

# *Conectividad a Bases de Datos Relacionales*

## *Programación Web-Dinámico*



# Introducción

---

- Bases de Datos Relacionales
- API de JDBC para acceso a bases de datos
- Cómo establecer una conexión
- Cómo crear un objeto Statement
- Cómo ejecutar un query (consulta)
- Cómo procesar un objeto ResultSet
- Creación de clases de acceso a datos
- Ejemplo de una tienda virtual con acceso a BD
- Ejecución de sentencias SQL con JDBC
- Transacciones
- Transacciones en JDBC

# Bases de Datos Relacionales

---

- La mayoría de los sistemas de bases de datos existentes son relacionales!
- Fuerte teoría detrás de las bases de datos relacionales:
  - Normalización (formas normales)
  - Reglas de integridad de datos
  - Llaves primarias, foráneas, índices
  - Guardan información consistente
  - Transacciones (Begin Transaction, Commit/Rollback)
  - Etc.
- Herramientas gráficas de modelado ampliamente usadas
  - Un diseño de una base de datos relacional puede ser fácilmente comunicado a través de un diagrama ER

# Bases de Datos Relacionales II

---

- Existe una gran cantidad de implementaciones de bases de datos relacionales de alta calidad. Han evolucionado por más de treinta años.
- Tienen modelos matemáticos bien definidos para la manipulación de datos, tales como:
  - Álgebra relacional
  - Cálculo relacional
- Son poderosas y ¿¿estándar??
- SQL (Structured Query Language), permite la creación de tablas, inserción de registros en una tabla, actualización y borrado de registros, consultas (queries), etc.

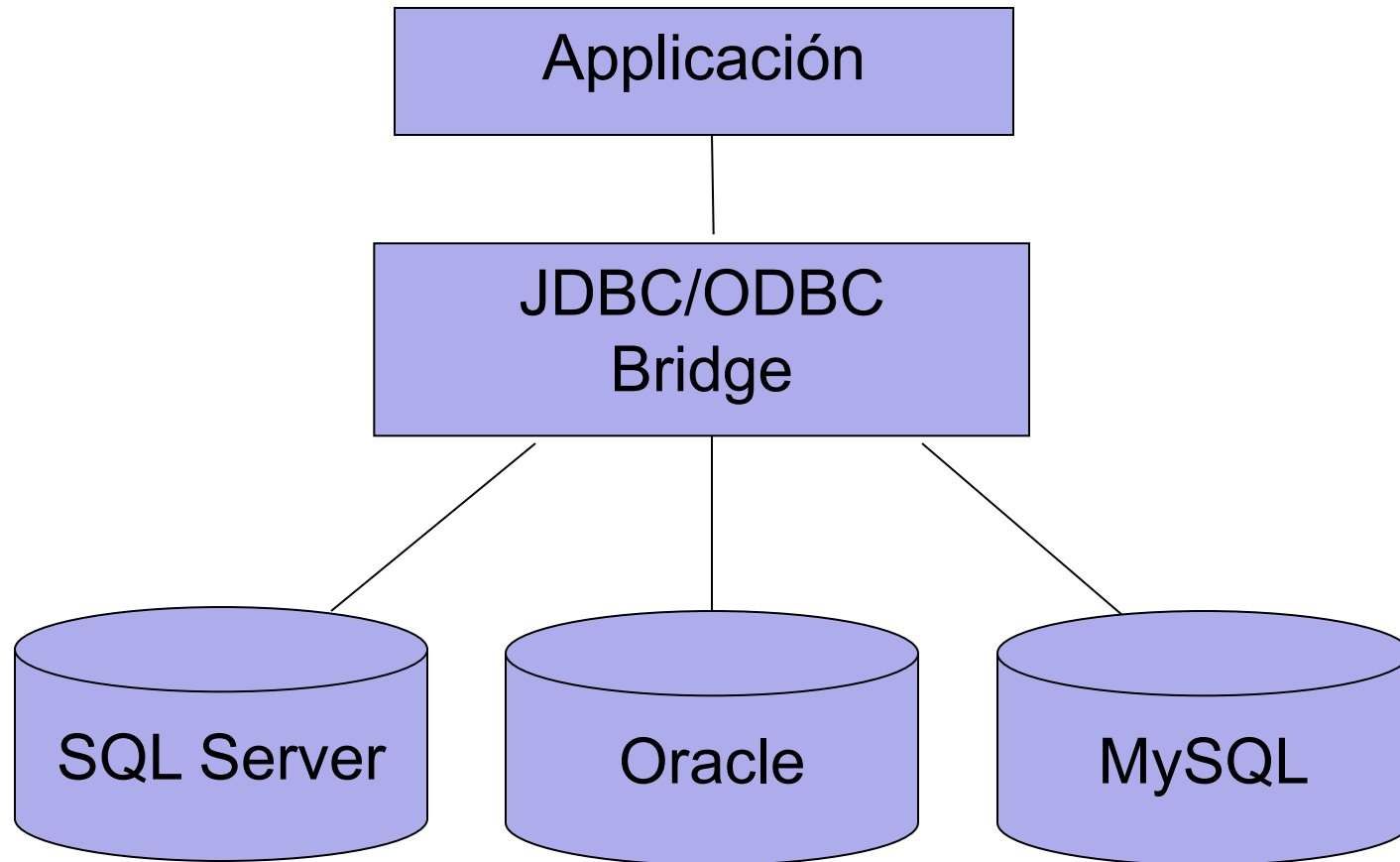
# Bases de Datos Relacionales III

---

- Algunas implementaciones difieren en los detalles, y además hay diferentes versiones de SQL.
- Cada implementador de un RDBMS incorpora características únicas en su producto (Transact-SQL de SQL Server, PL/SQL de Oracle, etc.).
- Los sistemas de bases de datos relacionales permiten resolver problemas complejos de procesamiento de datos, de una manera relativamente simple, y con soluciones elegantes.

# JDBC Bridge

---



# JDBC

---

- Proporciona una forma estándar de acceso a bases de datos relacionales. La API JDBC se encuentra en el paquete `java.sql`.
- El código utilizado en JDBC “debería” funcionar para cualquier base de datos relacional.
- Puede ser que sea necesario utilizar un driver ODBC (en este caso se tendría que utilizar un JDBC:ODBC bridge).
- Si existe el driver nativo Java, solamente es necesario utilizar el driver JDBC.

# Precaución con los drivers

---

- Diferentes drivers soportan diferentes métodos de acceso a la misma base de datos.
- Algunos drivers sólo son capaces de conectarse a una base de datos que reside en el mismo sistema de archivos (misma computadora).
- Otros drivers pueden conectarse a una base de datos utilizando una URL, y potencialmente funcionar sobre Internet (aunque habrá que considerar los firewalls).
- Diferentes bases de datos pueden soportar diferentes operaciones. Algunos drivers para bases de datos no funcionan correctamente!



# Consultas y actualizaciones utilizando la API JDBC

---

- Los siguientes pasos son necesarios para realizar consultas o actualizaciones en una base de datos:
  - Cargar el driver
  - Conexión a la base de datos
  - Crear un objeto `Statement`
  - Ejecutar el `Statement`
  - Procesar el objeto `ResultSet`.

# Cargar el driver

---

- `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`  
Esto especifica el driver que será cargado.
- En los ejercicios en clase usaremos un driver diferente para conectarnos una base de datos MySQL.
- El driver especificado puede ser usado en llamadas subsecuentes a:  
`DriverManager.getConnection();`

# Conexión a la base de datos

---

- Para establecer una conexión necesitamos la clase `DriverManager`, como se muestra a continuación:

```
Connection con = DriverManager.getConnection(
    "jdbc:odbc:shopdb", "user", "password" );
```

- Se especifica la URL de la base de datos, el nombre de usuario, y el password.
- La URL en este ejemplo apunta a la computadora local.
- También es posible tener URLs con otros drivers.

# Creación y ejecución de un objeto Statement

- Un objeto `Statement` es creado a continuación:.

```
Statement statement = con.createStatement();
```

- Un objeto `Statement` es utilizado para ejecutar llamadas SQL sobre la base de datos.

- Algunos métodos comunes para ejecutar consultas y actualizaciones son los siguientes:

```
ResultSet rs = statement.executeQuery(String query);
```

**Regresa un `ResultSet` con el conjunto de registros que son el resultado de la consulta.**

```
int nRows = statement.executeUpdate(String update);
```

**Ejecuta el SQL especificado por el parámetro `update`.**

# Procesamiento del objeto ResultSet

- Se itera sobre el objeto `ResultSet`, para extraer los campos de cada tupla.

```
ResultSet rs = stmt.executeQuery(select);
while(rs.next()) {
    System.out.println
        (rs.getString("Description"));
}
```

En este ejemplo invocamos al método `getString`. También se pueden obtener otros tipos de datos, no sólo `String` (consulta el API JDBC).

# Metadatos del ResultSet

---

- Metadatos – datos acerca de los datos.
- Los metadatos del `ResultSet` incluyen el número de columnas, y el nombre de cada columna.
- Los metadatos se pueden utilizar para escribir código de base de datos más “inteligente”. Por ejemplo, para generar tablas de datos genéricas.
- Este código podría usarse en helper classes. De esta manera se puede simplificar el código de la aplicación.

# Diseño de clases para acceso a datos

---

- Para ejemplos sencillos, la API de JDBC es suficiente.
- Para casos más complejos, la inclusión de código SQL directamente en el JSP no es una práctica muy recomendable y puede introducir problemas.
- El mejor lugar para colocar todo el código de acceso a la base de datos es en clases separadas.
- También debe considerarse si la implementación de los cálculos deben llevarse a cabo en Java o en SQL:
  - Para casos sencillos, código SQL puede ser mucho más rápido.
  - Para casos complejos, código en Java puede ser más sencillo de programar.

# Ejemplo: Tienda Virtual

---

- En este ejemplo retomaremos la tienda virtual que presentamos con anterioridad. La tienda vende productos electrodomésticos.
- Cada producto tiene las siguientes propiedades o atributos: id, nombre, descripción, precio, e imagen.
- Cada producto será representado como un objeto de la clase `Producto`.
- En nuestro ejemplo veremos como guardar los productos en la BD Relacional. También veremos como escribir consultas (queries) que recuperen una lista de productos, y un producto en particular.



# Clase Producto

---

Esta es la clase que representará los productos:

```
public class Producto {
    private int id;
    private String nombre;
    private String descripcion;
    private double precio;
    private String imagen;

    // los constructores,
    // los métodos get/set para cada atributo, y
    // el método toString() son omitidos
}
```

# Interface DBInterface

- `DBInterface` es la interface que definirá los métodos que podrán ser invocados desde la aplicación. Para nuestro ejemplo sólo tendremos tres métodos.

```
public interface DBInterface {  
    public Producto recuperaProducto(int id);  
    public ArrayList recuperaProductos();  
    public void insertaProducto(Producto producto);  
}
```

# Clase DBRelational:

## Implementación de DBInterface

- `DBRelational` es la implementación de la interface `DBInterface`. La clase `DBRelational` implementará los tres métodos definidos en `DBInterface`.

```
public class DBRelational implements DBInterface{

    public Producto recuperaProducto(int id){
        ... ..
    }
    public ArrayList recuperaProductos() {
        ... ..
    }
    public void insertaProducto(Producto producto) {
        ... ..
    }
}
```

# Estableciendo la conexión a la BD

Para establecer la conexión a la BD lo hacemos en el constructor de DBRelational:

```
public DBRelational(){
    try {
        //obtiene la conexión a la BD tienda
        //con el usuario root
        conn = DriverManager.getConnection
            ("jdbc:mysql://localhost/tienda?user=root");
    }
    catch(java.sql.SQLException e){
        System.out.println("Error en la conexión...");
        e.printStackTrace();
    }
}
```

# Guardando algunos productos

```
public void insertaProducto(Producto producto) {
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        //ejecuta el statement
        String sql = "INSERT INTO productos(nombre,
            descripcion, precio, imagen) VALUES ('" +
            producto.getNombre() + "', '" +
            producto.getDescripcion() + "', " +
            producto.getPrecio() + "', '" +
            producto.getImagen() + "')";
        stmt.execute(sql);
        stmt.close();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

# Guardando algunos productos

Para llamar al método `insertaProducto`:

```
public static void main(String[] args) {  
  
    DBRelational db = new DBRelational();  
  
    db.insertaProducto(new Producto (6, "Horno",  
        "Descripción horno", 1599, "horno.jpg"));  
  
    db.insertaProducto(new Producto (7, "Plancha",  
        "Descripción plancha", 560, "plancha.jpg"));  
  
}
```

# Recuperando todos los productos

```
public ArrayList recuperaProductos() {
    ArrayList listaProductos = new ArrayList();
    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        //ejecuta el query y recupera el resultset
        rs = stmt.executeQuery("SELECT * FROM productos");
        //iteramos sobre el ResultSet
        while (rs.next()) {
            listaProductos.add(new Producto(
                rs.getInt("idProducto"), rs.getString("nombre"),
                rs.getString("descripcion"), rs.getInt("precio"),
                rs.getString("imagen"));
        }
        stmt.close();
    }
    catch (SQLException e){
        e.printStackTrace();
    }
    return listaProductos;
}
```

# Recuperando un producto particular

```
public Producto recuperaProducto(int idProducto){
    Statement stmt = null;
    ResultSet rs = null;
    Producto producto = null;
    try {
        stmt = conn.createStatement();
        //ejecuta el query y recupera el resultset
        rs = stmt.executeQuery("SELECT * FROM productos WHERE" +
                               "idProducto=" + idProducto);
        //iteramos sobre el ResultSet
        if (rs.next()) {
            producto = new Producto(
                rs.getInt("idProducto"), rs.getString("nombre"),
                rs.getString("descripcion"), rs.getInt("precio"),
                rs.getString("imagen"));
        }
        stmt.close();
    }
    catch (SQLException e){
        e.printStackTrace();
    }
    return producto;
}
```



# JSP que muestra todos los productos (fragmento 1 de 2)

```
<body>
  <h1>Tienda Virtual</h1>
  <table border="1">
    <tr>
      <td>Imagen</td>
      <td>Id</td>
      <td>Nombre</td>
      <td>Descripci&oacute;</td>
      <td>Precio</td>
    </tr>
  <%
    DBInterface db = new DBRelational();
    ArrayList lista = db.recuperaProductos();
    Iterator it = lista.iterator();
    while (it.hasNext()){
      Producto prod = (Producto)it.next();
    %>
```

# JSP que muestra todos los productos (fragmento 2 de 2)

```
<tr>
  <td></td>
  <td>
    <a href="showProd.jsp?prodId=<%=prod.getId() %>">
      <%=prod.getId() %></a>
    </td>
  <td><%=prod.getNombre() %></td>
  <td><%=prod.getDescripcion() %></td>
  <td><%=prod.getPrecio() %></td>
</tr>

<%
}
%>
</table>

</body>
```

# JSP que muestra todos los productos



JSP Page - Internet Explorer provided by Dell

http://localhost:8080/WebDinamico/tiendaProductos.jsp

Archivo Edición Ver Favoritos Herramientas Ayuda

JSP Page

## Tienda Virtual

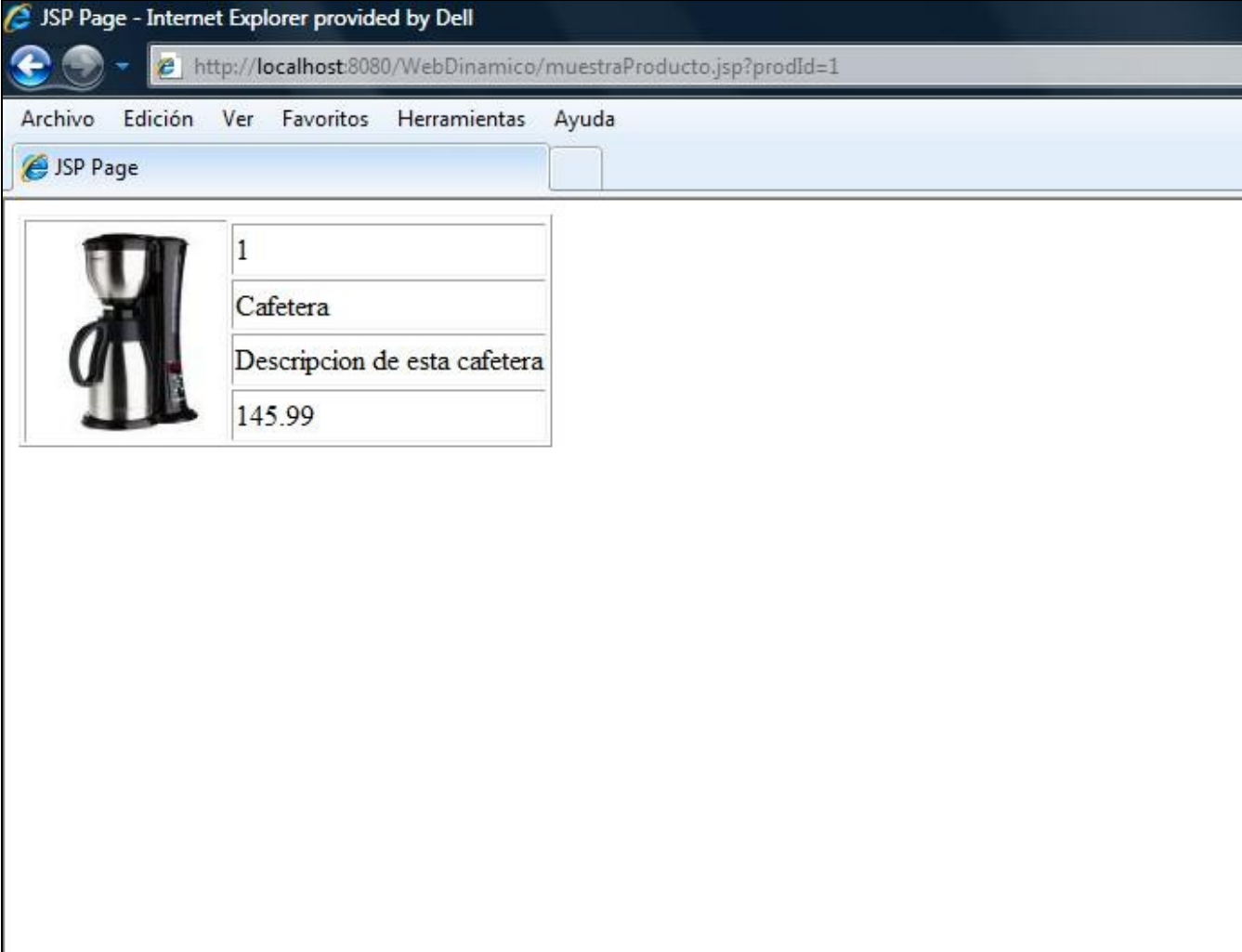
Imagen	Id	Nombre	Descripción	Precio
	<a href="#">1</a>	Cafetera	Descripcion de esta cafetera	145.99
	<a href="#">2</a>	Licuadora	Descripcion de esta licuadora	100.5
	<a href="#">3</a>	Horno	Descripcion de esta horno	134.67

# JSP que muestra un producto particular


```
<body>
  <table border="1">
  <%
String prodId = request.getParameter("prodId");
int id = Integer.parseInt(prodId);

DBInterface db = new DBRelational(ruta);
Producto prod = db.recuperaProducto(id);
%>
    <tr><td rowspan="5">
      </td>
    </tr>
    <tr><td><%=prod.getId()%></td></tr>
    <tr><td><%=prod.getNombre()%></td></tr>
    <tr><td><%=prod.getDescripcion()%></td></tr>
    <tr><td><%=prod.getPrecio()%></td></tr>
  </table>
</body>
```

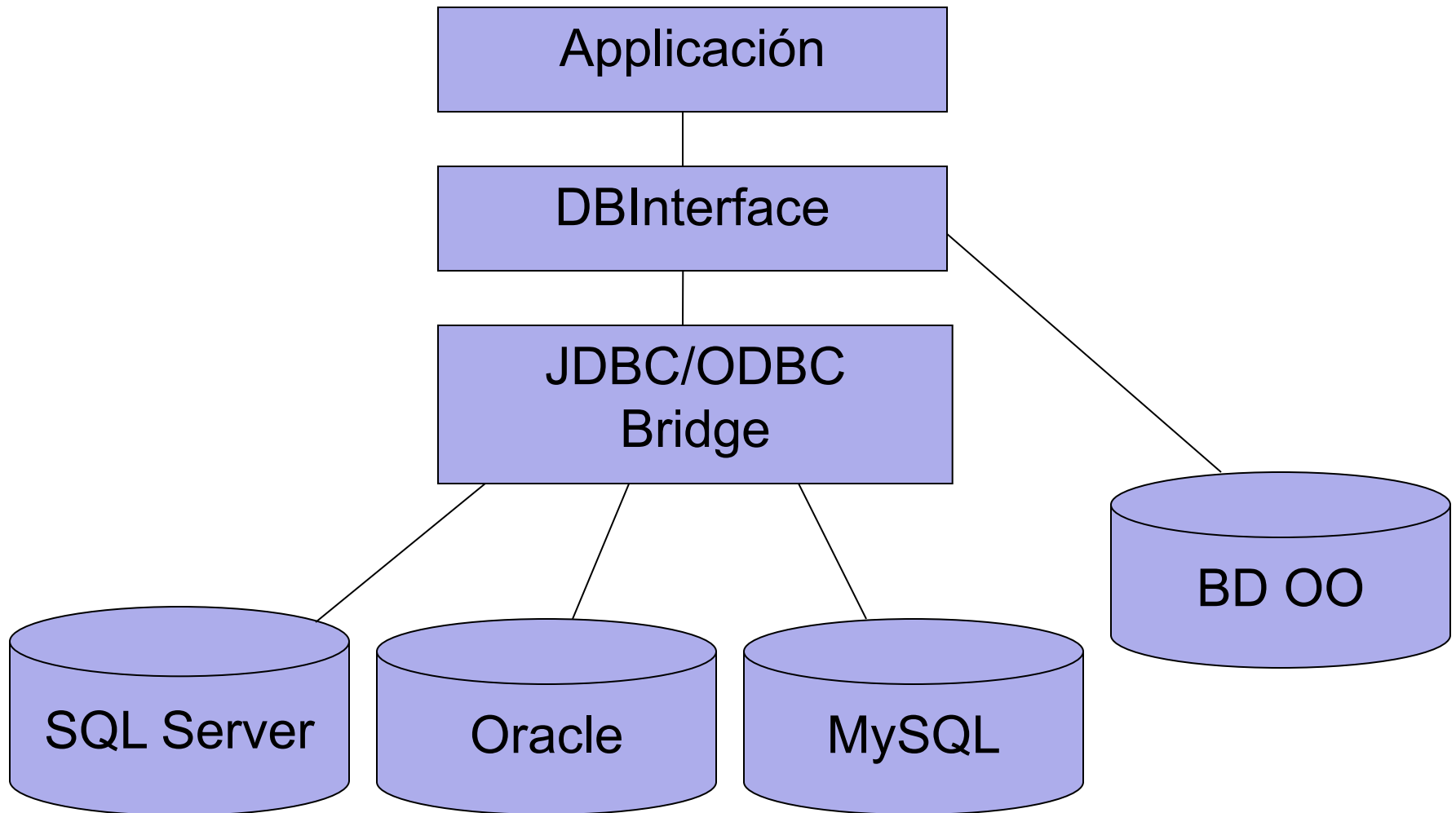
# JSP que muestra un producto particular



The screenshot shows an Internet Explorer browser window titled "JSP Page - Internet Explorer provided by Dell". The address bar displays the URL "http://localhost:8080/WebDinamico/muestraProducto.jsp?prodId=1". The menu bar includes "Archivo", "Edición", "Ver", "Favoritos", "Herramientas", and "Ayuda". The browser window contains a table with the following data:

	1
	Cafetera
	Descripcion de esta cafetera
	145.99

# Acceso a Bases de Datos



# Generalización de código de acceso a base de datos

---

- No es buena práctica colocar código SQL directamente en el código JSP.
- Definir una clase de acceso a BD como una *interface*, y así se pueden tener las siguientes ventajas:
  - Código más conciso
  - Diferentes implementaciones
  - Acceso a diferentes fuentes de datos
  - Código más fácil de entender
  - Código más fácil de depurar
- En nuestro ejemplo utilizamos la interface `DBInterface` y tres diferentes implementaciones: BD Relacional, BD Orientada a Objetos, datos en memoria.

# Transacciones

---

Las transacciones tienen cuatro propiedades (ACID):

- A. Atomicity. Una transacción debe ser atómica. Todas las operaciones en una transacción deben de llevarse a cabo o ninguna. TODO o NADA.
- C. Consistency. Una transacción debe dejar a la base de datos en un estado consistente después de llevarse a cabo.
- I. Isolation. Las transacciones deben ser independientes-aisladas unas de otras. No interferir una con otra.
- D. Durability. Toda transacción debe garantizar que los datos afectados son durables (que persistirán).



# Transacciones en JDBC

## Ejemplo de una transacción en JDBC:

```
try {
    con.setAutoCommit(false);
    statement = con.createStatement();
    statement.executeUpdate( update1 );
    statement.executeUpdate( update2 );
    //si las dos operaciones se llevaron a cabo
    //exitosamente entonces comprometemos los cambios
    con.commit();
}
catch(SQLException e) {
    //si algo salió mal, cancelamos los cambios;
    //echamos hacia atrás los cambios
    con.rollback();
}
```

# Transacciones en JDBC

Del ejemplo anterior tenemos tres sentencias importantes que se requieren para manejar transacciones en JDBC.

- `con.setAutoCommit(false);`

Esto indica que iniciaremos una transacción. Por default la propiedad `AutoCommit` está en `true`, lo cual significa que cualquier ejecución de una sentencia SQL es por sí sola una transacción.

Colocando `AutoCommit` en `false` garantizamos que los cambios no se registren en la base de datos hasta que la transacción haya finalizado. Esto es hasta que se ejecute `Commit` o `Rollback`.

# Transacciones en JDBC

- `con.commit()` ;

Cuando se llega a esta sentencia, significa que todas las operaciones que forman parte de la transacción se llevaron a cabo exitosamente. Ejecutando `commit` garantizamos que los cambios realizados sean llevados a cabo en la BD.

- `con.rollback()` ;

El ejecutar esta sentencia significa que alguna de las operaciones que forman parte de la transacción falló. `rollback` garantizará que ninguna de las operaciones que forman parte de la transacción, sean llevadas a cabo en la base de datos.

# Resumen

---

Hemos cubierto lo siguiente:

- Cómo utilizar JDBC para establecer la conectividad con una base de datos relacional; así como correr queries y sentencias SQL sobre una fuente de datos JDBC.
- Cómo tener el código de acceso a BD en clases separadas, para tener JSPs más claras.

# Resumen

---

- Cómo separar interface de implementación. De esta forma tener diferentes fuentes de datos (bases de datos).
- Tienda virtual que accesa una BD relacional.
- Transacciones y su manejo en JDBC.