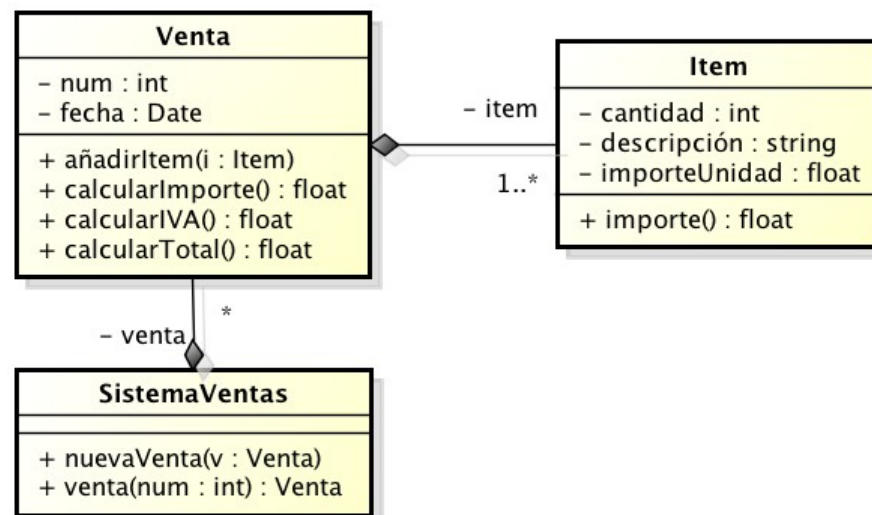


Seminario: Acceso a bases de datos desde Java

1. Persistencia de clases
2. Serialización
3. Java Database Connection (JDBC)
4. Data Access Objects (DAO)
5. Object-Relational Mapping (ORM)
6. Java Persistence API (JPA)

Acceso a bases de datos

- Necesario en la mayoría de aplicaciones no triviales
- Normalmente no se contempla en el diseño inicial
- La implementación directa es una aplicación que trabaja íntegramente en memoria



- Implementación de la clase Item:

```
package ventasbd;

public class Item {
    int cantidad;
    String descripcion;
    float importeUnidad;

    public Item(int cantidad, String descripcion, float importeUnidad) {
        this.cantidad = cantidad;
        this.descripcion = descripcion;
        this.importeUnidad = importeUnidad;
    }

    public float importe() {
        return cantidad * importeUnidad;
    }
}
```

•Implementación de la clase Venta:

```
package ventasbd;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;

public class Venta {
    int num;
    Date fecha;
    ArrayList<Item> items;

    public Venta(int num) {
        this.num = num;
        fecha = new Date();
        items = new ArrayList<Item>();
    }

    public void anadirItem(Item i) {
        items.add(i);
    }
}
```

```
public float calcularImporte() {
    Iterator<Item> i = items.iterator();
    float suma = 0;

    while (i.hasNext()) {
        suma += i.next().importe();
    }

    return suma;
}

public float calcularIVA() {
    return 0.18f * calcularImporte();
}

public float calcularTotal() {
    return 1.18f * calcularImporte();
}
}
```

- Implementación de la clase SistemaVentas

```
package ventasbd;

import java.util.TreeMap;

class ErrorCreacionVenta extends Exception {}

public class SistemaVentas {
    TreeMap<Integer, Venta> ventas;

    public SistemaVentas() {
        ventas = new TreeMap<Integer, Venta>();
    }

    public void nuevaVenta(Venta v) {
        ventas.put(v.num, v);
    }

    public Venta venta(int num) {
        return ventas.get(num);
    }
}
```

•Implementación de la clase VentasBD:

```
package ventasbd;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.text.SimpleDateFormat;

public class VentasBD {

    public static void main(String[] args) {
        SistemaVentas sistemaVentas = new SistemaVentas();

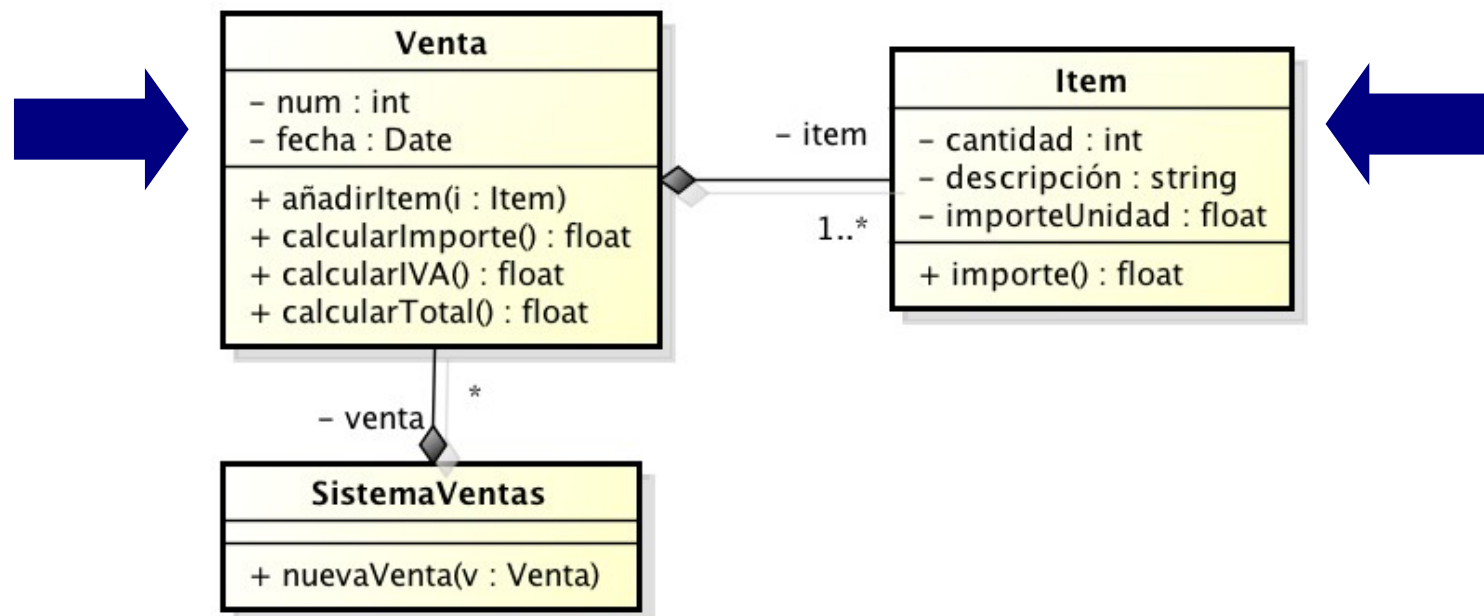
        Venta venta = new Venta(1);
        venta.anadirItem(new Item(4, "Barritas energéticas", 2.0f));
        venta.anadirItem(new Item(1, "Aceite cadena", 8.0f));
        venta.anadirItem(new Item(1, "Par de guantes", 15.0f));

        System.out.println ("Factura " + Integer.toString(venta.num) +
            " importe: " + venta.calcularTotal());

        sistemaVentas.nuevaVenta(venta);
    }
}
```

Persistencia de clases

- Normalmente existirán varias clases cuya información debe guardarse de manera **persistente**
- Su información no debe perderse al cerrar la aplicación



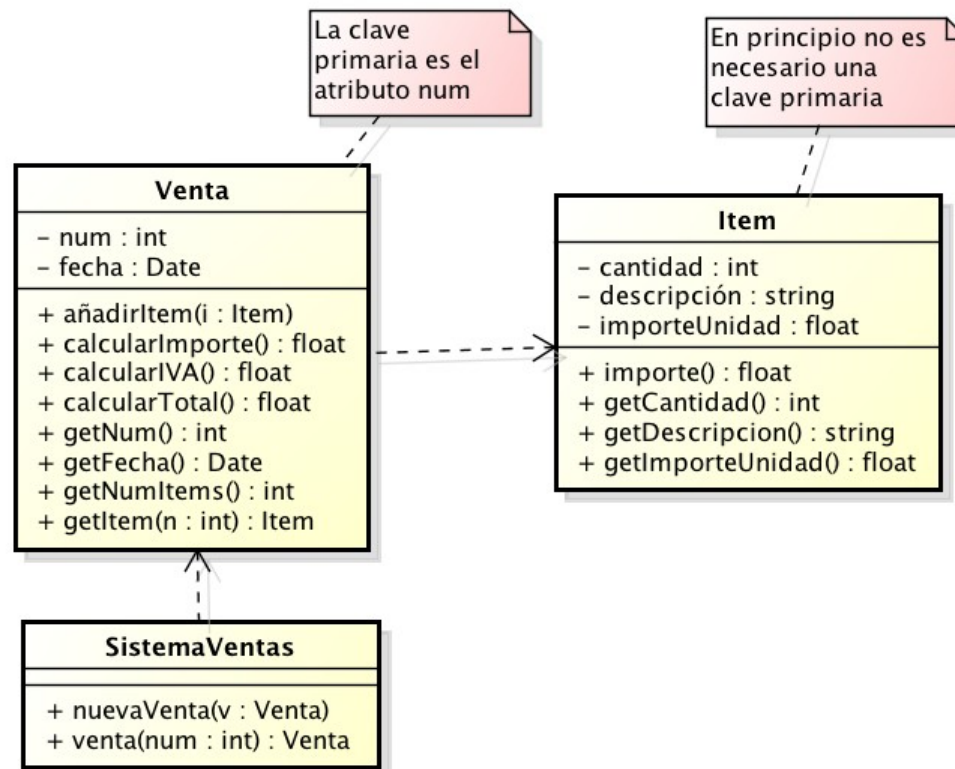
Solución mediante serialización

- La **serialización** en un fichero es una solución sencilla para proporcionar persistencia pero tiene varios inconvenientes:
 - Puede que por su volumen no sea posible mantener todos los objetos en memoria
 - El tiempo de arranque de la aplicación puede ser largo
 - Un fallo inesperado en la aplicación puede hacer perder información

Adaptación de las clases persistentes

- Las clases persistentes deben cumplir en general varias condiciones:
 - Incluir un atributo que funcione como clave primaria. Si no existe, se añade.
 - Incluir observadores y modificadores para leer/escribir los atributos que van a ser guardados en la base de datos
 - Muchas relaciones en memoria desaparecen y se sustituyen por relaciones entre tablas

- Diagrama del ejemplo tras eliminar las relaciones y añadir observadores



powered by astah®

Creación de tablas

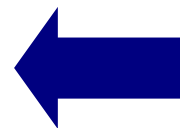
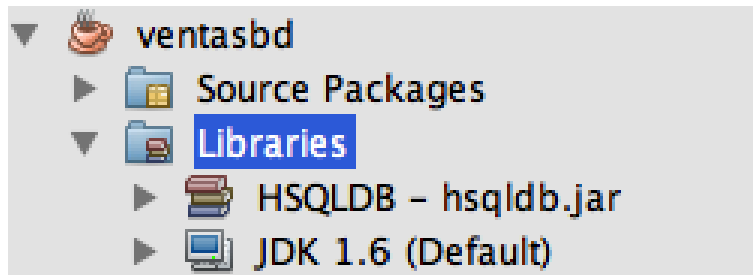
- Normalmente crearemos una tabla por clase persistente
- Las relaciones entre clases deben trasladarse a relaciones E-R (1 a 1, 1 a muchos, muchos a muchos) y éstas a conexiones mediante llaves foráneas

```
create table ventas (  
    num int primary key,  
    fecha timestamp,  
);
```

```
create table items (  
    num_venta int references ventas(num),  
    cantidad int,  
    descripcion varchar(40),  
    importeUnidad numeric(6,2)  
);
```

JDBC

- JDBC (Java Database Connectivity) es el API básico de Java para acceso a bases de datos
- Sencillo de utilizar
- Conexión con virtualmente todas las bases de datos del mercado
- Es necesario mucho código “de fontanería” para traslación de clases a SQL y viceversa



Incluir driver JDBC de la base de datos en proyecto

```
Connection conn = null;
Statement stmt = null;

// Conexión
try {
    conn = DriverManager.getConnection(
        "jdbc:hsqldb:file:ventasdb",
        "sa", "");
    stmt = conn.createStatement();
}
catch(SQLException e) {
    System.out.println("Error al conectar con BD");
    return;
}

// Ejecución de comando SQL
try {
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

    stmt.execute("insert into ventas values (" +
        Integer.toString(venta.getNum()) + ", " +
        "'" + df.format(venta.getFecha()) + "'" +
        ")");
}
catch(SQLException e) {
    System.out.println("Error al guardar venta");
}

// Desconexión
try {
    conn.close();
}
catch(SQLException e) {
}
```

Gestión de la conexión

- Es aconsejable encapsular la conexión a la BD en un singleton de forma que esté disponible en cualquier punto de la aplicación

```
class ErrorConexionBD extends Exception {}

public class ConexionBD {
    Connection conn;
    Statement stmt;

    static ConexionBD instancia = null;

    private ConexionBD() throws ErrorConexionBD {
        try {
            conn = DriverManager.getConnection("jdbc:hsqldb:file:ventasdb");
            stmt = conn.createStatement();
        }
        catch(SQLException e) {
            throw new ErrorConexionBD();
        }
    }
}
```

```
public Connection getConnection() {
    return conn;
}

public Statement getStatement() {
    return stmt;
}

public static void crearConexion() throws ErrorConexionBD {
    if (instancia == null) {
        instancia = new ConexionBD();
    }
}

public static ConexionBD instancia() {
    return instancia;
}

public static void desconectar() {
    if (instancia != null) {
        try {
            instancia.stmt.execute("shutdown");
            instancia.stmt.close();
            instancia.conn.close();
            instancia = null;
        }
        catch(SQLException e) {
        }
    }
}
```

El método rápido y sucio

- Sustituiremos las operaciones sobre estructuras de datos por código equivalente JDBC para acceso a bases de datos

```
public class SistemaVentas {
    // TreeMap<Integer, Venta> ventas;

    public SistemaVentas() {
        // ventas = new TreeMap<Integer, Venta>();
    }

    public void nuevaVenta(Venta v) throws ErrorCreacionVenta {
        // ventas.put(v.num, v);

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        try {
            ConexionBD.instancia().getStatement().execute(
                "insert into ventas values (" +
                Integer.toString(v.getNum()) + ", " +
                "'" + sdf.format(v.getFecha()) + "'"
            );
        } catch (SQLException e) {
            throw new ErrorCreacionVenta();
        }
    }
}
```



```
public Venta venta(int num) {  
    // return ventas.get(num);  
  
    try {  
        ResultSet rs = ConexionBD.instancia().getStatement().executeQuery(  
            "select fecha from ventas where num=" + Integer.toString(num)  
        );  
  
        if (rs.next()) {  
            return new Venta(num, rs.getDate(1));  
        }  
    } catch (SQLException e) {  
        return null;  
    }  
}
```

```
public class Venta {
    int num;
    Date fecha;
    //ArrayList<Item> items;

    public Venta(int num) {
        this.num = num;
        fecha = new Date();
        //items = new ArrayList<Item>();
    }

    public void anadirItem(Item i) throws ErrorCreacionItem {
        //items.add(i);

        try {
            ConexionBD.instancia().getStatement().execute(
                "insert into items values (" +
                Integer.toString(num) + ", " +
                Integer.toString(i.getCantidad()) + ", " +
                "'" + i.getDescripcion() + "', " +
                Float.toString(i.getImporteUnidad()) + ")"
            );

        } catch (SQLException e) {
            throw new ErrorCreacionItem();
        }
    }

    public int getNum() {
        return num;
    }

    public Date getFecha() {
        return fecha;
    }

    public void setFecha(Date fecha) {
        this.fecha = fecha;
    }
}
```

```
public int getNumItems() {
    try {
        ResultSet rs = ConexionBD.instancia().getStatement().executeQuery(
            "select count(*) from items where num_venta=" +
            Integer.toString(num)
        );
        return rs.getInt(1);
    } catch (SQLException e) {
    }

    return 0;
}

public Collection<Item> getItems() {
    ArrayList<Item> items = new ArrayList<Item>();

    try {
        ResultSet rs = ConexionBD.instancia().getStatement().executeQuery(
            "select cantidad, descripcion, importeUnidad from items where " +
            "num_venta=" + Integer.toString(num)
        );

        while (rs.next()) {
            items.add(new Item(rs.getInt(1), rs.getString(2), rs.getFloat(3)));
        }

    } catch (SQLException e) {
    }

    return items;
}
```

```
public float calcularImporte() {
    try {
        ResultSet rs = ConexionBD.instancia().getStatement().executeQuery(
            "select sum(cantidad * importeUnidad) from items where num_venta=" +
            Integer.toString(num)
        );
        return rs.getInt(1);
    } catch (SQLException e) {
    }

    return 0;
}

public float calcularIVA() {
    return 0.18f * calcularImporte();
}

public float calcularTotal() {
    return 1.18f * calcularImporte();
}
}
```

```
public class VentasBD {
    public static void main(String[] args) {
        try {
            ConexionBD.crearConexion();
        }
        catch(ErrorConexionBD e) {
            System.out.println("Error de conexión con BD");
        }

        SistemaVentas sistemaVentas = new SistemaVentas();

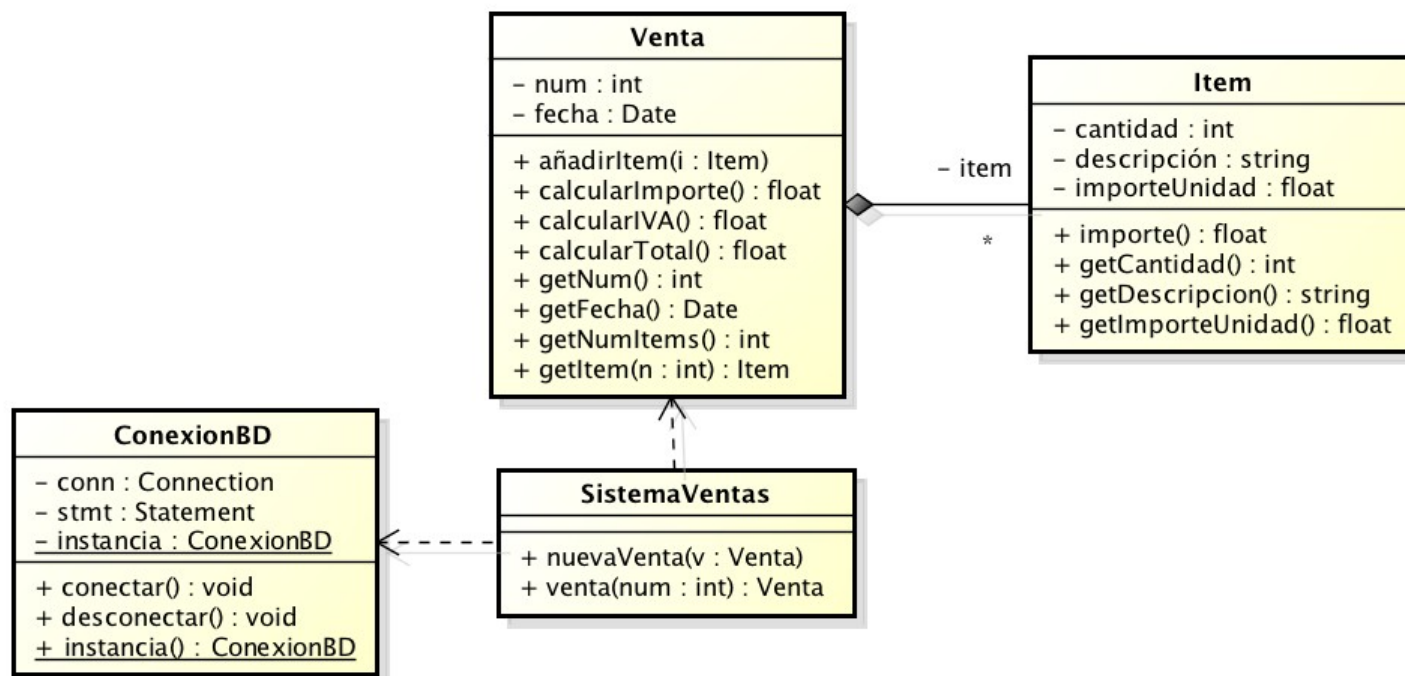
        Venta venta = null;
        try {
            venta = new Venta(3);

            sistemaVentas.nuevaVenta(venta);
            venta.anadirItem(new Item(4, "Barritas energéticas", 2.0f));
            venta.anadirItem(new Item(1, "Aceite cadena", 8.0f));
            venta.anadirItem(new Item(1, "Par de guantes", 15.0f));
        }
        catch(Exception e) {
            System.out.println("Error al registrar la venta");
        }

        ConexionBD.desconectar();
    }
}
```

Operaciones en cascada

- En el caso de las composiciones es frecuente al cargar un objeto el cargar en cascada los objetos compuestos
- De esta forma ya están disponibles en memoria y se evitan accesos posteriormente



```
public class SistemaVentas {
    public SistemaVentas() {

    }

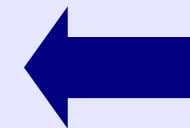
    public void nuevaVenta(Venta v) throws ErrorCreacionVenta {

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        try {
            ConexionBD.instancia().getStatement().execute(
                "insert into ventas values (" +
                Integer.toString(v.getNum()) + ", " +
                "'" + sdf.format(v.getFecha()) + "'"
                );

            for (Item item: v.getItems()) {
                ConexionBD.instancia().getStatement().execute(
                    "insert into items values (" +
                    Integer.toString(v.getNum()) + ", " +
                    Integer.toString(item.getCantidad()) + ", " +
                    "'" + item.getDescripcion() + "', " +
                    Float.toString(item.getImporteUnidad()) + ")"
                );
            }

        } catch (SQLException e) {
            throw new ErrorCreacionVenta();
        }
    }
}
```



Creación en
cascada
de los items

```
public Venta venta(int num) {
    Venta v = null;

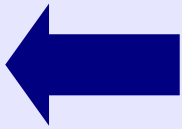
    try {
        ResultSet rs = ConexionBD.instancia().getStatement().executeQuery(
            "select fecha from ventas where num=" + Integer.toString(num)
        );

        if (rs.next()) {
            v = new Venta(num);
            v.setFecha(rs.getDate(1));

            ResultSet rsi = ConexionBD.instancia().getStatement().executeQuery(
                "select cantidad, descripcion, importeUnidad from items" +
                "where num_venta=" + Integer.toString(num)
            );

            while (rsi.next()) {
                v.anadirItem(new Item(rsi.getInt(1),
                                     rsi.getString(2),
                                     rsi.getFloat(3)));
            }
        }
    } catch (Exception e) {
    }

    return v;
}
```



Carga en
cascada
de los items


```
public class VentasBD {
    public static void main(String[] args) {
        // TODO code application logic here
        try {
            ConexionBD.crearConexion();
        }
        catch(ErrorConexionBD e) {
            System.out.println("Error de conexión con BD");
        }

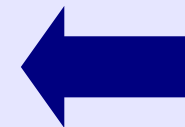
        SistemaVentas sistemaVentas = new SistemaVentas();

        Venta venta = null;
        try {
            venta = sistemaVentas.venta(3);
        }
        catch(Exception e) {
            System.out.println("Error al obtener items");
        }

        Collection<Item> items = venta.getItems();

        for (Item i: items) {
            System.out.println(Integer.toString(i.getCantidad()) +
                " " + i.getDescripcion() + " " + Float.toString(i.getImporteUnidad()));
        }

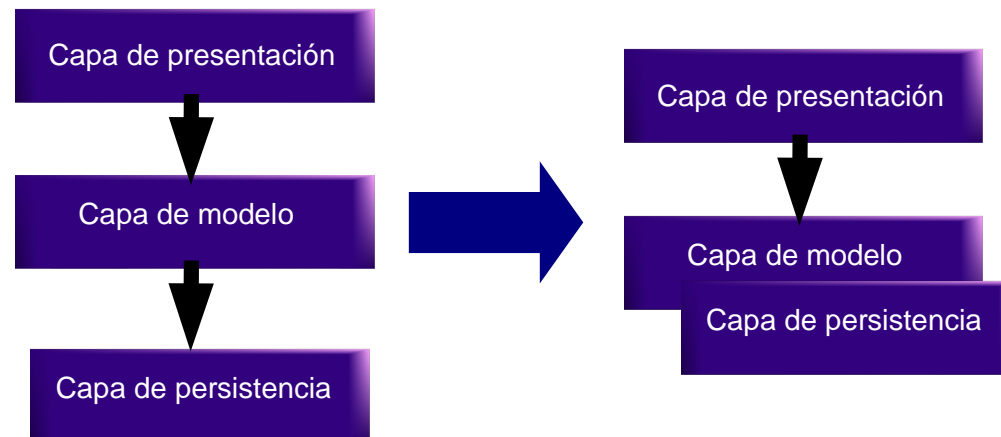
        ConexionBD.desconectar();
    }
}
```



Carga completa de
la venta

Por qué la solución rápida y sucia no es buena

- Las soluciones anteriores son nefastas porque mezclan la lógica del negocio con el acceso a bases de datos
- El código SQL está repartido por todos los objetos del modelo
- Un cambio en una tabla implica repasar todo el modelo

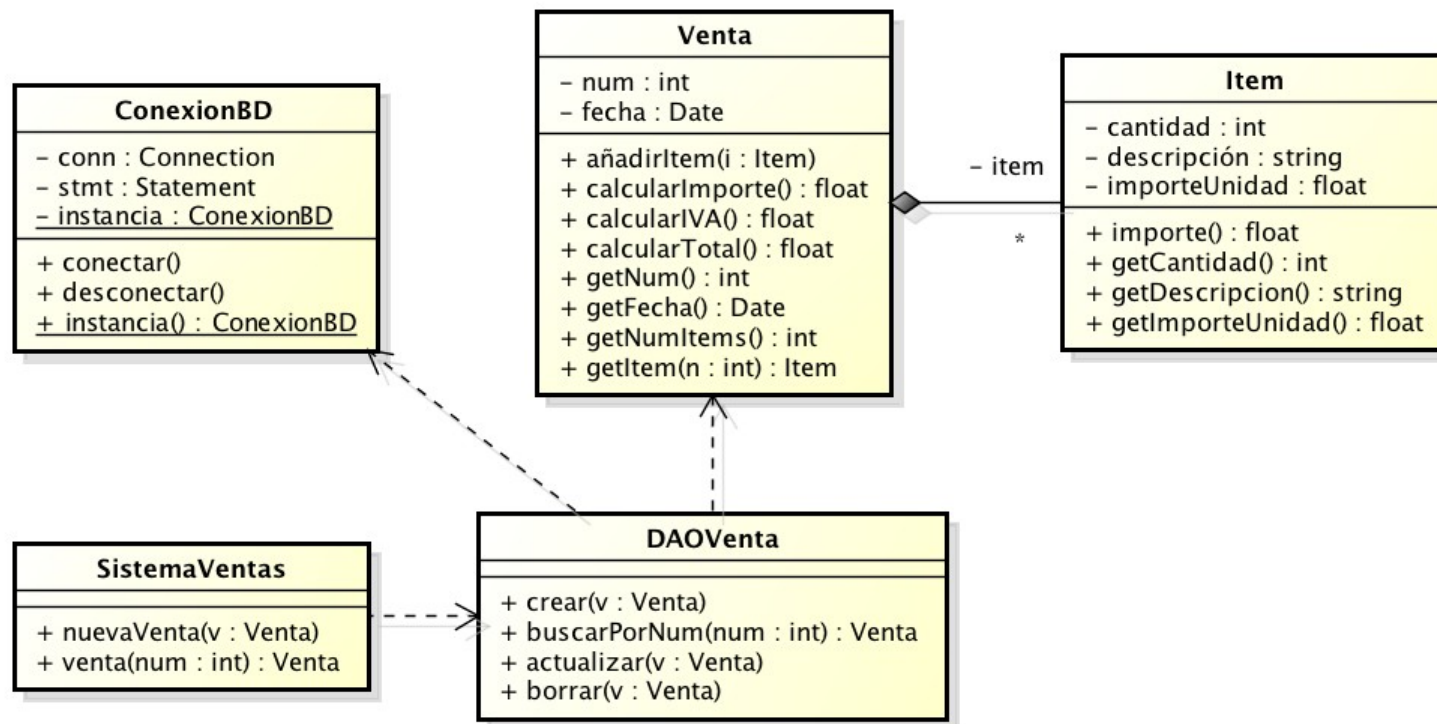


Por qué la solución
rápida y sucia no es
buena

El patrón DAO

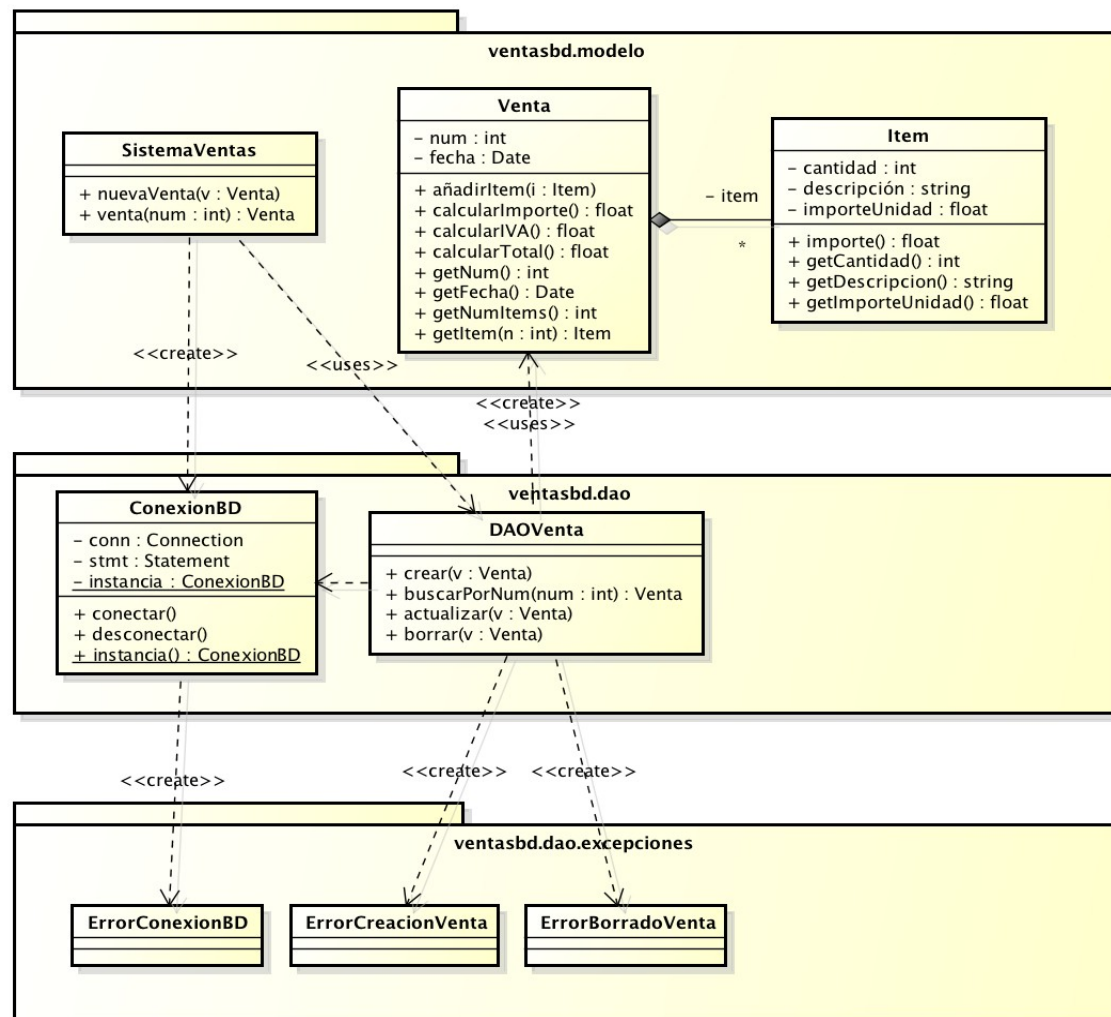
- El patrón DAO (Data Access Object) nos proporciona una receta para aislar el código del modelo del código del acceso a la base de datos
- Un DAO es una clase (normalmente singleton) que hace de intermediario con la base de datos para una clase de objetos dada
- Normalmente implementa un esquema CRUD (Create, Retrieve, Update and Delete)

- Este sería el esquema de nuestro ejemplo con un DAO para gestionar las ventas



powered by astah*

- Este sería el esquema de nuestro ejemplo con un DAO para gestionar las ventas



```
public class DAOVenta {
    static DAOVenta instancia = null;

    SimpleDateFormat sdf;

    private DAOVenta() {
        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    }

    public void crear(Venta v) throws ErrorCreacionVenta {
        try {
            ConexionBD.instancia().getStatement().execute(
                "insert into ventas values (" +
                Integer.toString(v.getNum()) + ", " +
                "'" + sdf.format(v.getFecha()) + "'" +
                ");

            for (Item item: v.getItems()) {
                ConexionBD.instancia().getStatement().execute(
                    "insert into items values (" +
                    Integer.toString(v.getNum()) + ", " +
                    Integer.toString(item.getCantidad()) + ", " +
                    "'" + item.getDescripcion() + "', " +
                    Float.toString(item.getImporteUnidad()) + ")"
                );
            }
        } catch (SQLException e) {
            throw new ErrorCreacionVenta();
        }
    }
}
```

```
public Venta buscarPorNum(int num) {
    Venta v = null;

    try {
        ResultSet rs = ConexionBD.instancia().getStatement().executeQuery(
            "select fecha from ventas where num=" + Integer.toString(num)
        );

        if (rs.next()) {
            v = new Venta(num);
            v.setFecha(rs.getDate(1));

            ResultSet rsi = ConexionBD.instancia().getStatement().executeQuery(
                "select cantidad, descripcion, importeUnidad from items where " +
                "num_venta=" + Integer.toString(num)
            );

            while (rsi.next()) {
                v.anadirItem(new Item(rsi.getInt(1), rsi.getString(2),
                    rsi.getFloat(3)));
            }
        }
    } catch (Exception e) {
    }

    return v;
}
```

```
public void actualizar(Venta v) {
    // Implementar
}

public void borrar(Venta v) throws ErrorBorradoVenta {
    try {
        ConexionBD.instancia().getStatement().execute(
            "delete from items where num_venta=" + Integer.toString(v.getNum()));
        ConexionBD.instancia().getStatement().execute(
            "delete from ventas where num=" + Integer.toString(v.getNum()));
    } catch (SQLException e) {
        throw new ErrorBorradoVenta();
    }
}

public static DAOVenta instancia() {
    if (instancia == null) {
        instancia = new DAOVenta();
    }

    return instancia;
}
}
```



```
package ventasbd.modelo;

import ventasbd.dao.exception.ErrorConexionBD;
import ventasbd.dao.ConexionBD;
import ventasbd.dao.DAOVenta;
import ventasbd.dao.exception.ErrorCreacionVenta;

public class SistemaVentas {

    public SistemaVentas() throws ErrorConexionBD {
        ConexionBD.crearConexion();
    }

    public void nuevaVenta(Venta v) throws ErrorCreacionVenta {
        DAOVenta.instancia().crear(v);
    }

    public Venta venta(int num) {
        return DAOVenta.instancia().buscarPorNum(num);
    }

    public void cerrar() {
        ConexionBD.desconectar();
    }
}
```

```
public class VentasBD {
    public static void main(String[] args) {
        SistemaVentas sistemaVentas = null;
        try {
            sistemaVentas = new SistemaVentas();
        }
        catch(ErrorConexionBD e) {
            System.out.println("Error de conexión con BD");
            return;
        }

        Venta venta = sistemaVentas.venta(3);

        if (venta != null) {
            System.out.println("Venta num. " + Integer.toString(venta.getNum()) +
                               " Fecha: " + new SimpleDateFormat("dd-MM-yyyy HH:mm").format(
                                   venta.getFecha()));

            System.out.println("\nProductos:");
            Collection<Item> items = venta.getItems();

            for (Item i: items) {
                System.out.println(Integer.toString(i.getCantidad()) + " " +
                                    i.getDescripcion() + " " +
                                    Float.toString(i.getImporteUnidad()));
            }
            System.out.println("\nSuma: " + Float.toString(venta.calcularImporte()));
            System.out.println("IVA: " + Float.toString(venta.calcularIVA()));
            System.out.println("Total: " + Float.toString(venta.calcularTotal()));
        }

        sistemaVentas.cerrar();
    }
}
```

Por qué la solución anterior es mejorable

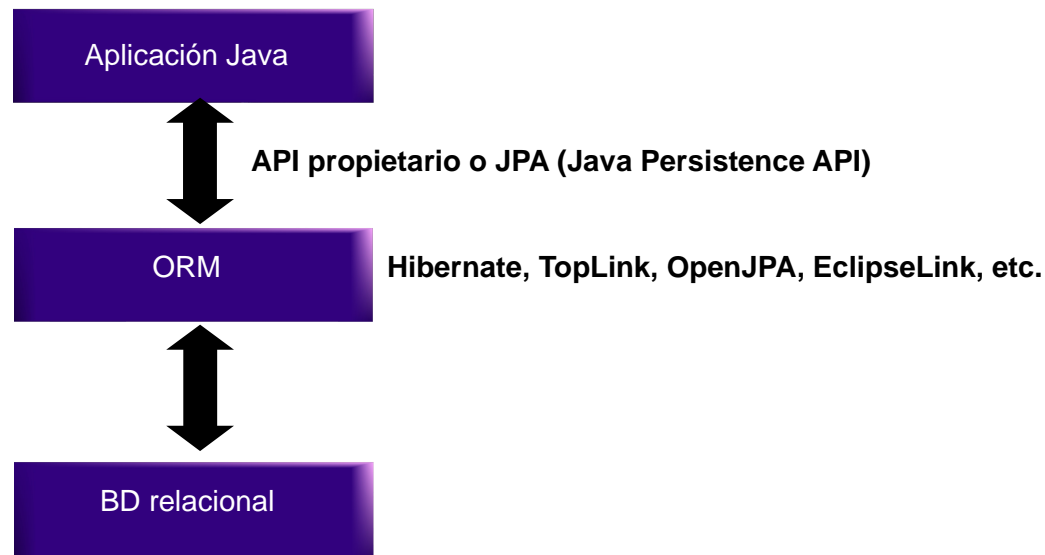
- Mucho código “poco creativo” de traducción de objetos a sentencias SQL y viceversa
- Lento, tedioso y propenso a errores
- Un cambio en el esquema de la base de datos obliga a revisar todas las sentencias SQL

Bases de datos orientadas a objetos

- Surgieron a principios de los 90 para resolver el problema de “impedancia” entre los lenguajes OO y las bases de datos relacionales
- Permiten guardar estructuras de objetos relacionados directamente en la base de datos
- Su difusión ha sido limitada:
 - Falta de lenguajes estándar de consulta como SQL
 - Desconfianza por parte del mercado, que sigue confiando en las bases de datos relacionales

Mapeado Objeto/Relacional

- Solución mixta: permite usar una base de datos relacional pero usando una interfaz OO
- Un mapeado objeto/relacional u ORM se encarga de generar el código SQL necesario para mapear los objetos en memoria con tablas en la BD



Ventajas de los ORMs

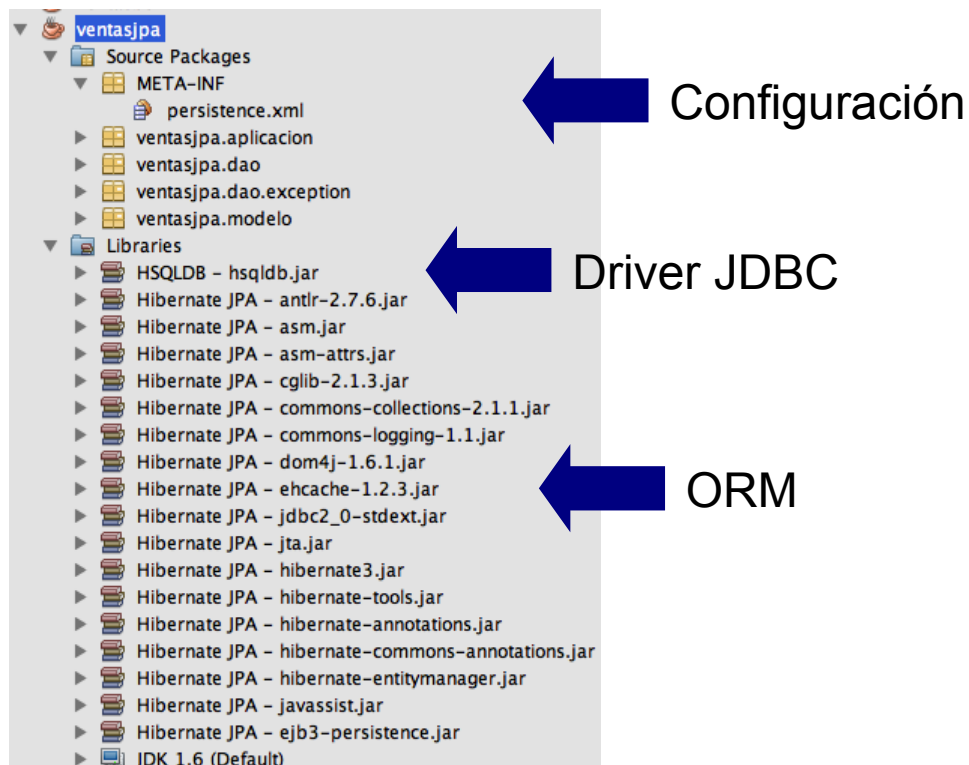
- Código mucho más simple y legible, sin SQL empotrado
- Fácil migración de un SGBD a otro
- Requiere pocos o ningunos cambios en el diagrama de clases inicial → las relaciones entre clases se mantienen

Java Persistence API

- El Java Persistence API o JPA es un API estándar propuesto por Sun para trabajar con ORMs
- La mayoría de los ORMs actuales soportan JPA (aparte de un API propio en muchos casos)
- Permite cambiar de un ORM a otro sin tener que modificar el código

Configuración de JPA

- Un proyecto que use JPA debe incluir:
 - El driver JDBC de la BD como librería
 - El ORM de la BD como librería
 - El fichero de configuración persistence.xml



- El fichero persistence.xml define una unidad de persistencia, que recopila la información necesaria para trabajar con JPA:
 - Nombre de la unidad de persistencia
 - Tipo de base de datos y driver JDBC
 - Usuario y password de la base de datos
 - Si se desea generar el esquema de la base de datos automáticamente
 - La lista de clases persistentes
- Dependiente del ORM (ver documentación).
Netbeans tiene asistentes de ayuda

•Conexión con HSQLDB usando Hibernate como ORM:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

  <persistence-unit name="ventasPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>ventasjpa.modelo.Item</class>
    <class>ventasjpa.modelo.Venta</class>
    <properties>
      <property name="toplink.target-database" value="HSQL"/>
      <property name="hibernate.connection.username" value="sa"/>
      <property name="hibernate.connection.driver_class" value="org.hsqldb.jdbcDriver"/>
      <property name="hibernate.connection.password" value=""/>
      <property name="hibernate.connection.url"
        value="jdbc:hsqldb:file:/Users/ajrueda/NetbeansProjects/ventasjpa/ventasdb"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

Mapeado de clases

- Cada clase a mapear debe cumplir con lo siguiente:
 - Implementar la interfaz *Serializable*
 - Incluir un constructor sin argumentos
 - Incluir la anotación `@Entity`
 - Incluir un atributo que funcione como clave primaria, anotado con `@Id`

```
@Entity
public class Venta implements Serializable {
    @Id
    int num;
    @Temporal(TemporalType.TIMESTAMP)
    Date fecha;

    @OneToMany(cascade=CascadeType.ALL)
    List<Item> items;

    public Venta(int num) {
        this.num = num;
        fecha = new Date();
        items = new ArrayList<Item>();
    }

    public Venta() {
        this(-1);
    }

    public void anadirItem(Item i) {
        items.add(i);
    }

    public int getNum() {
        return num;
    }

    public Date getFecha() {
        return fecha;
    }

    public void setFecha(Date fecha) {
        this.fecha = fecha;
    }
    ...
}
```

```
@Entity
public class Item implements Serializable {
    @Id
    @GeneratedValue
    long id;

    int cantidad;
    String descripcion;
    float importeUnidad;

    public Item() {
        this(0, "", 0);
    }

    public Item(int cantidad,
                String descripcion,
                float importeUnidad) {
        this.cantidad = cantidad;
        this.descripcion = descripcion;
        this.importeUnidad = importeUnidad;
    }

    public int getCantidad() {
        return cantidad;
    }
    ...
}
```

- Puede indicarse el nombre de la tabla donde se va a almacenar la entidad y el nombre de la columna asociada a cada atributo
 - `@Entity(name="Ventas")`
 - `@Column(name="Importe")`
- Los atributos que no deban ser guardados se marcan con `@Transient`
- Debe indicarse la forma en que se mapea un tipo temporal (Date, Calendar) en la base de datos (DATE, TIME, TIMESTAMP)
 - `@Temporal(TemporalType.TIMESTAMP)`
- La clave puede ser autogenerada por la base de datos y por varios métodos
 - `@GeneratedValue`

Mapeado de relaciones

- Las relaciones deben anotarse indicando el tipo:
 - @OneToOne
 - @OneToMany
 - @ManyToOne
 - @ManyToMany
- El parámetro *cascade* indica el tipo de operaciones en cascada a realizar con las entidades relacionadas (PERSIST, MERGE, REMOVE, REFRESH, DETACH)

- El parámetro *fetch* (EAGER, LAZY) indica la política de carga de las entidades relacionadas: inmediatamente (por defecto) o perezosa
- Si la relación es bidireccional, el parámetro *mappedBy* indica la relación en la otra entidad

```
// En clase Venta:  
@OneToMany(cascade=ALL, fetch=EAGER, mappedBy="num_venta")  
List<Item> items;  
  
// En clase Item:  
@ManyToOne  
Venta venta;
```

Iniciación de JPA

- Es necesario obtener un *EntityManager* para trabajar con JPA
- Este se obtiene a partir de un *EntityManagerFactory*
- Conviene nuevamente usar un singleton para hacer disponible el *EntityManager*

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("ventasPU");  
EntityManager em = emf.createEntityManager();
```



```
public class GestorPersistencia {
    EntityManagerFactory emf;
    EntityManager em;

    static GestorPersistencia instancia = null;

    private GestorPersistencia() {
        emf = Persistence.createEntityManagerFactory("ventasPU");
        em = emf.createEntityManager();
    }

    public EntityManager getEntityManager() {
        return em;
    }

    public static void crearConexion() throws ErrorConexionBD {
        if (instancia == null) {
            instancia = new GestorPersistencia();
        }
    }

    public static GestorPersistencia instancia() {
        return instancia;
    }

    public static void desconectar() {
        if (instancia != null) {
            instancia.em.getTransaction().begin();
            instancia.em.createNativeQuery("shutdown").executeUpdate();
            instancia.em.getTransaction().commit();

            instancia.em.close();
            instancia.emf.close();
            instancia = null;
        }
    }
}
```

Guardar un objeto nuevo

- La operación *persist()* permite añadir un objeto nuevo a la base de datos
- Si se han activado las operaciones en cascada, los objetos relacionados se guardan también

```
Venta v = new Venta(1);

try {
    em.getTransaction().begin();
    em.persist(v);
    em.getTransaction().commit();
}
catch(EntityExistsException e) {
    if (em.getTransaction().isActive())
        em.getTransaction().rollback();
}
```

Recuperación de un objeto

- La recuperación de un objeto a partir de su clave primaria se realiza con la operación *find()*
- Una vez en memoria, puede accederse a los objetos relacionados con independencia de la política de carga de estos
- Un objeto en memoria puede ser refrescado nuevamente de la base de datos con *refresh()*

```
Venta v = em.find(Venta.class, 1);  
  
for (Item i: v.getItems()) {  
    System.out.println(  
        Integer.toString(i.getCantidad()) + " " +  
        i.getDescripcion() + " " +  
        Float.toString(i.getImporteUnidad())  
    );  
}
```

Actualización de un objeto

- Para actualizar un objeto basta con hacerlo en el contexto de una transacción
- También se puede modificar y salvar con *merge()*

```
Venta v = em.find(Venta.class, 1);  
  
em.getTransaction().begin();  
v.anadirItem(new Item(1, "Aceite cadena", 10));  
em.getTransaction().commit();
```

```
Venta v = em.find(Venta.class, 1);  
v.anadirItem(new Item(1, "Aceite cadena", 10));  
  
em.getTransaction().begin();  
em.merge(v);  
em.getTransaction().commit();
```

Borrado en la base de datos

- El borrado de objetos se realiza mediante la operación *remove()*

```
Venta v = em.find(Venta.class, 1);  
  
em.getTransaction().begin();  
em.remove(v);  
em.getTransaction().commit();
```

Consultas en la base de datos

- JPA incluye un lenguaje flexible de consulta similar a SQL

```
TypedQuery<Venta> ventas = em.createQuery("select v from Venta v, Venta.class");  
Collection<Venta> = ventas.getResultList();
```

```
TypedQuery<Venta> ventas = em.createQuery(  
    "select v from Venta v where v.num=:num", Venta.class  
).setParameter("num", 5);
```

```
TypedQuery<Venta> ventas = em.createQuery(  
    "select v from Venta v where v.fecha between :f1 and :f2", Venta.class  
).setParameter("f1", fecha1, TemporalType.TIMESTAMP)  
    .setParameter("f2", fecha2, TemporalType.TIMESTAMP);
```

JPA y DAOs

- Con JPA el uso de DAOs no tiene la misma importancia, aunque puede mantenerse

```
public class DAOVenta {
    static DAOVenta instancia = null;

    SimpleDateFormat sdf;

    private DAOVenta() {
        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    }

    public void crear(Venta v) throws ErrorCreacionVenta {
        EntityManager em = GestorPersistencia.instancia().getEntityManager();

        try {
            em.getTransaction().begin();
            em.persist(v);
            em.getTransaction().commit();
        }
        catch(EntityExistsException e) {
            if (em.getTransaction().isActive())
                em.getTransaction().rollback();
            throw new ErrorCreacionVenta();
        }
    }
}
```

```
public Venta buscarPorNum(int num) {
    EntityManager em = GestorPersistencia.instancia().getEntityManager();

    return em.find(Venta.class, num);
}

public void actualizar(Venta v) throws ErrorActualizacionVenta {
    EntityManager em = GestorPersistencia.instancia().getEntityManager();

    try {
        em.getTransaction().begin();
        em.merge(v);
        em.getTransaction().commit();
    }
    catch(EntityExistsException e) {
        em.getTransaction().rollback();
        throw new ErrorActualizacionVenta();
    }
}

public void borrar(Venta v) throws ErrorBorradoVenta {
    EntityManager em = GestorPersistencia.instancia().getEntityManager();

    try {
        em.getTransaction().begin();
        em.remove(v);
        em.getTransaction().commit();
    }
    catch(EntityExistsException e) {
        em.getTransaction().rollback();
        throw new ErrorBorradoVenta();
    }
}

public static DAOVenta instancia() {
    if (instancia == null) {
        instancia = new DAOVenta();
    }

    return instancia;
}
}
```


Referencias de JPA

- Manual y referencia: www.objectdb.com
- Tutorial de Java EE 6:
<http://download.oracle.com/javaee/6/tutorial/doc/bnbpz.html>