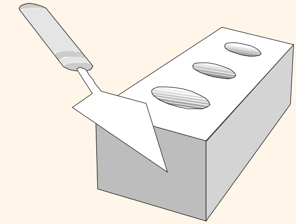
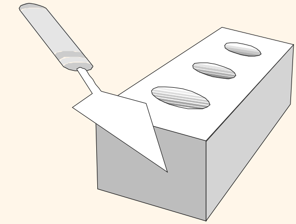


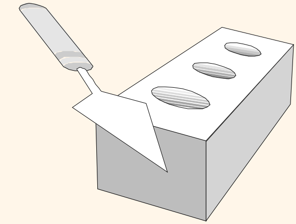
Desenvolvendo Aplicações de Banco de Dados

Capítulo 6



Índice

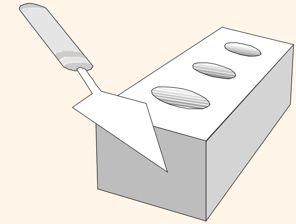
- Banco de dados e aplicações
- SQL embutido
- JDBC
- SQLJ
- Hibernate
- Stored procedures



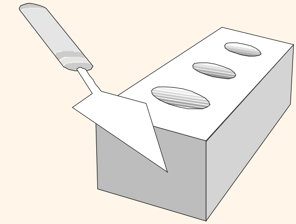
Banco de Dados e Aplicações

- Banco de dados fazem uma coisa muito bem: armazenar os dados para acesso eficiente.
- Nem todos os aspectos de uma aplicação são modeláveis.
  - Comportamento dinâmico.
  - Como garantir que uma corrida vai sair da fila e virar uma corrida efetivada?

Banco de Dados e Aplicações

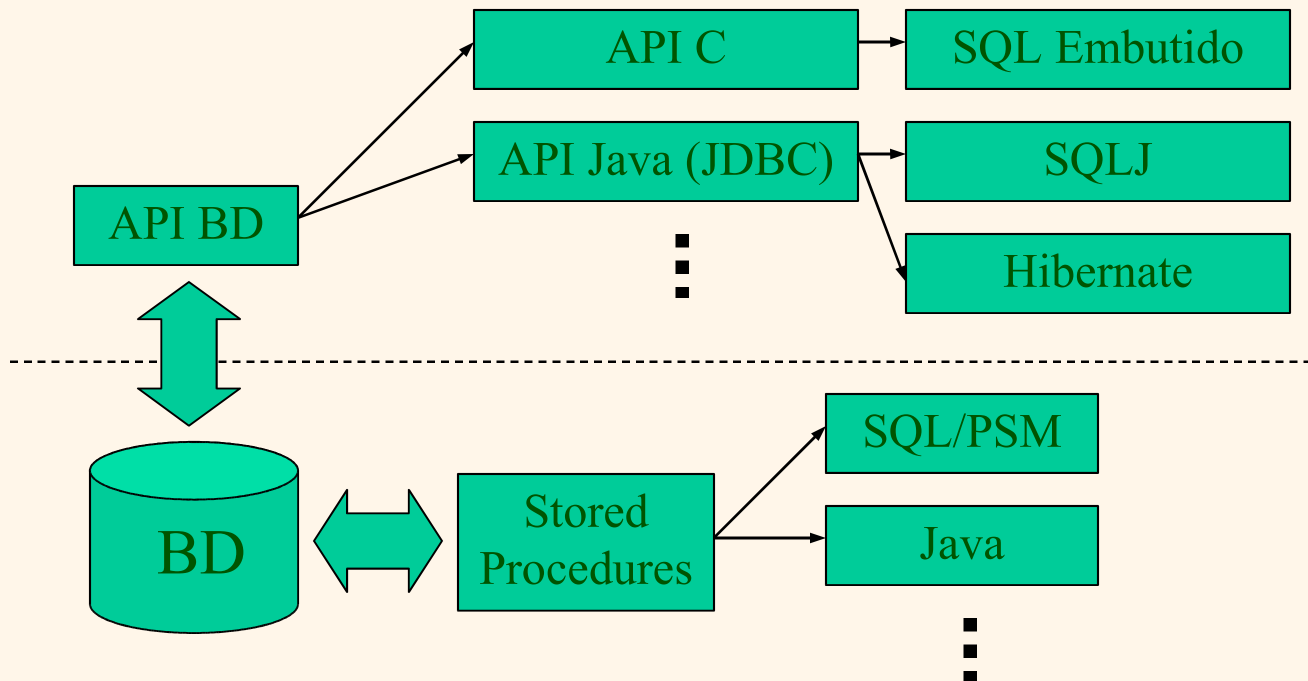


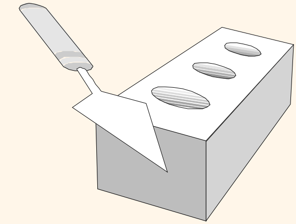
- Não é possível separar dados de algoritmos!
  - Jim Gray: *"We [the data] were separated from our procedural twin at birth"*.
- Temos que fazer a ponte entre o banco de dados e a aplicação.
- Um problema: diferença de impedância.
  - Procedural: Os bancos de dados trabalham apenas com conjuntos de tuplas.
  - OO: O modelo relacional não suporta diretamente noções OO como herança, polimorfismo e agregação.



Modelos de Aplicação

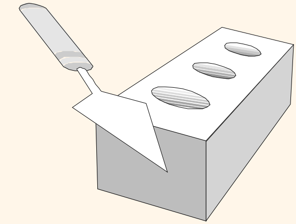
- Onde colocar a lógica procedural?
  - Fora do SGBD (API)
  - Dentro do SGBD (Stored Procedures, BDOO)





Tecnologias

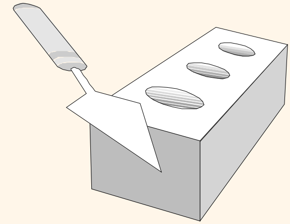
- SQL Embutido (API C, pré-processador)
- JDBC (API Java)
- SQLJ (API Java, pré-processador)
- Hibernate (API Java, mapeamento OR)
- SQL/PSM e Java (stored procedures)



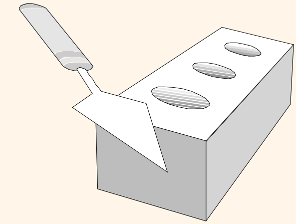
SQL Embutido

- SQL pode ser chamado em um programa C/C++.
  - Um pré-processador converte os comandos SQL em chamadas especiais de API.
  - Depois o compilador C é usado para compilar o código.
  - As declarações SQL podem se referir às variáveis locais (incluindo variáveis especiais usadas para retorno de estado).
  - Deve existir uma declaração para conectar ao banco de dados correto.

SQL Embutido



- Construções da linguagem:
  - Conectando ao banco de dados:
    EXEC SQL CONNECT
  - Declarando variáveis:
    EXEC SQL BEGIN (END) DECLARE SECTION
  - Executando comandos do SQL:
    EXEC SQL <statement>;

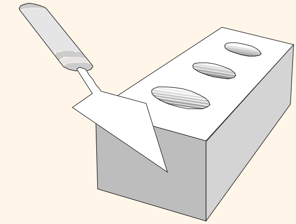


SQL Embutido: Variáveis

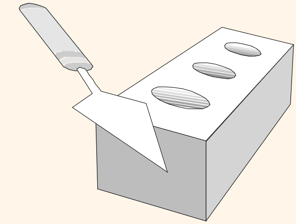
```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];   //CHARACTER(20)
long c_sid;         //INTEGER
short c_rating;     //SMALLINT
float c_age;        //REAL
EXEC SQL END DECLARE SECTION
```

- Duas variáveis de "erro" especiais:
  - SQLCODE (long, é negativa se um erro ocorreu)
  - SQLSTATE (char[6], valores (códigos) predefinidos para erros comuns)

Cursores



- Podemos declarar um cursor em uma relação ou uma consulta (a qual gera uma relação).
- Podemos *abrir* um cursor, e repetidamente *acessar* uma tupla e *mover* o cursor, até que todas as tuplas sejam acessadas (recuperadas).
  - Podemos usar uma cláusula especial, chamada ORDER BY, nas consultas que serão acessadas através de um cursor, para controlar a ordem na qual as tuplas serão retornadas.
    - Os campos em ORDER BY também devem estar na cláusula SELECT.
  - A cláusula ORDER BY, que ordena as tuplas da resposta, é *somente* permitida no contexto de um cursor.
- Podemos também modificar / excluir uma tupla apontada por um cursor.



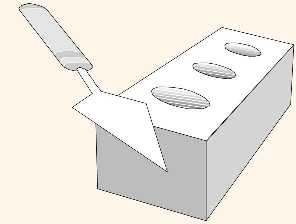
Cursores: Um Exemplo

```
EXEC SQL DECLARE sinfo CURSOR FOR
        SELECT  S.sname
        FROM  Sailors S, Boats B, Reserves R
        WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color=‘red’
        ORDER BY  S.sname
```

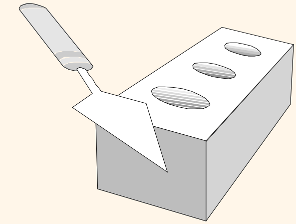
- Retorna nomes dos marinheiros que reservaram um barco vermelho, em ordem alfabética
- Note que é ilegal substituir S.sname por, por exemplo, S.sid na cláusula ORDER BY!
- Podemos usar o comando FETCH para ler os valores do conjunto:

```
FETCH sinfo INTO :sname
```

SQL Embutido: Um Exemplo



```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT S.sname, S.age      FROM Sailors S
    WHERE S.rating > :c_minrating
    ORDER BY S.sname;
do {
    EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
    printf(“%s is %d years old\n”, c_sname, c_age); //imprimindo
} while (SQLSTATE != ‘02000’);
EXEC SQL CLOSE sinfo;
```

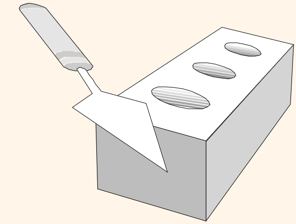


SQL Dinâmico

- Consultas SQL nem sempre são conhecidas em tempo de compilação (e.g., planilha, gráficos que precisam acessar o SGBD):
- Permite o acesso a API do banco de dados.

→Exemplo:

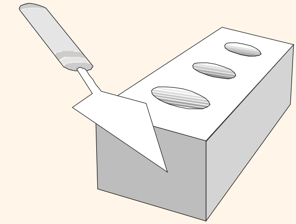
```
char c_sqlstring[ ]={“DELETE FROM Sailors WHERE raiting>5”};

EXEC SQL PREPARE readytogo FROM :c_sqlstring;

EXEC SQL EXECUTE readytogo;
```



APIs Para Banco de Dados

Ao invés de modificar o compilador, adicionar uma biblioteca com as chamadas ao banco de dados (API)

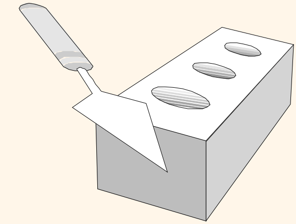
- Interface padronizada especial: procedimentos/objetos
- Passar strings SQL da linguagem de programação, apresentar os resultados de uma forma mais amigável
- *JDBC* da Sun: Java API
- Considera o SGBD neutro
  - um "driver" *captura* as chamadas e as traduz no código do SGBD específico
  - o banco de dados pode estar em uma rede



JDBC: Arquitetura

- Quatro componentes da arquitetura:
  - Aplicações (inicia e termina as conexões, submete as declarações SQL)
  - *"Driver Manager"- (Gerenciador de Driver)* (carrega o driver JDBC)
  - Driver (conecta a fonte de dados, transmite requisições e retorna /traduz os resultados e os códigos de erro)
  - *"DataSource" - (fonte de dados)* (processa as declarações SQL e retorna os resultados)

JDBC: Arquitetura



Quatro tipos de drivers:

<u>Bridge:</u>

- Traduz as funções JDBC chamadas por uma API não nativa. Exemplo: JDBC-ODBC bridge. Pode alterar a performance.

<u>Tradutor direto para a API nativa, via "*non-Java-Driver*":</u>

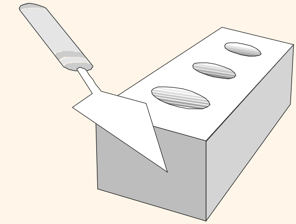
- Acessa diretamente o *DataSource*.
- O *driver* deve estar instalado em todas as estações que rodam a aplicação.

<u>Network bridge:</u>

- Envia comandos pela rede para um servidor, que traduz as requisições do JDBC para métodos específicos do SGBD. Precisa apenas de um pequeno driver JDBC em cada cliente.

<u>Tradutor direto para a API nativa via *driver* Java:</u>

- Converte as chamadas JDBC para acesso direto ao *DataSource,* sem camadas intermediárias.
- Precisa do driver Java específico do SGBD para cada cliente.



JDBC: Classes e Interfaces

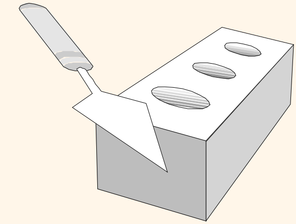
Os passos para enviar uma consulta ao banco de dados:

i) Carregar o driver JDBC

ii) Conectar à fonte de dados

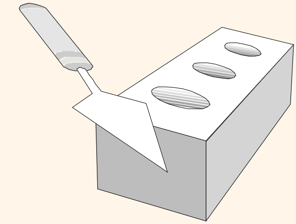
Conexão ao *datasource, autenticação e alocação de recursos.*

iii) Executar o SQL



Gerenciamento do Driver JDBC

- Todos os drivers são gerenciados pela classe DriverManager
- Carregando o driver JDBC:
  - No código Java:
    Class.forName("oracle.jdbc.driver.Oracledriver");
  - Quando iniciamos a aplicação Java (na linha de comando):
    -Djdbc.drivers=oracle.jdbc.driver.Oracledriver



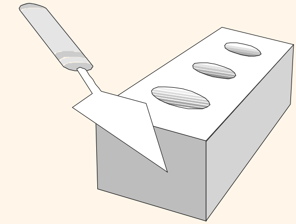
Conexões no JDBC

Interagimos com uma fonte de dados*(datasource)* através de sessões. Cada conexão identifica uma sessão lógica.

- JDBC URL:
  jdbc:<driver>:<otherParameters>

<u>Exemplo:</u>

```
String url="jdbc:oracle:www.bookstore.com:3083";
Connection con;
try {
    con = DriverManager.getConnection(url,usedId,password);
} catch (SQLException excpt) { …}
```



Interface da Classe Connection

- public int getTransactionIsolation() e
  void setTransactionIsolation(int level)

  Define um nível de isolamento para a conexão corrente.

- void setReadOnly(boolean b) e

  void setReadOnly(boolean b)

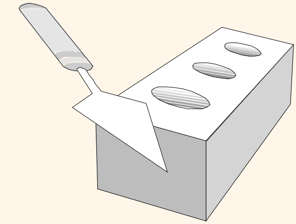
  Especifica se as transações na conexão são somente leitura

- public boolean getAutoCommit() and
  void setAutoCommit(boolean b)

  Se *autocommit* está ativo, então cada declaração SQL é considerada uma transação individual. Do contrário, uma transação é salva usando commit(), ou abortada usando rollback().

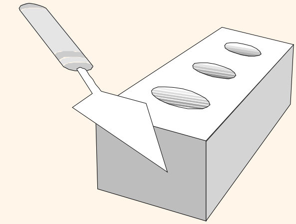
- public boolean isClosed()

  Verifica se a conexão ainda está aberta.



Executando Declarações SQL

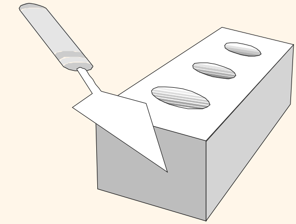
- Três formas diferentes de executar declarações SQL:
  - Statement (declarações SQL executadas uma vez)
  - PreparedStatement (declarações SQL compiladas no servidor)
  - CallableStatment (chamadas de stored procedures)



Executando Declarações SQL

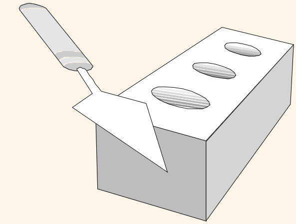
```
String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatment pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1, sid);
pstmt.setString(2, sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4, age);

// sabemos que nenhuma linha é retornada, então
    usamos executeUpdate()
int numRows = pstmt.executeUpdate();
```

ResultSets



- PreparedStatement.executeUpdate somente retorna o número de registros afetados
- PreparedStatement.executeQuery retornam dados, encapsulados em um objeto ResultSet (um cursor)

```
ResultSet rs=pstmt.executeQuery(sql);
// rs é um novo cursor
While (rs.next()) {
  // processa os dados
}
```

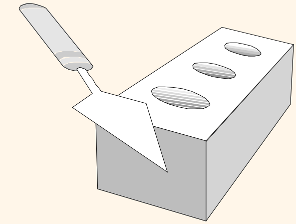


ResultSets

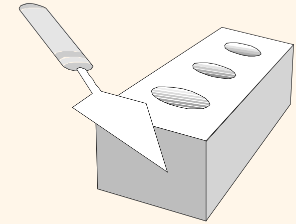
Um ResultSet é um cursor muito poderoso:

- previous(): move uma linha para trás
- absolute(int num): move para a linha com o número especificado
- relative (int num): move para frente ou para trás a quantidade de linhas especificada
- first() and last(): primeira e última linhas, respectivamente

Tipos de Dados em Java e SQL

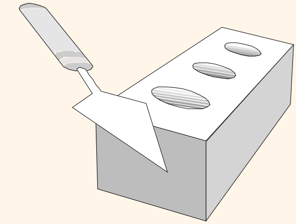


SQL Type	Java class	ResultSet get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.TimeStamp	getTimestamp()



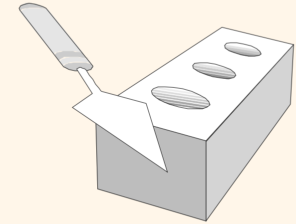
JDBC: Exceções e Alertas

- Chamadas de java.sql podem levantar uma *SQLException* se um erro ocorrer.
- *SQLWarning* é uma subclasse de SQLException; (não são levantadas e sua existência tem que ser explicitamente testada)



JDBC: Exceções e Alertas

```
try {
   stmt=con.createStatement();
   warning=con.getWarnings();
   while(warning != null) {
      // tratando SQLWarnings;
      warning = warning.getNextWarning():
   }
   con.clearWarnings();
   stmt.executeUpdate(queryString);
   warning = con.getWarnings();
   …
} //end try
catch( SQLException SQLe) {
   // tratando a exceção
}
```



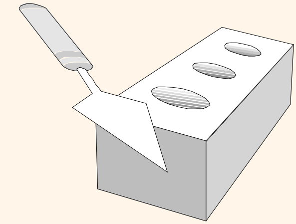
Metadados do Banco de Dados

O objeto DatabaseMetaData fornece informações sobre o Sistema de Banco de Dados.

Ex:

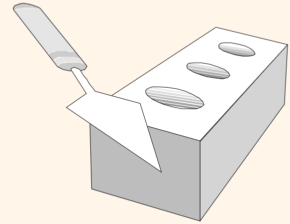
```
DatabaseMetaData md = con.getMetaData();
// imprime informações sobre o driver:
System.out.println(
   “Name:” + md.getDriverName() +
   “version: ” + md.getDriverVersion());
```

Metadados do Banco de Dados



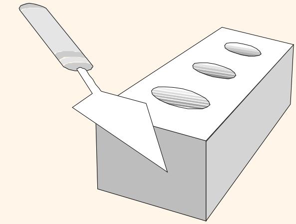
```
DatabaseMetaData md=con.getMetaData();
ResultSet trs=md.getTables(null,null,null,null);
String tableName;
While(trs.next()) {
    tableName = trs.getString(“TABLE_NAME”);
    System.out.println(“Table: “ + tableName);
    // imprime todos os atributos
    ResultSet crs = md.getColumns(null,null,tableName, null);
    while (crs.next()) {
        System.out.println(crs.getString(“COLUMN_NAME” + “, “);
    }
}
```

JDBC: Um Exemplo



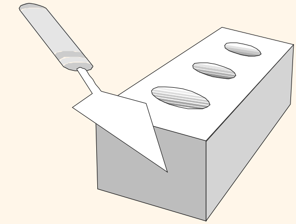
```
Connection con = // conecta à base de dados
  DriverManager.getConnection(url, ”login", ”pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try { // trata exceções
    // loop para imprimir as tuplas
    while (rs.next()) {
        String s = rs.getString(“name");
        Int n = rs.getFloat(“rating");
        System.out.println(s + "   " + n);
    }
} catch(SQLException ex) {
    System.out.println(ex.getMessage ()
        + ex.getSQLState () + ex.getErrorCode ());
}
```

SQLJ



Complementa o JDBC com um modelo estático de consulta: O compilador pode realizar verificações de sintaxe, verificações de tipos fortes, consistência da consulta com o esquema

- Todos os argumentos sempre pertencem a mesma variável:
```
#sql var= {
   SELECT name, rating INTO :name, :rating
   FROM Books WHERE sid = :sid };
```
- Compare ao JDBC:
```
PreparedStatement stmt = connection.prepareStatement
("SELECT name, rating  FROM Books WHERE sid = ?");
stmt.setInt(1, sid);
ResultSet var = stmt.executeQuery();
```



SQLJ: Uma Visão Geral

Programa .sqlj

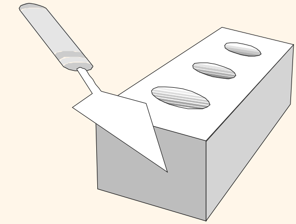
BD

SQLJ Translator

Programa .java

Compilador Java

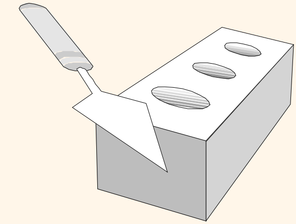
Programa .class



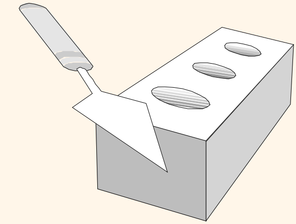
SQLJ: Uma Visão Geral

- Vantagens
  - Detalhes são escondidos; mais legível.
  - Verificação sintática do SQL em tempo de compilação.
- Desvantagens
  - Exige um passo extra de pré-compilação.
  - O programa resultante não é Java; quebra ferramentas.
  - Mensagens de erro que não refletem o código criado.

Código SQLJ



```
Int sid; String name; Int rating;
#sql iterator Sailors(Int sid, String name);
Sailors sailors;

#sql sailors = {
    SELECT sid, name
   FROM Sailors WHERE rating = :rating
};

// exibe os resultados
while (sailors.next()) {
    System.out.println(sailors.sid + sailors.name));
}
sailors.close();
```

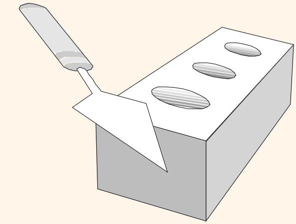


SQLJ Iterators

Dois tipos de iterators ("cursores"):

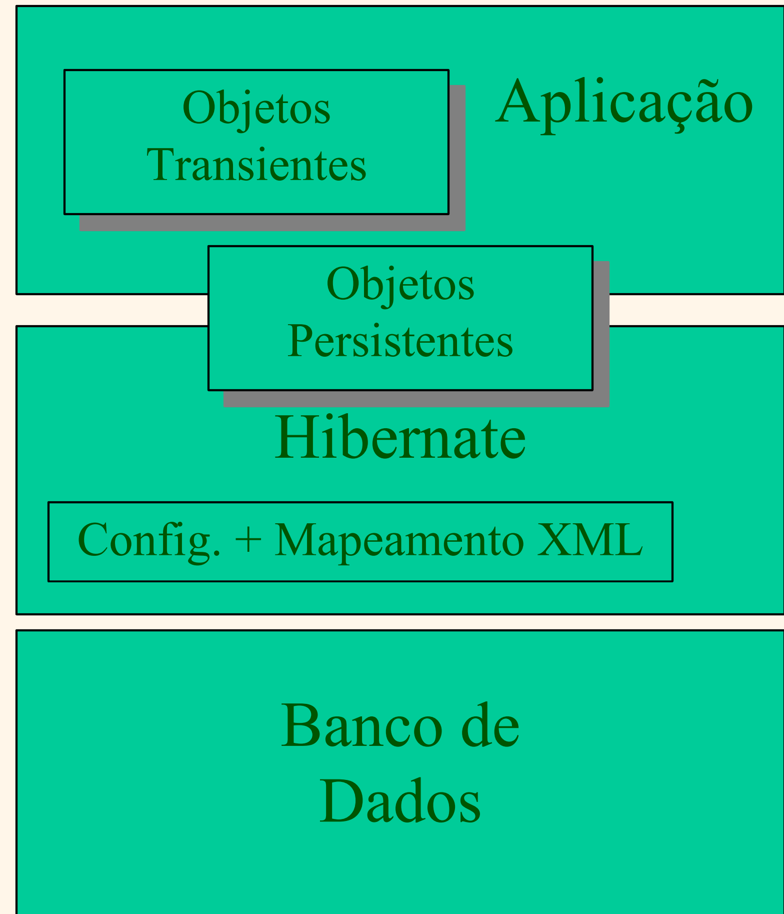
- Named iterator
  - Precisa de um tipo e um nome de variável, e então permite recuperar as colunas pelo nome.
  - Exemplo no slide anterior.
- Positional iterator
  - Precisa somente de um tipo de variável, e usa então a construção FETCH .. INTO:

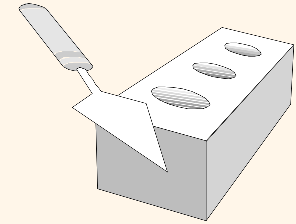
```
#sql iterator Sailors(Int, String);
Sailors sailors;
#sql sailors = …
while (true) {
    #sql {FETCH :sailors INTO :sid, :name} ;
    if (sailors.endFetch()) { break; }
    // processa as informações do navegador
}
```



Hibernate: Arquitetura

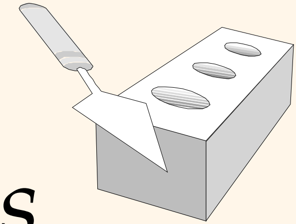
- O diagrama mostra Hibernate usando o banco de dados e configuração de dados para fornecer serviços e objetos persistentes para a aplicação.





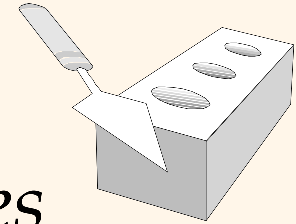
Hibernate: Mapeamento

- Classes Persistentes: São classes compostas de métodos get() and set() para todos os atributos de uma tabela de uma determinada relação.
- Mapeamento de um objeto em relacional que pode ser definido em um documento XML. Este documento é necessário para leitura e escrita na tabela e representação das relações entre esta e outras tabelas.



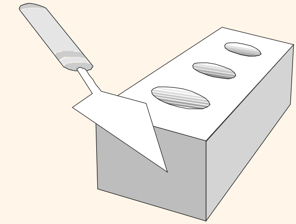
Hibernate: Manipulação de Dados

- Classes e métodos para manipulação dos dados que compõem as tabelas dos relacionamentos.
- Geralmente chamados classes DAO.
- Contém lógica que permite a comunicação com o servidor a partir de uma transação particular onde dados podem ser inseridos, atualizados, apagados, recuperados entre outras ações.



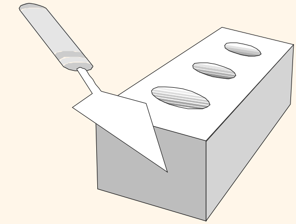
Hibernate: Exemplo de Classes Persistentes

```
public class Blog
{
      private Long id;  private String name;

      public Long getId() {  return id;  }
      public void setId(Long id)  {  this. id = id;  }
      public String getName()  {  return name;  }
      public void setName(String name)  {
            this.name = name;
      }
}
```



Hibernate: Exemplo de Mapeamento

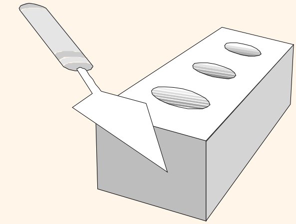
```
<hibernate-mapping>
  <class  name = “blog”  table = “BLOGS”>
    <id  name = “id”    column = “BLOG_ID”>
      <generator class = “native”/>
    </id>
    <property  name = “name” column=“Name”
                    not-null=“true” unique=“true”/>
    <bag name=“items” lazy=“true”
           order-by=“DATE_TIME” cascade=“all”>
      <key column=“BLOG_ID”/>
     <one-to-many   class=“blogItem”>
     </bag>
  </class>
</hibernate-mapping>
```



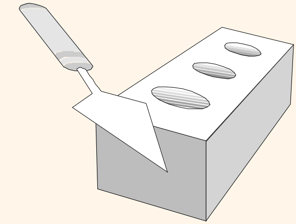
Hibernate: Exemplo de Código

```
public class BlogMain {
    public Blog createBlog(String name) throws HibernateException {
    Blog  blog = new Blog();
    blog.setName = name;
    Session session = _sessions.openSession();
    Transaction tx = null
    try {
            tx = session.beginTransaction;  session.save(Blog);  tx.commit();
    } catch(HibernateException he) {
            throw he;
    } finally {
            session.close();
    }
    return Blog
}
```

Stored Procedures

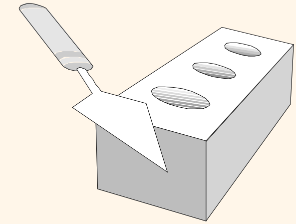


- O que é um stored procedure:
  - Um programa executado através de uma simples declaração SQL
  - É executado no espaço de processo do servidor
- Vantagens:
  - Pode encapsular a lógica da aplicação enquanto permanece “fechado” para os dados
  - Reuso da lógica da aplicação por diferentes usuários
  - Evita o retorno de tuplas uma a uma através de cursores



Por Que Utilizar Procedures?

- Eles permitem que o servidor efetue operações complexas em seus bancos de dados sem envolver o software cliente.
- Eles podem ser compartilhados por todas as aplicações cliente que acessam o banco de dados. Não é necessário programar a mesma lógica dentro de cada aplicação; em vez disso você só programa e testa ela uma vez no servidor.
- Eles reduzem tráfico na rede.
- Stored procedures permitem que você divida tarefas complexas em módulos menores e mais lógicos.
- Stored procedures são bastante úteis para efetuar tarefas de processamento periódico.
- Procedures fornecem melhor concorrência entre o cliente e o servidor.



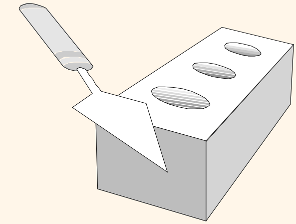
Stored Procedures: Exemplos

```
CREATE PROCEDURE ShowNumReservations
   SELECT S.sid, S.sname, COUNT(*)
   FROM Sailors S, Reserves R
   WHERE S.sid = R.sid
   GROUP BY S.sid, S.sname
```

Stored procedures podem ter parâmetros:

- Três diferentes modos: IN, OUT, INOUT

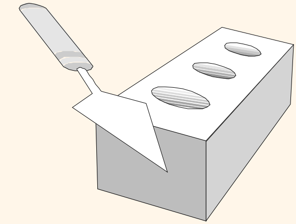
```
CREATE PROCEDURE IncreaseRating(
   IN sailor_sid INTEGER, IN increase INTEGER)
UPDATE Sailors
   SET rating = rating + increase
   WHERE sid = sailor_sid
```



Stored Procedures: Exemplos

Stored procedures não precisam ser escritos em SQL:

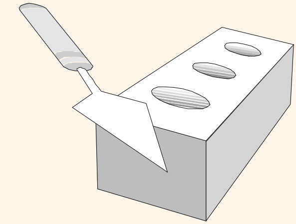
```
CREATE PROCEDURE TopSailors(
   IN num INTEGER)
LANGUAGE JAVA
EXTERNAL NAME “file:///c:/storedProcs/rank.jar”
```



Chamando Stored Procedures

- Em SQL embutido:

```
EXEC SQL BEGIN DECLARE SECTION
Int sid;
Int rating;
EXEC SQL END DECLARE SECTION

// atribuindo a sid e rating os valores:

EXEC CALL IncreaseRating(:sid,:rating);
```



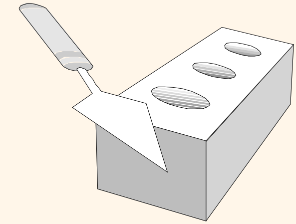
Chamando Stored Procedures

JDBC:

```
CallableStatement cstmt=
   con.prepareCall("{call ShowSailors}");
ResultSet rs = cstmt.executeQuery();
while (rs.next()) {   …}
```

SQLJ:

```
#sql iterator ShowSailors(…);
ShowSailors showsailors;
#sql showsailors={CALL ShowSailors};
while (showsailors.next()) {    …}
```



SQL/PSM

A maioria dos SGBDs permitem aos usuários escrever stored procedures em uma linguagem simples de uso geral (parecida com o SQL)

→ O padrão SQL/PSM é um representante

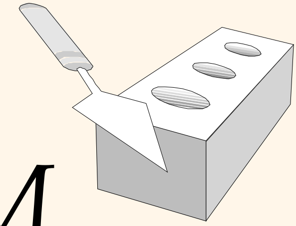
Declarando um procedure:

```
CREATE PROCEDURE name(p1, p2, …, pn)
//declarações de variáveis locais
//código do procedimento;
```

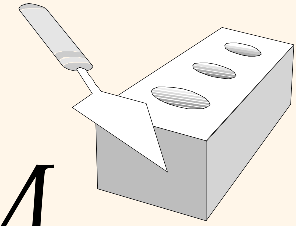
Declarando uma função:

```
CREATE FUNCTION name (p1, …, pn) RETURNS sqlDType
//declarações de variáveis locais
//código da função;
```

Principais Construções SQL/PSM



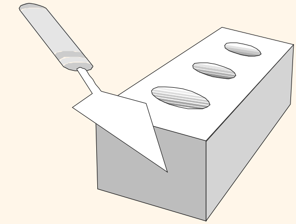
```
CREATE FUNCTION rate Sailor
       (IN sailorId INTEGER)
       RETURNS INTEGER
DECLARE rating INTEGER
DECLARE numRes INTEGER
SET numRes = (SELECT COUNT(*)
                         FROM Reserves R
                        WHERE R.sid = sailorId)
IF (numRes > 10) THEN rating =1;
ELSE rating = 0;
END IF;
RETURN rating;
```



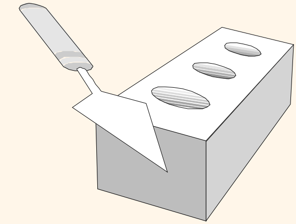
Principais Construções SQL/PSM

- Variáveis locais (DECLARE)
- Retorno (RETURN) de valores de função (FUNCTION)
- Atribui variáveis com SET
- Laços e controles de fluxo:
  - IF (condição) THEN declarações;
    ELSEIF (condição) declarações;
    … ELSE declarações; END IF;
  - LOOP declarações; END LOOP
- Consultas podem fazer parte das expressões
- Podem usar cursores naturalmente sem "EXEC SQL"

Resumo



- Não é possível separar dados de algoritmos!
- Podemos colocar a lógica procedural:
  - Fora do SGBD (API)
  - Dentro do SGBD
- APIs fazem a ponte entre a aplicação e o SGBD.
- SQL Embutido: Uma forma de minimizar a complexidade da API. Usa um pré-processador.
- O mecanismo de cursor permite operar em registros individuais.



Resumo

- SQLJ: SQL Embutido para Java.
- Hibernate: Uma forma OO de esconder a complexidade da API.
- Stored procedures executam a lógica da aplicação diretamente no servidor. Podem ser escritos em várias linguagens (SQL/PSM, Java e outras).