sem uso. Esse código pode ficar muito maior se quisermos ir além. Pois o que acontece no caso de `con.close` lançar uma exception? Quem a tratará?

Trabalhar com recursos caros, como conexões, sessões, threads e arquivos, sempre deve ser muito bem pensado. Deve-se tomar cuidado para não deixar nenhum desses recursos abertos, pois poderá vazar algo precioso da nossa aplicação. Como veremos durante o curso, é importante centralizar esse tipo de código em algum lugar, para não repetir uma tarefa complicada como essa.

Além disso, há a estrutura do Java 7 conhecida como *try-with-resources*. Ela permite declarar e inicializar, dentro do try, objetos que implementam `AutoCloseable`. Dessa forma, ao término do try, o próprio compilador inserirá instruções para invocar o close desses recursos, além de se precaver em relação a exceções que podem surgir por causa dessa invocação. Nosso código ficaria mais reduzido e organizado, além do escopo de `con` só valer dentro do try:

```

try (Connection con = new ConnectionFactory().getConnection()) {
    // faz um monte de operações.
    // que podem lançar exceptions runtime e SQLException
} catch (SQLException e) {
    System.out.println(e);
}

```

Para saber mais: A má prática Statement

Em vez de usar o `PreparedStatement`, você pode usar uma interface mais simples chamada `Statement`, que simplesmente executa uma cláusula SQL no método `execute`:

```

Statement stmt = con.createStatement();
stmt.execute("INSERT INTO ...");
stmt.close();

```

Mas prefira a classe `PreparedStatement` que é mais rápida que `Statement` e deixa seu código muito mais limpo.

Geralmente, seus comandos SQL conterão valores vindos de variáveis do programa Java; usando `Statements`, você terá que fazer muitas concatenações, mas usando `PreparedStatements`, isso fica mais limpo e fácil.

Para saber mais: JodaTime

A API de datas do Java, mesmo considerando algumas melhorias da `Calendar` em relação a `Date`, ainda é muito pobre. Na versão 8 do Java temos uma API nova, a `java.time`, que facilita bastante o trabalho. Ela é baseada na excelente biblioteca de datas chamada `JodaTime`.

2.10 – DAO – DATA ACCESS OBJECT

Já foi possível sentir que colocar código SQL dentro de suas classes de lógica é algo nem um pouco elegante e muito menos viável quando você precisa manter o seu código.

Quantas vezes você não ficou bravo com o programador responsável por aquele código ilegível?

A ideia a seguir é remover o código de acesso ao banco de dados de suas classes de lógica e colocá-lo em uma classe responsável pelo acesso aos dados. Assim o código de acesso ao banco de dados fica em um lugar só, tornando mais fácil a manutenção.

Que tal se pudéssemos chamar um método adiciona que adiciona um Contato ao banco?

Em outras palavras quero que o código a seguir funcione:

```
// adiciona os dados no banco
Misterio bd = new Misterio();
bd.adiciona("meu nome", "meu email", "meu endereço", meuCalendar);
```

Tem algo estranho nesse código. Repare que todos os parâmetros que estamos passando são as informações do contato. Se contato tivesse 20 atributos, passaríamos 20 parâmetros? Java é orientado a Strings? Vamos tentar novamente: em outras palavras quero que o código a seguir funcione:

```
// adiciona um contato no banco
Misterio bd = new Misterio();
```

```
// método muito mais elegante
bd.adiciona(contato);
```

Tentaremos chegar ao código anterior: seria muito melhor e mais elegante poder chamar um único método responsável pela inclusão, certo?

```
public class TestaInsere {

    public static void main(String[] args) {

        // pronto para gravar
        Contato contato = new Contato();
        contato.setNome("Caelum");
        contato.setEmail("contato@caelum.com.br");
        contato.setEndereco("R. Vergueiro 3185 cj87");
        contato.setDataNascimento(Calendar.getInstance());

        // grave nessa conexão!!!
        Misterio bd = new Misterio();

        // método elegante
        bd.adiciona(contato);

        System.out.println("Gravado!");
    }
}
```

O código anterior já mostra o poder que alcançaremos: através de uma única classe seremos capazes de acessar o banco de dados e, mais ainda, somente através dessa classe será possível acessar os dados.

Esta ideia, inocente à primeira vista, é capaz de isolar todo o acesso a banco em classes bem simples, cuja instância é um **objeto** responsável por **acessar os dados**. Da responsabilidade deste objeto surgiu o nome de **Data Access Object** ou simplesmente **DAO**, um dos mais famosos padrões de projeto (*design pattern*).

O que falta para o código acima funcionar é uma classe chamada ContatoDao com um método chamado adiciona. Vamos criar uma que se conecta ao banco ao construirmos uma instância dela:

```
public class ContatoDao {

    // a conexão com o banco de dados
    private Connection connection;

    public ContatoDao() {
        this.connection = new ConnectionFactory().getConnection();
    }
}
```

Agora que todo ContatoDao possui uma conexão com o banco, podemos focar no método adiciona, que recebe um Contato como argumento e é responsável por adicioná-lo através de código SQL:

```
public void adiciona(Contato contato) {
    String sql = "insert into contatos " +
        "(nome,email,endereco,dataNascimento)" +
        " values (?,?,,?)";

    try {
        // prepared statement para inserção
        PreparedStatement stmt = con.prepareStatement(sql);

        // seta os valores

        stmt.setString(1,contato.getNome());
        stmt.setString(2,contato.getEmail());
        stmt.setString(3,contato.getEndereco());
        stmt.setDate(4, new Date(
            contato.getDataNascimento().getTimeInMillis()));

        // executa
        stmt.execute();
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

Encapsulamos a SQLException em uma RuntimeException mantendo a ideia anterior da ConnectionFactory de desacoplar o código de API de JDBC.

2.11 – EXERCÍCIOS: JAVABEANS E CONTATO DAO

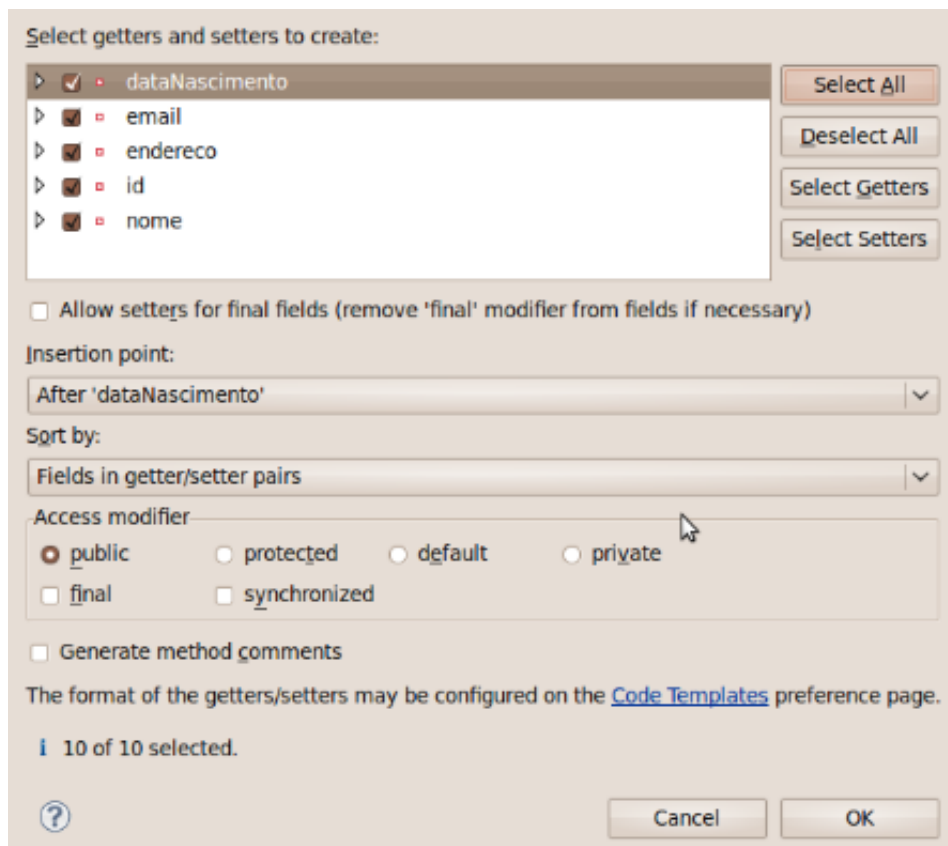
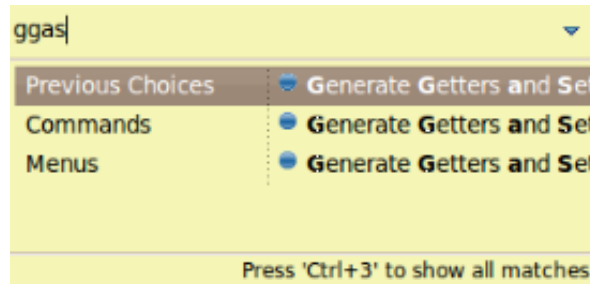
1. Crie a classe de Contato no pacote `br.com.caelum.jdbc.modelo`. Ela será nosso JavaBean para representar a entidade do banco. Deverá ter id, nome, email, endereço e uma data de nascimento.

Coloque os atributos na classe e gere os getters e setters para cada atributo:

```
1 public class Contato {
2     private Long id;
3     private String nome;
4     private String email;
5     private String endereco;
6     private Calendar dataNascimento;
7 }
```

Dica: use o atalho do Eclipse para gerar os getters e setters. Aperte `Ctrl + 3`,

digite ggas que é a abreviação de Generate getters and setters e selecione todos os getters e setters.



2. Vamos desenvolver nossa classe de DAO. Crie a classe ContatoDao no pacote br.com.caelum.jdbc.dao. Seu papel será gerenciar a conexão e inserir Contatos no banco de dados.

Para a conexão, vamos criá-la no construtor e salvar em um atributo:

```
1 public class ContatoDao {  
2  
3     // a conexão com o banco de dados  
4     private Connection connection;  
5  
6     public ContatoDao() {  
7         this.connection = new ConnectionFactory().getConnection();  
8     }  
9  
10 }
```

Use o Eclipse para ajudar com os imports! (atalho *Ctrl + Shift + O*)

O próximo passo é criar o método de adição de contatos. Ele deve receber um objeto do tipo Contato como argumento e encapsular totalmente o trabalho com o banco de dados. Internamente, use um PreparedStatement como vimos na aula para executar o SQL:

```
1 public void adiciona(Contato contato) {
2     String sql = "insert into contatos " +
3         "(nome,email,endereco,dataNascimento)" +
4         " values (?, ?, ?, ?)";
5
6     try {
7         // prepared statement para inserção
8         PreparedStatement stmt = connection.prepareStatement(sql);
9
10        // seta os valores
11        stmt.setString(1, contato.getNome());
12        stmt.setString(2, contato.getEmail());
13        stmt.setString(3, contato.getEndereco());
14        stmt.setDate(4, new Date(
15            contato.getDataNascimento().getTimeInMillis()));
16
17        // executa
18        stmt.execute();
19        stmt.close();
20    } catch (SQLException e) {
21        throw new RuntimeException(e);
22    }
23 }
```

ATENÇÃO: Lembre-se de importar as classes de SQL do pacote `java.sql`, inclusive a classe `Date`!

3. Para testar nosso DAO, desenvolva uma classe de testes com o método `main`. Por exemplo, uma chamada `TestaInsere` no pacote `br.com.caelum.jdbc.teste`. Gere o `main` pelo Eclipse.

O nosso programa de testes deve, dentro do `main`, criar um novo objeto `Contato` com dados de teste e chamar a nova classe `ContatoDao` para adicioná-lo ao banco de dados:

```
1 // pronto para gravar
2 Contato contato = new Contato();
3 contato.setNome("Caelum");
4 contato.setEmail("contato@caelum.com.br");
5 contato.setEndereco("R. Vergueiro 3185 cj57");
6 contato.setDataNascimento(Calendar.getInstance());
7
8 // grave nessa conexão!!!
9 ContatoDao dao = new ContatoDao();
10
11 // método elegante
```

```
12 dao.adiciona(contato);
13
14 System.out.println("Gravado!");
```

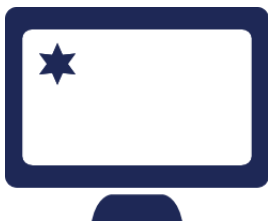
Execute seu programa e veja se tudo correu bem.

4. Verifique se o contato foi adicionado. Abra o terminal e digite:

```
mysql -u root
use fj21;
select * from contatos;
```

exit para sair do console do MySQL.

Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](#)

2.12 – FAZENDO PESQUISAS NO BANCO DE DADOS

Para pesquisar também utilizamos a interface `PreparedStatement` para montar nosso comando SQL. Mas como uma pesquisa possui um retorno (diferente de uma simples inserção), usaremos o método `executeQuery` que retorna todos os registros de uma determinada query.

O objeto retornado é do tipo `ResultSet` do JDBC, o que nos permite navegar por seus registros através do método `next`. Esse método retornará `false` quando chegar ao fim da pesquisa, portanto ele é normalmente utilizado para fazer um laço nos registros:

```
// pega a conexão e o Statement
Connection con = new ConnectionFactory().getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");

// executa um select
ResultSet rs = stmt.executeQuery();
```



```
// itera no ResultSet
while (rs.next()) {
}

rs.close();
stmt.close();
con.close();
```

Para retornar o valor de uma coluna no banco de dados, basta chamar um dos métodos get do ResultSet, dentre os quais, o mais comum: getString.

```
// pega a conexão e o Statement
Connection con = new ConnectionFactory().getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");

// executa um select
ResultSet rs = stmt.executeQuery();

// itera no ResultSet
while (rs.next()) {
    String nome = rs.getString("nome");
    String email = rs.getString("email")

    System.out.println(nome + " :: " + email);
}

stmt.close();
con.close();
```

Recurso Avançado: O cursor

Assim como o cursor do banco de dados, só é possível mover para o próximo registro. Para permitir um processo de leitura para trás é necessário especificar na abertura do ResultSet que tal cursor deve ser utilizado.

Mas, novamente, podemos aplicar as ideias de **DAO** e criar um método getList() no nosso ContatoDao. Mas o que esse método retornaria? Um ResultSet? E teríamos o código de manipulação de ResultSet espalhado por todo o código? Vamos fazer nosso getList() devolver algo mais interessante, uma lista de Contato:

```
PreparedStatement stmt = this.connection
    .prepareStatement("select * from contatos");
ResultSet rs = stmt.executeQuery();

List<Contato> contatos = new ArrayList<Contato>();

while (rs.next()) {
```

```

// criando o objeto Contato
Contato contato = new Contato();
contato.setNome(rs.getString("nome"));
contato.setEmail(rs.getString("email"));
contato.setEndereco(rs.getString("endereco"));

// montando a data através do Calendar
Calendar data = Calendar.getInstance();
data.setTime(rs.getDate("dataNascimento"));
contato.setDataNascimento(data);

// adicionando o objeto à lista
contatos.add(contato);
}

rs.close();
stmt.close();

return contatos;

```

2.13 – EXERCÍCIOS: LISTAGEM

1. Crie o método `getLista` na classe `ContatoDao`. Importe `List` de `java.util`:

```

1 public List<Contato> getLista() {
2     try {
3         List<Contato> contatos = new ArrayList<Contato>();
4         PreparedStatement stmt = this.connection.
5             prepareStatement("select * from contatos");
6         ResultSet rs = stmt.executeQuery();
7
8         while (rs.next()) {
9             // criando o objeto Contato
10            Contato contato = new Contato();
11            contato.setId(rs.getLong("id"));
12            contato.setNome(rs.getString("nome"));
13            contato.setEmail(rs.getString("email"));
14            contato.setEndereco(rs.getString("endereco"));
15
16            // montando a data através do Calendar
17            Calendar data = Calendar.getInstance();
18            data.setTime(rs.getDate("dataNascimento"));
19            contato.setDataNascimento(data);
20
21            // adicionando o objeto à lista
22            contatos.add(contato);
23        }
24        rs.close();
25        stmt.close();
26        return contatos;
27    } catch (SQLException e) {
28        throw new RuntimeException(e);
29    }

```

30 }

2. Vamos usar o método `getLista` para listar todos os contatos do nosso banco de dados.

Crie uma classe chamada `TestaLista` com um método `main`:

a. Crie um `ContatoDao`:

```
ContatoDao dao = new ContatoDao();
```

b. Liste os contatos com o DAO:

```
List<Contato> contatos = dao.getLista();
```

c. Itere nessa lista e imprima as informações dos contatos:

```
for (Contato contato : contatos) {  
    System.out.println("Nome: " + contato.getNome());  
    System.out.println("Email: " + contato.getEmail());  
    System.out.println("Endereço: " + contato.getEndereco());  
    System.out.println("Data de Nascimento: " +  
        contato.getDataNascimento().getTime() + "\n");  
}
```

3. Rode o programa anterior clicando da direita no mesmo, *Run*, *Run as Java Application* (aproveite para aprender a tecla de atalho para executar a aplicação).

2.14 – UM POUCO MAIS...

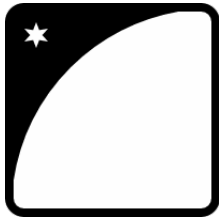
1. Assim como o MySQL existem outros bancos de dados gratuitos e opensource na internet. O HSQLDB é um banco desenvolvido em Java que pode ser acoplado a qualquer aplicação e libera o cliente da necessidade de baixar qualquer banco de dados antes da instalação de um produto Java!

2. O Hibernate tomou conta do mercado e virou febre mundial pois não se faz necessário escrever uma linha de código SQL!

3. Se um projeto não usa nenhuma das tecnologias de ORM (Object Relational Mapping) disponíveis, o mínimo a ser feito é seguir o DAO.

Você não está nessa página a toa

Você chegou aqui porque a Caelum é referência nacional em cursos de Java,



Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

[Consulte as vantagens do curso *Java para Desenvolvimento Web*.](#)

2.15 – EXERCÍCIOS OPCIONAIS

1. A impressão da data de nascimento ficou um pouco estranha. Para formatá-la, pesquise sobre a classe SimpleDateFormat.
2. Crie uma classe chamada DAOException que estenda de RuntimeException e utilize-a no seu ContatoDao.
3. Use cláusulas where para refinar sua pesquisa no banco de dados. Por exemplo:
where nome like 'C%'
4. Crie o método pesquisar que recebe um id (int) e retorna um objeto do tipo Contato.

Desafios

1. Faça conexões para outros tipos de banco de dados disponíveis.

2.16 – OUTROS MÉTODOS PARA O SEU DAO

Agora que você já sabe usar o PreparedStatement para executar qualquer tipo de código SQL e ResultSet para receber os dados retornados da sua pesquisa fica simples, porém maçante, escrever o código de diferentes métodos de uma classe típica de DAO.

Veja primeiro o método altera, que recebe um contato cujos valores devem ser alterados:

```
1 public void altera(Contato contato) {
2     String sql = "update contatos set nome=?, email=?, endereco=?" +
3         "dataNascimento=? where id=?";
4     try {
5         PreparedStatement stmt = connection.prepareStatement(sql);
6         stmt.setString(1, contato.getNome());
7         stmt.setString(2, contato.getEmail());
```

```

8      stmt.setString(3, contato.getEndereco());
9      stmt.setDate(4, new Date(contato.getDataNascimento()
10         .getTimeInMillis()));
11      stmt.setLong(5, contato.getId());
12      stmt.execute();
13      stmt.close();
14  } catch (SQLException e) {
15      throw new RuntimeException(e);
16  }
17 }

```

Não existe nada de novo nas linhas acima. Uma execução de query! Simples, não?

O código para remoção: começa com uma query baseada em um contato, mas usa somente o id dele para executar a query do tipo delete:

```

1  public void remove(Contato contato) {
2      try {
3          PreparedStatement stmt = connection.prepareStatement("delete" +
4              "from contatos where id=?");
5          stmt.setLong(1, contato.getId());
6          stmt.execute();
7          stmt.close();
8      } catch (SQLException e) {
9          throw new RuntimeException(e);
10     }
11 }

```

2.17 – EXERCÍCIOS OPCIONAIS – ALTERAR E REMOVER

1. Adicione o método para alterar contato no seu ContatoDao.

```

1  public void altera(Contato contato) {
2      String sql = "update contatos set nome=?, email=?, "+
3          "endereco=?, dataNascimento=? where id=?";
4
5      try {
6          PreparedStatement stmt = connection
7              .prepareStatement(sql);
8          stmt.setString(1, contato.getNome());
9          stmt.setString(2, contato.getEmail());
10         stmt.setString(3, contato.getEndereco());
11         stmt.setDate(4, new Date(contato.getDataNascimento()
12             .getTimeInMillis()));
13         stmt.setLong(5, contato.getId());
14         stmt.execute();
15         stmt.close();
16     } catch (SQLException e) {
17         throw new RuntimeException(e);
18     }
19 }

```

2. Adicione o método para remover contato no seu ContatoDao

```
1 public void remove(Contato contato) {
2     try {
3         PreparedStatement stmt = connection
4             .prepareStatement("delete from contatos where id=?");
5         stmt.setLong(1, contato.getId());
6         stmt.execute();
7         stmt.close();
8     } catch (SQLException e) {
9         throw new RuntimeException(e);
10    }
11 }
```

3. Use os métodos criados anteriormente para fazer testes com o seu banco de dados: atualize e remova um contato.

4. Crie uma classe chamada Funcionario com os campos id (Long), nome, usuario e senha (String).

5. Crie uma tabela no banco de dados chamada funcionarios.

6. Crie uma classe DAO para Funcionario.

7. Use-a para instanciar novos funcionários e colocá-los no seu banco.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](#)

CAPÍTULO ANTERIOR:

[Enfrentando o Java na Web](#)

PRÓXIMO CAPÍTULO:

[O que é Java EE?](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter

