3.2. Creación de un proyecto JPA en Eclipse contra una BD MySQL.

Índice

Introducción

Objetivos

En esta unidad perseguimos los siguientes objetivos:

- Desarrollar aplicaciones Java que accedan a bases de datos relacionales, apoyándose en algún *Framework* ORM.
- Configurar un proyecto Eclipse para trabajar con el *Framework* ORM JPA.
- Interactuar con el *Framework* JPA dentro del proyecto, utilizando las clases y métodos que JPA pone a disposición de sus usuarios.

Configuración del proyecto JPA

La perspectiva Database Development de Eclipse

Eclipse tiene una perspectiva denominada *Database Development*que te permitirá configurar una conexión con una base de datos.

Aunque no es un paso absolutamente necesario, resulta muy útil porque te permitirá inspeccionar la estructura de la base de datos directamente desde Eclipse.

Además, cuando llegue el momento de crear la aplicación JPA, Eclipse se basará en la información suministrada por esta conexión para crear automáticamente por nosotros las clases de entidad, ahorrándonos ese trabajo.

El siguiente video muestra los pasos para cambiar a la perspectiva *Database Development* y añadir una conexión para inspeccionar la base de datos *FERRETERIA*.
https://vimeo.com/telefonicaed/review/271423998/50991e5c3b

Introducción a JPA

Ya sabemos que un *Framework* ORM implementa la técnica de Mapeo objeto-relacional, proporcionando una estructura de objetos que representa la base de datos.

Pues bien, se denomina **capa de persistencia** a la capa que encapsula el comportamiento para mantener dichos objetos.

JPA (Java Persistence API) es el *framework* estándar proporcionado por Java Enterprise Edition (Java EE) para la capa de persistencia que implementa el concepto de ORM.

Características de JPA (*Java Persistence API*):

- **Persistencia utilizando POJOs** (https://es.wikipedia.org/wiki/Plain_Old_Java_Object)**:** cualquier clase Java podemos convertirla en una clase de entidad, simplemente agregando anotaciones.
- **No intrusivo:** JPA es una capa separada de los objetos a persistir. Las clases de entidad no requieren de ninguna funcionalidad en particular, ni saber de la existencia del API JPA.
- **Consultas utilizando objetos java:** JPA permite ejecutar consultas expresadas en objetos Java y sus relaciones sin necesidad de utilizar el lenguaje SQL. Las consultas son traducidas por el API JPA en el código SQL equivalente.

- **Configuración simplificada:** muchas de las opciones de configuración de JPA tienen una valor por defecto. Sin embargo, si queremos personalizarlas, es muy fácil hacerlo, ya sea con anotaciones o por medio de de un archivo XML de configuración.
- **Integración con la especificación Java EE:** JPA se sitúa, dentro de las aplicaciones Java, en una capa independiente del resto, por lo que se integra perfectamente con el resto de capas de la aplicación sin alterarlas.

JPA **no es una herramienta completa, sino que puede ser considerada como una especificación que necesita un proveedor que la implemente**. En este sentido, Eclipse nos asiste a la hora de configurar un proyecto JPA para descargar de internet una implementación.

Son implementaciones de JPA: EclipseLink, Apache OpenJPA, Toplink, Hibernate, etc.

En el siguiente apartado crearás un proyecto JPA y podrás comprobar cómo Eclipse te permite descargar una implementación.

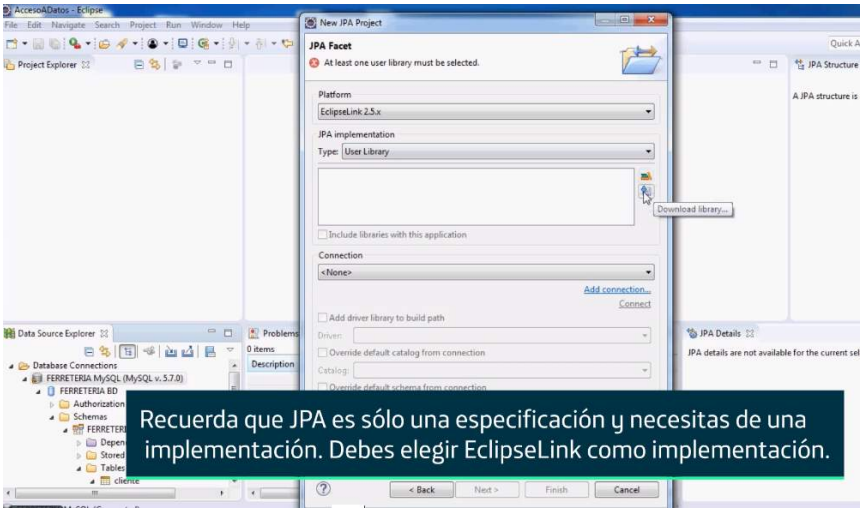
Configurar un proyecto para trabajar con JPA

En el siguiente apartado, prepararás un proyecto Java perfectamente configurado para trabajar con JPA.

Eclipse cuenta con la perspectiva JPA para este tipo de aplicaciones. Si sigues los pasos que verás en el vídeo, podrás crear un proyecto de tipo *JPA Project* y crear las clases de entidad automáticamente. Lo harás con ayuda de la información suministrada por la conexión a la base de datos *FERRETERIA* que dejaste preparada en el anterior vídeo.
https://vimeo.com/telefonicaed/review/271424014/d16ab5ff4b

Vamos a recordar **tres pasos importantes** que has llevado a cabo **para configurar tu proyecto JPA**:

1.



Elección de EclipseLink.

Como comentamos anteriormente, JPA necesita de un fabricante que implemente su especificación. En el paso que ves en la imagen hemos escogido *EclipseLink* como implementación, pero no tenemos su librería descargada en nuestro equipo, por lo tanto, hemos pulsado el botón *Download Library* para descargarlo de internet. El software descargado quedará situado dentro de una carpeta del espacio de trabajo; puedes utilizar el explorador de Windows para comprobarlo.

El botón situado justo encima de *Download Library*, representado por una pila de libros, es el botón*Manage Libraries* y se utiliza para crear una nueva librería o editar las existentes. Será útil en caso de que tengamos el software ya descargado en alguna ubicación de nuestro sistema de archivos. Pero, para nuestro ejemplo, no lo hemos utilizado.

2.

Proceso de descarga de EclipseLink.

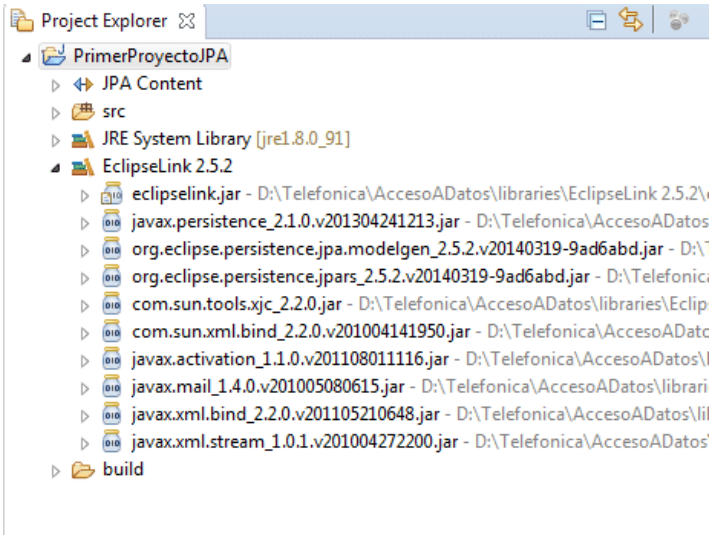
Una vez terminado el proceso de descarga, la nueva librería aparecerá dentro del apartado *JPA Implementation*. En los siguientes proyectos JPA que crees dentro del mismo espacio de trabajo ya aparecerá la librería y no será necesario repetir el paso anterior: sólo dejaremos marcada o desmarcada la casilla de verificación, en función de si deseamos utilizarla o no.

3.

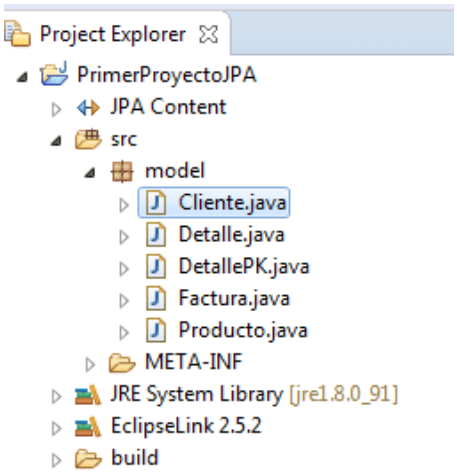
Recuerda que pudiste crear automáticamente las clases de entidad gracias a que antes dejaste abierta en Eclipse una conexión a la base de datos *FERRETERIA*, que fue utilizada por el sistema como guía para desarrollar el código de manera automática.

La estructura del nuevo proyecto

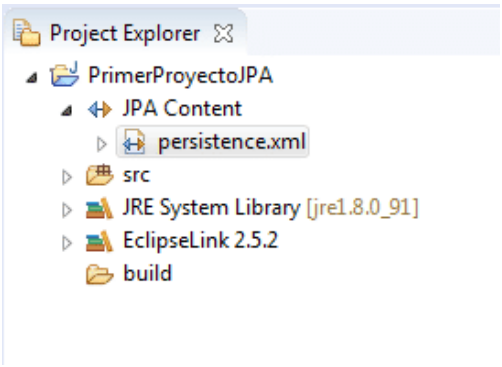
Estructura del proyecto JPA:



Nuestro proyecto no es más que un proyecto de consola que tiene agregadas las librerías y características de un proyecto JPA. En primer lugar, además de las librerías estándar del JRE, cuenta con las **librerías de EclipseLink**. Además, puedes ver junto a cada archivo JAR la ubicación dentro del sistema de archivos en la que está situado.



Las clases de entidad quedan situadas en un paquete denominado ***model***.



Los proyectos JPA cuentan con un importante archivo de configuración denominado ***persistence.xml*** que estudiaremos más detenidamente en el siguiente apartado.

IMPORTANTE: Tu nuevo proyecto está pensado para trabajar con una base de datos MySQL, así que no olvides importar el driver de MySQL en el proyecto como aprendiste en la unidad anterior (clic derecho en el nombre del proyecto / Properties / Java Build Path / Add External jars).

El fichero persistence.xml

El archivo ***persistence.xml*** contiene la configuración de las denominadas ***Unidades de persistencia***, un concepto de vital importancia y que aclararemos en este apartado.

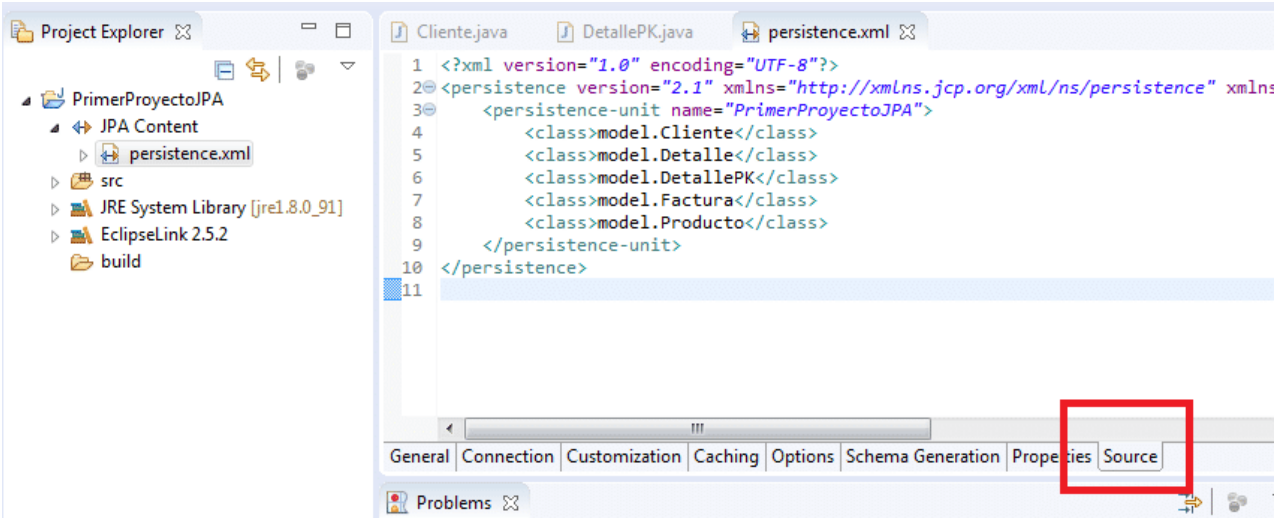
Cada **unidad de persistencia** encierra la configuración de conexión con una base de datos determinada.

Una unidad de persistencia va encerrada entre las etiquetas:

```
<persistence-unit name="nombreUnidadPersistencia">
  .....
</persistence-unit>
```

Donde *nombreUnidadPersistencia* es un identificador de vital importancia para hacer referencia a una conexión u otra dentro del código de nuestra aplicación.

Si una misma aplicación JPA debe acceder a varias bases de datos, tendrá que existir una unidad de persistencia para cada una de ellas.

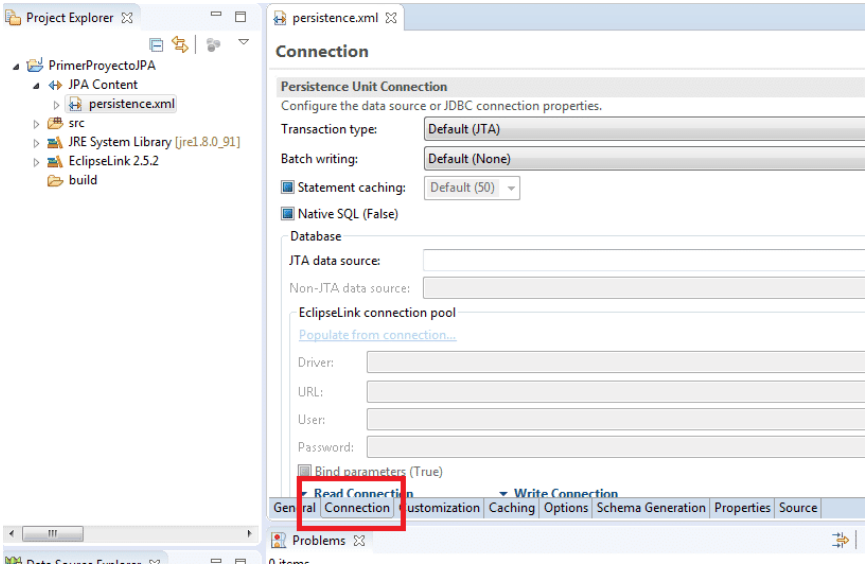


Utiliza el explorador de proyectos para abrir el archivo *persistence.xml* y abre la pestaña *Source* para ver la vista de código XML. Observa que, por defecto, Eclipse ha dado nombre a la unidad de persistencia utilizando el nombre del proyecto. Puedes dejarlo así o cambiarlo, si lo deseas.

Pero todavía hay algo importante que falta por añadir a la unidad de persistencia: los datos de conexión con la base de datos *FERRETERIA*, el equivalente a la cadena de conexión que utilizábamos en JDBC.

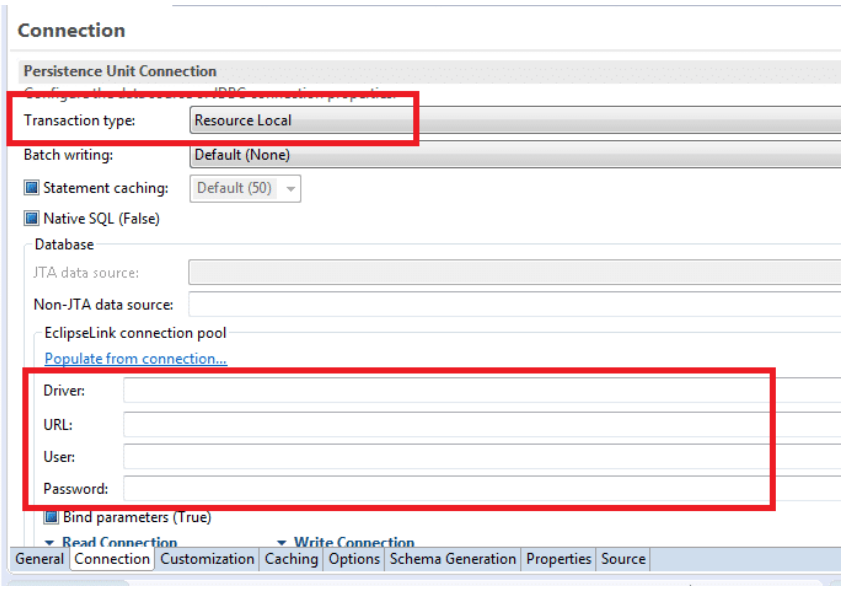
Podríamos llevar a cabo esta tarea a mano, pero **vamos a volver a utilizar los recursos que nos presta Eclipse para ahorrarnos trabajo**. Realiza los pasos que te indicamos a continuación:

1.



Continúa manteniendo abierto el archivo *persistence.xml*, pero esta vez en la ficha *Connection*.

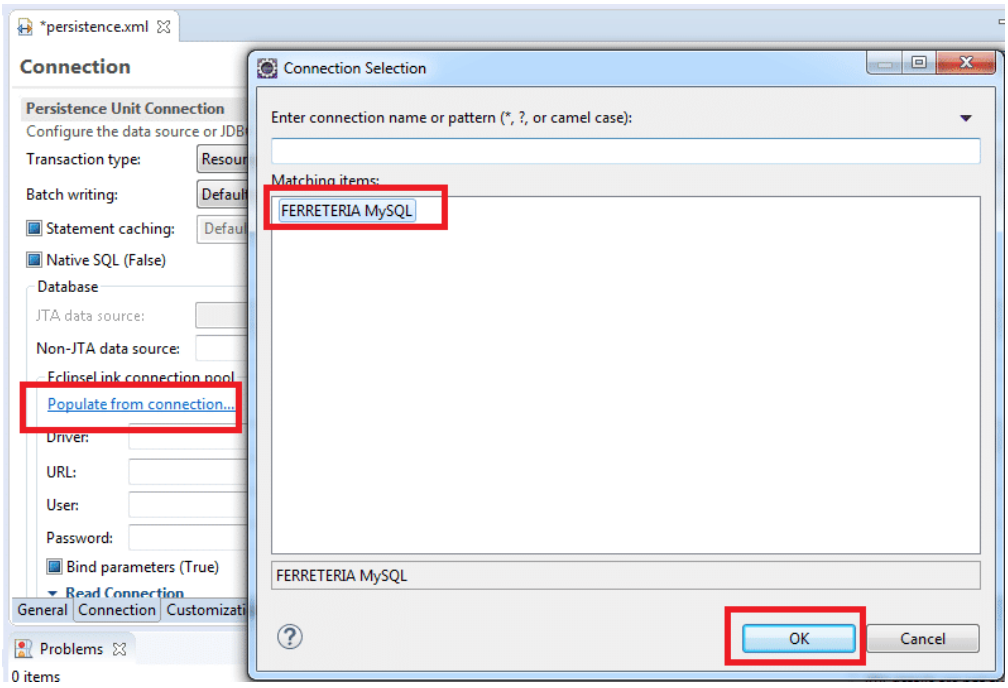
2.



Selecciona *Resource Local* en la lista desplegable *Transaction type* y verás cómo se habilitan las cajas de texto para especificar los valores de *Driver*, *URL*, *User* y *Password*.

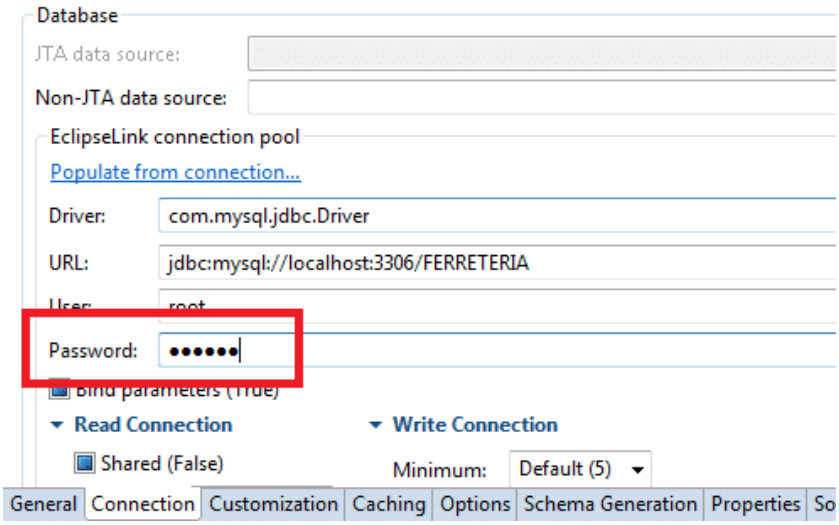
Eclipse utilizará la información de la perspectiva *Database Development* para localizar las conexiones a bases de datos abiertas.

3.



Haz clic en el enlace *Populate from connection.* Selecciona la conexión *FERRETERIA MySQL* y pulsa *OK*.

4.



Eclipse ha completado todos los datos de conexión menos el *Password*, que debes rellenarlo tú con la contraseña que especificaste cuando instalaste MySQL.

El archivo persistence.xml ya ha quedado perfectamente configurado para comenzar a trabajar. Éste es su contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
	<persistence-unit name="PrimerProyectoJPA" transaction-
type="RESOURCE_LOCAL">
		<class>model.Cliente</class>
		<class>model.Detalle</class>
		<class>model.DetallePK</class>
		<class>model.Factura</class>
		<class>model.Producto</class>
		<properties>
			<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/FERRETERIA"/>
			<property name="javax.persistence.jdbc.user"
value="root"/>
			<property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
			<property name="javax.persistence.jdbc.password"
value="amelia"/>
		</properties>
	</persistence-unit>
</persistence>
```

Hasta ahora, todo lo que has hecho han sido labores de configuración del proyecto. En los siguientes apartados escribirás el código de tus aplicaciones JPA.

Aplicación práctica

Inventario de la FERRETERIA

Ahora es el momento de escribir el código necesario para lograr que nuestra aplicación sea capaz de mostrar al usuario el inventario de *FERRETERIA*, es decir, el listado de artículos disponibles.

Lo que realmente hará nuestra aplicación es obtener una colección de objetos de la clase *Producto*, uno por cada fila existente en la tabla *PRODUCTO* de la base de datos.

En esta tarea entrará en juego la clase de entidad *Producto*, por lo que, antes de nada, vamos a revisarla más exhaustivamente.

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.List;

@Entity
@NamedQuery(name="Producto.findAll", query="SELECT p FROM Producto p")
public class Producto implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	private String codigo;

	private String descripcion;

	private float minimo;

	private float precio;

	private float stock;

	//bi-directional many-to-one association to Detalle
	@OneToMany(mappedBy="producto")
	private List<Detalle> detalles;

	public Producto() {
	}

	public String getCodigo() {
		return this.codigo;
	}

	public void setCodigo(String codigo) {
		this.codigo = codigo;
	}
```

```
	public String getDescripcion() {
		return this.descripcion;
	}

	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}

	public float getMinimo() {
		return this.minimo;
	}

	public void setMinimo(float minimo) {
		this.minimo = minimo;
	}

	public float getPrecio() {
		return this.precio;
	}

	public void setPrecio(float precio) {
		this.precio = precio;
	}

	public float getStock() {
		return this.stock;
	}

	public void setStock(float stock) {
		this.stock = stock;
	}

	public List<Detalle> getDetalles() {
		return this.detalles;
	}

	public void setDetalles(List<Detalle> detalles) {
		this.detalles = detalles;
	}

	public Detalle addDetalle(Detalle detalle) {
		getDetalles().add(detalle);
		detalle.setProducto(this);

		return detalle;
	}

	public Detalle removeDetalle(Detalle detalle) {
		getDetalles().remove(detalle);
		detalle.setProducto(null);

		return detalle;
	}

}
```

Analicemos paso a paso la clase de entidad *Producto* y sus anotaciones:

@Entity

Lo más importante es la anotación *@Entity* que indica que se trata de una clase de entidad. Esta anotación va a menudo acompañada de la anotación *@Table*, que indica el nombre de la tabla física en la base de datos que se relaciona con la clase de entidad.

```
@Entity
@Table(name="producto")
public class Producto implements Serializable {
..
}
```

Cuando el nombre de la tabla física coincide con el nombre de la clase de entidad no es necesario utilizar la anotación *@Table*.

Por convención, las clases de entidad deberían ser nombradas con un sustantivo en singular. Ejemplo: para una tabla física llamada *Alumnos* en la base de datos, la clase de entidad debería llamarse *Alumno*.

@NamedQuery(name="Producto.findAll", query="SELECT p FROM Producto p")

Esta anotación define lo que se llama una **consulta con nombre**. Bajo el identificador *Producto.findAll* podemos hacer referencia a la consulta *SELECT p FROM Producto p*.

La consulta está definida en el lenguaje JPQL (*Java Persistence Query Language*), lenguaje de consulta orientado a objetos que forma parte de la especificación JPA y que estudiaremos más detenidamente en otro apartado.

@Id

Esta anotación se coloca delante de un atributo para indicar que se trata de la clave principal. En ocasiones va acompañada de la anotación *@GeneratedValued* para especificar la forma en la que la clave principal será generada.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int idCliente;
```

En este ejemplo estamos indicando que el atributo *idCliente* es la clave principal y que queremos que sea de tipo identidad, es decir, que se vaya auto generando de manera incremental.

@OneToMany(mappedBy="producto")

Esta anotación indica que existe una asociación de tipo *uno a muchos* entre la tabla producto y la tabla detalle.

//bi-directional many-to-one association to Detalle
@OneToMany(mappedBy="producto")
private List<Detalle> detalles;

El atributo *detalles* es una colección de elementos de tipo *Detalle* que contendrá para cada objeto *Producto* tantos objetos *Detalle* como ventas existan para dicho producto.

Este tipo de anotaciones pueden ser:
@OneToMany
@ManyToOne
@OneToOne

public Detalle addDetalle(Detalle detalle) {...}

Puesto que cada objeto *Producto* puede estar asociado a varias ventas (objetos *Detalle*), este método permite añadir un nuevo objeto *Detalle*.

public Detalle removeDetalle(Detalle detalle) {...}

Puesto que cada objeto *Producto* puede estar asociado a varias ventas (objetos *Detalle*), este método permite eliminar una venta (*Detalle*).

Llegó el momento de crear la clase *Principal* y ejecutar el proyecto.

¡Ojo! No la crees en el paquete *model*, déjala fuera, ya que el paquete *model* está pensado para contener sólo las clases de entidad.

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
```

```
import model.Producto;

public class Principal {

	public static void main(String[] args) {

		listado();

	}

	public static void listado() {
		EntityManagerFactory factoria =
Persistence.createEntityManagerFactory("PrimerProyectoJPA");
		EntityManager em = factoria.createEntityManager();
		TypedQuery<Producto> query =
em.createNamedQuery("Producto.findAll", Producto.class);
		List<Producto> productos = query.getResultList();

		for (Producto p : productos) {
			System.out.println(p.getCodigo() + " - " +
p.getDescripcion() +
			  " - " + p.getPrecio() + " € - " +  p.getStock() + "
unidades.");
		}
	}
}
```

Si ya has ejecutado el programa, habrás obtenido el listado de artículos del almacén. Ahora estudiaremos las nuevas clases Java que hemos empleado.

Clases Java

El paquete *javax.persistence:*

Las clases que forman parte de la librería que implementa JPA (*EclipseLink*, en nuestro caso) se encuentran situadas en el paquete *javax.persistence*. Vamos a estudiar las clases que hemos utilizado en el ejemplo anterior.

javax.persistence.Persistence
Utilizamos la clase *Persistence* para invocar a su método estático *createEntityManagerFactory()*pasando como argumento el nombre de la unidad de persistencia. Esto nos permitirá obtener un objeto de la clase *EntityManagerFactory*, imprescindible para interactuar con la base de datos a través de la unidad de persistencia.

javax.persistence.EntityManagerFactory
Necesitamos nuestro objeto *EntityManagerFactory* para obtener el *EntityManager* a través de la llamada al método *createEntityManager()*.

javax.persistence.EntityManager
El objeto *EntityManager* actúa como administrador de las clases de entidad, permitiéndonos efectuar todas las operaciones CRUD (*Create*, *Read*, *Update and Delete*) en la base de datos a través de las clases de entidad.

javax.persistence.TypedQuery
TypeQuery es una Interfaz utilizada para controlar la ejecución de consultas tipificadas, es decir, podemos especificar el tipo de objetos que devolverá la consulta.

Sobre el objeto *TypeQuery* se podrá invocar al método *getResultList()*, que retornará el conjunto de objetos resultado de la ejecución de la consulta; para nuestro ejemplo, todos los productos de la *FERRETERIA*.

List<Producto> productos = query.getResultList();

Facturas con sus detalles

Vamos a completar el programa anterior para que, además de mostrar el inventario, muestre un listado de facturas, incluyendo debajo de cada una sus líneas de detalle.

Ten en cuenta que cada objeto de la clase *Factura* es un compuesto que lleva, además de los datos propios de la factura, el objeto *Cliente* asociado y la colección de detalles (objetos *Detalle*). Y que por cada objeto *Detalle* podemos, además, acceder a los datos del objeto *Producto* relacionado.

En conclusión, **obteniendo la relación de facturas obtenemos el resto de la información asociada a través de todo el modelo de objetos**. El programa modificado quedaría así:

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

import model.Detalle;
import model.Factura;
import model.Producto;

public class Principal {

	public static void main(String[] args) {

		inventario();
```

```
                System.out.println();
                facturasDetalles();
        }

        public static void inventario() {
                EntityManagerFactory factoria =
Persistence.createEntityManagerFactory("PrimerProyectoJPA");
                EntityManager em = factoria.createEntityManager();
                TypedQuery<Producto> query =
em.createNamedQuery("Producto.findAll", Producto.class);
                List<Producto> productos = query.getResultList();

                for (Producto p : productos) {
                        System.out.println(p.getCodigo() + " - " +
p.getDescripcion() + " - " + p.getPrecio() + " € - " +  p.getStock() + "
unidades.");
                }
        }

        public static void facturasDetalles() {
                EntityManagerFactory factoria =
Persistence.createEntityManagerFactory("PrimerProyectoJPA");
                EntityManager em = factoria.createEntityManager();
                TypedQuery<Factura> query =
em.createNamedQuery("Factura.findAll", Factura.class);
                List<Factura> facturas = query.getResultList();

                for (Factura f : facturas) {
                        System.out.println(f.getNumero() + " - " + f.getFecha() +
" - " + f.getCliente().getNombre());
                        for (Detalle d : f.getDetalles()) {
                                Producto p = d.getProducto();
                                System.out.println("    " + p.getCodigo() + " - "
+ p.getDescripcion() + " - " + d.getPrecio() + "€ " + d.getUnidades());
                        }
                }
        }
}
```

Para cada factura estamos recorriendo sus líneas de detalle a partir del método *getDetalles()*. De la misma manera, para cada detalle accedemos al objeto *Producto* asociado y lo guardamos en un objeto cuya referencia se encuentra en la variable *p*.

Si has ejecutado el programa, es muy posible que arroje algún error y estés viendo la traza del error en la consola de Eclipse.

IMPORTANTE: La herramienta de generación automática de clases de entidad que nos brinda Eclipse es muy útil, pero puede cometer pequeños fallos a la hora de establecer los tipos de datos de las propiedades. En este caso, el programa no funciona y provoca un error de ejecución. Al examinar la traza del error podemos encontrar el siguiente texto:

Exception Description: The object [true], of class [class java.lang.Boolean], from mapping [org.eclipse.persistence.mappings.DirectToFieldMapping[pagado-->FACTURA.PAGADO]] with descriptor [RelationalDescriptor(model.Factura --> [DatabaseTable(FACTURA)])], could not be converted to [class java.lang.Byte].

El conflicto lo está provocando la clase *Factura* porque el atributo *pagado* está declarado como tipo *byte* y, sin embargo, el campo *PAGADO* de la base de datos fue declarado como tipo *BOOL* (aunque internamente sea un número entero de un solo dígito). Solucionamos el problema declarando el atributo *pagado* en la clase de entidad como tipo *boolean*.

Así es como queda el código Java de la entidad *Factura* tras cambiar el tipo de dato del atributo *pagado* y sus métodos *get/set* asociados.

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;
import java.util.List;

@Entity
@NamedQuery(name="Factura.findAll", query="SELECT f FROM Factura f")
public class Factura implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	private int numero;

	@Temporal(TemporalType.DATE)
	private Date fecha;

	private boolean pagado;

	//bi-directional many-to-one association to Detalle
	@OneToMany(mappedBy="factura")
	private List<Detalle> detalles;

	//bi-directional many-to-one association to Cliente
	@ManyToOne
	@JoinColumn(name="NIF")
	private Cliente cliente;

	public Factura() {
	}

	public int getNumero() {
		return this.numero;
	}

	public void setNumero(int numero) {
		this.numero = numero;
	}
```

```
	public Date getFecha() {
		return this.fecha;
	}

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

	public boolean getPagado() {
		return this.pagado;
	}

	public void setPagado(boolean pagado) {
		this.pagado = pagado;
	}

	public List<Detalle> getDetalles() {
		return this.detalles;
	}

	public void setDetalles(List<Detalle> detalles) {
		this.detalles = detalles;
	}

	public Detalle addDetalle(Detalle detalle) {
		getDetalles().add(detalle);
		detalle.setFactura(this);

		return detalle;
	}

	public Detalle removeDetalle(Detalle detalle) {
		getDetalles().remove(detalle);
		detalle.setFactura(null);

		return detalle;
	}

	public Cliente getCliente() {
		return this.cliente;
	}

	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}

}
```

Nuevas anotaciones

Estas son algunas anotaciones nuevas que han aparecido en la clase de entidad *Factura:*

@Temporal(TemporalType.DATE)

Esta anotación está colocada delante del atributo *fecha* para indicar cómo debe ser convertido el campo *FECHA* de la base de datos en el atributo *fecha* de la clase de entidad. Admite uno de los siguientes argumentos:

@Temporal(TemporalType.DATE)
Ignora la hora, quedando el campo acotado sólo a la fecha.

@Temporal(TemporalType.TIME)
Ignora la fecha, quedando el campo acotado sólo a la hora.

@Temporal(TemporalType.TIMESTAMP)
Tiene en cuanta la fecha y la hora.

@ManyToOne

Esta anotación establece una asociación de *muchos a uno*. En nuestro programa está registrando una asociación de muchas facturas a un solo cliente. Cada uno de los objetos *Factura* se relaciona con un solo objeto *Cliente* y lo hace a través del campo *NIF*.

```
//bi-directional many-to-one association to Cliente
@ManyToOne
@JoinColumn(name="NIF")
private Cliente cliente;
```

A partir de un objeto *Factura* tenemos disponible toda la información del cliente al que pertenece, a partir del atributo *cliente*.

@JoinColumn(name="NIF")

Acompaña una anotación de tipo *@ManyToOne* para indicar cuál es la columna que establece la asociación.

Las clases de entidad Detalle y DetallePK

La clave principal de la tabla *DETALLE* está compuesta por la unión de los campos *NUMERO* (número de factura) y *CODIGO* (código de producto).

Esta situación de clave primaria compuesta es un tanto especial dentro de las clases de entidad.

1. Comenzaremos analizando la clase *DETALLE*. Recordemos cómo está montado el código de la clase *Detalle*:

```
package model;

import java.io.Serializable;
import javax.persistence.*;

@Entity
@NamedQuery(name="Detalle.findAll", query="SELECT d FROM Detalle d")
public class Detalle implements Serializable {
	private static final long serialVersionUID = 1L;

	@EmbeddedId
	private DetallePK id;

	private float precio;

	private int unidades;

	//bi-directional many-to-one association to Factura
	@ManyToOne
	@JoinColumn(name="NUMERO")
	private Factura factura;

	//bi-directional many-to-one association to Producto
	@ManyToOne
	@JoinColumn(name="CODIGO")
	private Producto producto;

	public Detalle() {
	}

	public DetallePK getId() {
		return this.id;
	}

	public void setId(DetallePK id) {
		this.id = id;
	}

	public float getPrecio() {
		return this.precio;
	}

	public void setPrecio(float precio) {
		this.precio = precio;
	}
```

```
	public int getUnidades() {
		return this.unidades;
	}

	public void setUnidades(int unidades) {
		this.unidades = unidades;
	}

	public Factura getFactura() {
		return this.factura;
	}

	public void setFactura(Factura factura) {
		this.factura = factura;
	}

	public Producto getProducto() {
		return this.producto;
	}

	public void setProducto(Producto producto) {
		this.producto = producto;
	}

}
```

Ahora, vamos a analizar las partes más relevantes:

@EmbeddedId

Esta anotación, al igual que la anotación *@Id*, está indicando que el atributo que sigue es una clave primaria, pero además indica que no se trata de un dato elemental, sino de un objeto definido en otra clase, en este caso la clase *DetallePK*.

@EmbeddedId
private DetallePK id;

El atributo o propiedad *id* es un objeto de la clase *DetallePK*.

@ManyToOne

//bi-directional many-to-one association to Factura
@ManyToOne
@JoinColumn(name="NUMERO")
private Factura factura;

//bi-directional many-to-one association to Producto
@ManyToOne
@JoinColumn(name="CODIGO")
private Producto producto;

Cada objeto de la clase *Detalle* es un compuesto formado por la factura que incluye dicha línea de detalle, y el producto que se vende, además del resto de los atributos.

2. Vamos a analizar ahora el código de la clase *DetallePK*. Observa el código de la clase *DetallePK*:

```
package model;

import java.io.Serializable;
import javax.persistence.*;

@Embeddable
public class DetallePK implements Serializable {
	//default serial version id, required for serializable classes.
	private static final long serialVersionUID = 1L;

	@Column(insertable=false, updatable=false)
	private int numero;

	@Column(insertable=false, updatable=false)
	private String codigo;

	public DetallePK() {
	}
	public int getNumero() {
		return this.numero;
	}
	public void setNumero(int numero) {
		this.numero = numero;
	}
	public String getCodigo() {
		return this.codigo;
	}
	public void setCodigo(String codigo) {
		this.codigo = codigo;
	}

	public boolean equals(Object other) {
		if (this == other) {
			return true;
		}
		if (!(other instanceof DetallePK)) {
			return false;
		}
		DetallePK castOther = (DetallePK)other;
		return
			(this.numero == castOther.numero)
			&& this.codigo.equals(castOther.codigo);
	}

	public int hashCode() {
		final int prime = 31;
		int hash = 17;
		hash = hash * prime + this.numero;
		hash = hash * prime + this.codigo.hashCode();
```

```
                return hash;
        }
}
```

Vamos a comentar las partes más importantes:

@Embeddable

Esta anotación va delante de una clase que tiene como función embeber una clave primaria compuesta.

@Embeddable
public class DetallePK implements Serializable { ... }

Cada objeto de la clase *DetallePK* es una clave primaria embebida, que servirá de componente a la clase *Detalle*.

@Column(insertable=false, updatable=false)

La anotación *@Column* con los atributos *insertable* y *updatable y* con los valores *false*, provoca que el atributo que acompaña no sea tenido en cuenta en el momento en que el *framework* ORM autogenere las sentencias SQL *INSERT* y *UPDATE*, cuando haya que persistir el objeto.

@Column(insertable=false, updatable=false)
private int numero;

@Column(insertable=false, updatable=false)
private String codigo;

Los objetos de la clase *DetallePK* no están pensados para persistirse, sino para servir de componente dentro de un objeto *Detalle* que sí podría ser persistido (guardado físicamente en la base de datos). Por este motivo, las dos propiedades o atributos de la clase *DetallePK* llevan esta anotación.

Los métodos equals() y hashcode()

Observa que también se han sobrescrito los métodos *equals()* y *hashcode()*, métodos que vienen heredados de la superclase *Object*.

Recuerda que el método *hashcode()* devuelve el código *hash* asociado al objeto. El código *hash*es un identificador de 32 bits que identifica al objeto. Es posible sobrescribir el método *hashcode()* para personalizar la generación del identificador.

El método *equals()* compara el objeto actual con otro objeto y devuelve *true* si son iguales; *equals()* considera que dos métodos son iguales si devuelven el mismo código *hash*.

En la clase *DetallePK* se sobrescriben estos métodos para controlar la no existencia de clave primaria duplicada a la hora de persistir un objeto de la clase *Detalle*.

Si lo necesitas, puedes repasar los conceptos sobre este tema volviendo a la lección *6.2. Generalización / especialización: herencia* de la asignatura de programación; en concreto, en el apartado *Los métodos hashcode y equals*.

Buscar información de un cliente

A continuación, veremos cómo obtener un objeto *Cliente* a través del método *find()* del *EntityManager*.

Recuerda que para trabajar con JPA necesitas un objeto de la clase *EntityManager*, que será el administrador de esa base de datos virtual orientada a objetos que se crea a partir de la base de datos real.

En nuestro caso, esa base de datos virtual orientada a objetos está formada por objetos de las clases *Cliente*, *Factura*, *Detalle* y *Producto*.

Nuestro *EntityManager* cuenta con un método *find()* que nos permite obtener un objeto *Cliente*, *Factura*, *Detalle* o *Producto* a partir de la clave primaria.

```
Cliente cli = em.find(Cliente.class, "43434343A");
```

Esta simple línea de código te permitirá obtener el objeto *Cliente* con *NIF = "43434343A"*. Si no existe ningún cliente con el NIF especificado, el método *find()* devuelve *null*.

Nuestro programa va creciendo. Es hora de mejorar la interfaz de usuario añadiendo un menú de *opciones*, para que se pueda indicar la opción a realizar en tiempo de ejecución.

```
GESTIÓN FERRETERÍA
------------------------------
1. Inventario
2. Consultar todas las facturas
3. Información de un cliente
4. Añadir nuevo cliente
5. Añadir nueva factura
6. Listado de productos para reponer
7. Terminar programa
¿Qué opción eliges?
```

Este menú permitirá al usuario seleccionar la opción deseada o terminar escribiendo la opción 7.

Otro cambio que vamos a introducir es que sólo obtendremos el *EntityManager* una vez que estemos en el método *main* y no lo volveremos a obtener cada vez que invoquemos a cada uno de los métodos. Para ello, hemos tenido que declarar las variables *factoria* y *em* como propiedades privadas de clase para que estén disponibles en todos los métodos.

Por ahora, nuestro programa queda así:

```
import java.util.List;
import java.util.Scanner;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

import model.Cliente;
import model.Detalle;
import model.Factura;
import model.Producto;

public class Principal {
	private static Scanner lector;
	private static EntityManagerFactory factoria;
	private static EntityManager em;

	public static void main(String[] args) {

		factoria =
Persistence.createEntityManagerFactory("PrimerProyectoJPA");
		em = factoria.createEntityManager();

		lector = new Scanner(System.in);
		String opcion;

		do {
			System.out.println();
			System.out.println();
			System.out.println("GESTIÓN FERRETERÍA");
			System.out.println("-----------------------------");
			System.out.println("1. Inventario");
			System.out.println("2. Consultar todas las facturas");
			System.out.println("3. Información de un cliente");
			System.out.println("4. Añadir nuevo cliente");
			System.out.println("5. Añadir nueva factura");
			System.out.println("6. Listado de productos para
reponer");
			System.out.println("7. Terminar programa");
			System.out.println("¿Qué opción eliges?");
			opcion = lector.nextLine();

			switch (opcion) {
				case "1": inventario(); break;
				case "2": facturasDetalles(); break;
				case "3": consultaCliente(); break;
				case "4": aniadirCliente(); break;
				case "5": aniadirFactura(); break;
				case "6": listadoReposicion(); break;
```

```
                                case "7": System.out.println("Hasta pronto");
                        }

                } while (!opcion.equals("7"));

                lector.close();
        }

        public static void inventario() {
                TypedQuery<Producto> query =
em.createNamedQuery("Producto.findAll", Producto.class);
                List<Producto> productos = query.getResultList();

                for (Producto p : productos) {
                        System.out.println(p.getCodigo() + " - " +
p.getDescripcion() + " - " + p.getPrecio() + " € - " +  p.getStock() + "
unidades.");
                }
        }

        public static void facturasDetalles() {
                TypedQuery<Factura> query =
em.createNamedQuery("Factura.findAll", Factura.class);
                List<Factura> facturas = query.getResultList();

                for (Factura f : facturas) {
                        System.out.println(f.getNumero() + " - " + f.getFecha() +
" - " + f.getCliente().getNombre());
                        for (Detalle d : f.getDetalles()) {
                                Producto p = d.getProducto();
                                System.out.println("    " + p.getCodigo() + " - "
+ p.getDescripcion() + " - " + d.getPrecio() + "€ " + d.getUnidades());
                        }
                }
        }

        public static void consultaCliente() {
                System.out.println("Escribe NIF del cliente buscado: ");
                String nif = lector.nextLine();
                Cliente cli = em.find(Cliente.class, nif);
                if (cli == null) {
                        System.out.println("No se ha encontrado el cliente con
NIF = " + nif);
                }
                else {
                        System.out.println("Nombre: " + cli.getNombre());
                        System.out.println("Domicilio: " + cli.getDomicilio());
                        System.out.println("Teléfono: " + cli.getTlf());
                        System.out.println("Ciudad: " + cli.getCiudad());
                        System.out.println("FACTURAS: ");
                        for (Factura f : cli.getFacturas()) {
                                System.out.println("  " + f.getNumero() + " " +
f.getFecha());
                        }
                }
        }

        public static void aniadirCliente() {

        }
```

```
        public static void aniadirFactura() {

        }

        public static void listadoReposicion() {

        }
}
```

Tenemos desarrolladas las tres primeras opciones y, para las demás, el método está preparado en vacío para ser implementado más adelante.

Añadir un cliente

Hemos montado la estructura de la aplicación, pero faltan varios métodos por implementar. Veamos ahora la implementación del método *aniadirCliente()*, que añadirá una nueva fila en la tabla *CLIENTE*.

Añadir un nuevo cliente es tan sencillo como construir un nuevo objeto de tipo *Cliente* y luego invocar al método *persist()* del *EntityManager*. Eso sí, lo haremos dentro de una transacción. En primer lugar, se solicitará al usuario que introduzca el NIF del nuevo cliente y se efectuará una búsqueda por medio del método *find()*. Si el cliente existe, informaremos al usuario, y si no existe, le daremos de alta. En este caso, se solicitará al cliente que introduzca por teclado el resto de los datos.

```
public static void aniadirCliente() {

                System.out.println("NIF del nuevo cliente: ");
                String nif = lector.nextLine();
                Cliente cli = em.find(Cliente.class, nif);
                if (cli == null) {
                        cli = new Cliente();
                        cli.setNif(nif);
                        System.out.println("Nombre: ");
                        cli.setNombre(lector.nextLine());
                        System.out.println("Dirección: ");
                        cli.setDomicilio(lector.nextLine());
                        System.out.println("Teléfono: ");
                        cli.setTlf(lector.nextLine());
                        System.out.println("Ciudad: ");
                        cli.setCiudad(lector.nextLine());

                        EntityTransaction et = em.getTransaction();
                        et.begin();
                        em.persist(cli);
                        et.commit();
                }
                else {
                        System.out.println("Ya existe un cliente con NIF: " + nif);
                }

        }
```

Hemos creado un nuevo objeto *Cliente* cuya referencia está en la variable *cli*. Luego, **para persistir el nuevo cliente, hemos utilizado las siguientes líneas de código**:

EntityTransaction et = em.getTransaction();

A través el objeto *EntityManager* (*em*) obtenemos un objeto de tipo *EntityTransaction*, que nos servirá para gestionar las transacciones en las operaciones de actualización de la base de datos.

et.begin();

Inicia la transacción.

em.persist(cli);

Persiste el nuevo cliente en la base de datos.

et.commit();

Finaliza la transacción, realizando los cambios.

También se podría invocar al método *rollback()* para revertir estos cambios.

Las operaciones CRUD con JPA

CRUD (*Create, Read, Update y Delete*) hace referencia a las operaciones básicas que pueden realizarse sobre las filas de una tabla dentro de una base de datos relacional: crear, leer, actualizar y borrar.

Nuestro objeto *EntityManager* (*em* para nuestro programa) te permite realizar las cuatro operaciones CRUD a través de las clases de entidad, utilizando los siguientes métodos:

- ***persist(obj)***: persiste el objeto indicado en el argumento.
- ***find(entityClass, primaryKey)***: busca el valor especificado como *primaryKey* y devuelve el objeto encontrado, cuyo tipo viene especificado en el primer argumento.
- ***merge(obj)***: actualiza la fila de la base de datos, con los valores del objeto especificado como argumento.
- ***delete(obj)***: elimina de la base de datos la fila que corresponde, con el objeto pasado como argumento.

1. Crear nueva fila

```
cli = new Cliente();
cli.setNif("12345678A");
cli.setNombre("PERICO DE LOS PALOTES");
cli.setDomicilio("C/ PERICO, 45");
```

```
cli.setTlf("612345678");
cli.setCiudad("MADRID");

EntityTransaction et = em.getTransaction();
et.begin();
em.persist(cli);
et.commit();
```

2. Leer una fila

```
Cliente cli = em.find(Cliente.class, "43434343A");
System.out.println(cli.getNombre());
```

3. Actualizar una fila

```
Cliente cli = em.find(Cliente.class, "43434343A");
EntityTransaction et = em.getTransaction();
et.begin();
cli.setDomicilio("C/ Nueva, 25");
em.merge(cli);
et.commit();
```

4. Eliminar una fila

```
Cliente cli = em.find(Cliente.class, "43434343A");
EntityTransaction et = em.getTransaction();
et.begin();
em.remove(cli);
et.commit();
```

Java Persistence Query Language (JPQL)

JPQL es un lenguaje con una estructura muy similar a SQL, pero adaptado a la manipulación de objetos de entidad.

La gestión de las consultas JPQL será responsabilidad de un objeto *Query* o de un objeto *TypedQuery*.

Veamos el formato de la sentencia *SELECT* en JPQL:

```
SELECT atributos FROM ClaseEntidad Alias
WHERE criterio
GROUP BY atributos
```

```
HAVING criterio
ORDER BY atributos
```

En el *criterio* se puede utilizar cualquiera de los operadores de SQL.

Hay que tener en cuenta que la consulta no se realiza sobre la base de datos real, sino sobre la base de datos orientada a objetos virtual gestionada por el *EntityManager.*

Esto significa que todos los atributos hacen referencia a las propiedades de las clases de entidad.

Veamos algunos ejemplos:

Ejemplo 1:

```
SELECT cli FROM Cliente cli
```

Selecciona todos los clientes. El identificador *cli* es el alias que hace referencia a cada objeto *Cliente*obtenido.

Ejemplo 2:

```
SELECT cli FROM Cliente cli WHERE cli.ciudad = 'MADRID'
```

Selecciona todos los clientes de *Madrid* y devuelve una colección de objetos *Cliente*.

Ejemplo 3:

```
SELECT cli FROM Cliente cli WHERE cli.nombre LIKE '%PEREZ%'
```

Selecciona todos los clientes que contengan el apellido *PEREZ* en cualquier posición y devuelve una colección de objetos *Cliente*.

Ejemplo 4:

```
SELECT cli.nombre FROM Cliente cli WHERE cli.nombre LIKE '%PEREZ%'
```

Selecciona el nombre de todos los clientes que contengan el apellido *PEREZ* en cualquier posición. Devuelve el resultado en una sola columna y puede recogerse como una colección de objetos *String*.

Ejecutar una consulta JPQL de selección desde Java

Para ejecutar una consulta JPQL de selección desde Java necesitamos obtener un objeto *TypedQuery* a través del método *createQuery()* del *EntityManager*. Luego, debemos invocar al método ***getResultList()*** del objeto *TypedQuery*.

```
TypedQuery<Cliente> query = em.createQuery("SELECT cli FROM Cliente cli WHERE
cli.nombre LIKE '%PEREZ%'", Cliente.class);
List<Cliente> clientes = query.getResultList();
for (Cliente c : clientes) {
        System.out.println(c.getNombre() + " - " + c.getTlf());
}
```

La consulta anterior obtiene una colección de objetos *Cliente*, pero si sólo necesitamos una columna, podríamos obtener una colección de objetos *String*. Observa el siguiente ejemplo:

```
TypedQuery<String> query = em.createQuery("SELECT DISTINCT cli.ciudad FROM
Cliente cli", String.class);
List<String> ciudades = query.getResultList();
for (String c : ciudades) {
        System.out.println(c);
}
```

La consulta devuelve una sola columna, así que puede ser recogida en una colección de tipo *String*.

1. Consulta que devuelve un solo valor

Para ejecutar una consulta que estamos seguros de que sólo devuelve un objeto y no una colección de ellos, utilizamos el método ***getSingleResult()*** en lugar del método *getResultList()*. A continuación vamos a probar con una consulta que devuelve el último número de factura:

```
TypedQuery<Integer> query = em.createQuery("SELECT MAX(f.numero) FROM Factura
f", Integer.class);
Integer maxNumFactura = query.getSingleResult();
System.out.println("Último número de factura: " + maxNumFactura);
```

La consulta devuelve un único valor de tipo *Integer*.

2. Consulta que devuelve una colección de arrays

La mayor parte de las consultas JPQL que ejecutamos devolverán una colección de objetos de un tipo específico: *Cliente*, *Factura*, *Detalle* o *Producto. P*ero cuando especificamos un número determinado de columnas que ya no se corresponde con la estructura de ninguna de las clases de entidad, podemos recoger el resultado como una colección de arrays. Veamos un ejemplo:

```
TypedQuery<Object[]> query = em.createQuery("SELECT pro.codigo, pro.descripcion,
pro.minimo-pro.stock FROM Producto pro WHERE pro.stock < pro.minimo",
Object[].class);
List<Object[]> resultados = query.getResultList();
for (Object[] p : resultados) {
        System.out.println(p[0] + " - " + p[1] + " - " + p[2]);
}
```

Esta consulta selecciona los productos que hay que reponer y devuelve una colección de arrays de tres elementos: *codigo*, *descripcion* y una columna calculada: *minimo-stock*.

3. Consulta de acción

También es posible ejecutar consultas de acción con JPQL (*UPDATE*, *INSERT* o *DELETE*). Para ello, necesitamos obtener un objeto de tipo *Query* y ejecutar la consulta con el método *executeUpdate()*, que devolverá un número entero con la cantidad de objetos o filas afectadas.

```
EntityTransaction et = em.getTransaction();
et.begin();
Query query = em.createQuery("UPDATE Cliente c SET c.domicilio = 'C/ LUNA, 25'
WHERE c.nif='43434343A'");
int afectados = query.executeUpdate();
et.commit();
System.out.println("Objetos afectados: " + afectados);
```

Cambia la dirección del cliente con NIF = "43434343A".

4. Consulta con parámetros

JPQL permite declarar sentencias con parámetros que pueden variar en tiempo de ejecución. Un parámetro se define con una interrogación seguida de un número que identifica la posición del parámetro. El parámetro se establece con el método *setParameter(posición*, *valor)*. Veamos un ejemplo:

```
String sql = "SELECT pro FROM Producto pro WHERE pro.codigo = ?1";
TypedQuery<Producto> query = em.createQuery(sql, Producto.class);
query.setParameter(1, "TOR7");
Producto p = query.getSingleResult();
System.out.println(p.getDescripcion() + " - " + p.getPrecio()+"€");
```

Obtiene el objeto *Producto* que corresponde con el producto con código *TOR7*.

Listado de productos para reponer

A continuación vamos a implementar el método *listadoReposicion()*, que mostrará un listado con los productos pendientes de reposición y las unidades a reponer.

Recuerda que un producto necesita reposición si tiene un valor mayor en el campo *minimo* que en el campo *stock*.

```
public static void listadoReposicion() {
	TypedQuery<Object[]> query = em.createQuery("SELECT pro.codigo,
pro.descripcion, pro.minimo-pro.stock FROM Producto pro WHERE pro.stock <
pro.minimo", Object[].class);
	List<Object[]> resultados = query.getResultList();
	for (Object[] p : resultados) {
		System.out.println(p[0] + " - " + p[1] + " - " + p[2]);
	}
}
```

Este tipo de consulta la estudiamos en el apartado anterior y ahora la hemos integrado en la aplicación final.

Añadir una factura

Vamos a abordar ahora la tarea más laboriosa de nuestro programa: dar de alta una nueva factura implementando el método *aniadirFactura()*.

No es que añadir una nueva factura en la base de datos sea complicado. En realidad, no haremos nada que no hayamos estudiado ya en los apartados anteriores. Resulta laborioso porque implica varias tareas:

- Buscar al cliente al que pertenece al factura.
- Añadir la factura.
- Añadir las líneas de detalle de la factura, lo que implica también localizar los productos que van a venderse.

Veamos el código para el método *aniadirFactura()* y después lo analizaremos paso por paso.

```
public static void aniadirFactura() {
	System.out.println("NIF del cliente: ");
```

```
        String nif = lector.nextLine();
        Cliente cli = em.find(Cliente.class, nif);
        if (cli==null) {
                System.out.println("No existe el cliente con NIF = " + nif);
                System.out.println("Seleccione primero la opción 4");
        }
        else {
                System.out.println("Cliente: " + cli.getNombre());
                TypedQuery<Integer> query = em.createQuery("SELECT MAX(f.numero)
FROM Factura f", Integer.class);
                Integer numFactura = query.getSingleResult() + 1;
                System.out.println("Número de factura: " + numFactura);

                // Construimos objeto factura.
                Factura f = new Factura();
                f.setNumero(numFactura);
                f.setCliente(cli);
                f.setPagado(false);
                f.setFecha(new Date());
                // Añadimos la factura al cliente.
                cli.addFactura(f);

                // Iniciamos la transacción y persistimos la factura.
                EntityTransaction et = em.getTransaction();
                et.begin();
                em.persist(f);

                // Construimos los detalles de la factura.
                String continuar;
                do {
                        System.out.println("Código de artículo: ");
                        String codigo = lector.nextLine();
                        Producto pro = em.find(Producto.class, codigo);
                        if (pro==null) {
                                System.out.println("No se encuentra el producto
con código " + codigo);
                        }
                        else {
                                System.out.println("Descripción: " +
pro.getDescripcion());
                                System.out.println("Precio: " + pro.getPrecio());
                                System.out.println("¿Cuantas unidades desea? ");
                                int unidades = lector.nextInt();
                                lector.nextLine(); // Recoge el retorno de carro.

                                // Creamos el objeto Detalle con su primary key
compuesta.
                                DetallePK pk = new DetallePK();
                                pk.setCodigo(codigo);
                                pk.setNumero(numFactura);
                                Detalle d = new Detalle();
                                d.setId(pk);
                                d.setPrecio(pro.getPrecio());
                                d.setUnidades(unidades);
                                d.setProducto(pro);
                                d.setFactura(f);

                                // Persistimos el detalle.
                                em.persist(d);

                                // Añadimos el detalle a la factura.
```

```
                                f.addDetalle(d);
                        }
                        System.out.println("¿Deseas seguir comprando (S/N)? ");
                        continuar = lector.nextLine();
                } while (continuar.toUpperCase().equals("S"));

                et.commit();
        }
}
```

Vamos a dividir el código en partes para analizarlo poco a poco:

```
public static void aniadirFactura() {
        System.out.println("NIF del cliente: ");
        String nif = lector.nextLine();
        Cliente cli = em.find(Cliente.class, nif);
        if (cli==null) {
                System.out.println("No existe el cliente con NIF = " + nif);
                System.out.println("Seleccione primero la opción 4");
        }
        else {
                ......
        }
}
```

En primer lugar, solicitamos al usuario que teclee el NIF del cliente al que se le emitirá la factura y realizaremos una búsqueda para obtener el objeto *Cliente*.

- **Si no existe el cliente**, se informará al usuario, sugiriéndole que primero elija la opción 4 del menú (*Añadir nuevo cliente*).
- **Si el cliente existe**, continuaremos con el resto de los pasos hasta dar de alta la nueva factura.

```
System.out.println("Cliente: " + cli.getNombre());
TypedQuery<Integer> query = em.createQuery("SELECT MAX(f.numero) FROM Factura
f", Integer.class);
Integer numFactura = query.getSingleResult() + 1;
System.out.println("Número de factura: " + numFactura);
```

Una vez que hemos obtenido el objeto *Cliente*, obtenemos el número de la nueva factura a través de una consulta.

Los números de factura van consecutivos, así que **la nueva factura tendrá como número el mayor de los números de factura más una unidad.**

```
// Construimos objeto factura.
Factura f = new Factura();
```

```
f.setNumero(numFactura);
f.setCliente(cli);
f.setPagado(false);
f.setFecha(new Date());
// Añadimos la factura al cliente.
cli.addFactura(f);
```

Construimos un nuevo objeto *Factura* y se lo añadimos al cliente por medio del método*addFactura()*.

```
// Iniciamos la transacción y persistimos la factura.
EntityTransaction et = em.getTransaction();
et.begin();
em.persist(f);
```

Iniciamos la transacción y persistimos la nueva factura. Pero no cerramos después la transacción porque todavía falta construir y persistir las líneas de detalle (objetos *Detalle*).

```
// Construimos los detalles de la factura.
String continuar;
do {
        System.out.println("Código de artículo: ");
        String codigo = lector.nextLine();
        Producto pro = em.find(Producto.class, codigo);
        if (pro==null) {
                System.out.println("No se encuentra el producto con código " +
codigo);
        }
        else {
                System.out.println("Descripción: " +  pro.getDescripcion());
                System.out.println("Precio: " + pro.getPrecio());
                System.out.println("¿Cuantas unidades desea? ");
                int unidades = lector.nextInt();
                lector.nextLine(); // Recoge el retorno de carro.

                // Creamos el objeto Detalle con su primary key compuesta.
                DetallePK pk = new DetallePK();
                pk.setCodigo(codigo);
                pk.setNumero(numFactura);
                Detalle d = new Detalle();
                d.setId(pk);
                d.setPrecio(pro.getPrecio());
                d.setUnidades(unidades);
                d.setProducto(pro);
                d.setFactura(f);

                // Persistimos el detalle.
                em.persist(d);

                // Añadimos el detalle a la factura.
                f.addDetalle(d);
        }
        System.out.println("¿Deseas seguir comprando (S/N)? ");
        continuar = lector.nextLine();
```

```
} while (continuar.toUpperCase().equals("S"));
```

De manera repetitiva, mientras el usuario quiera, permitiremos añadir nuevas líneas de detalle para la factura. Por cada detalle se solicita al usuario el código de producto y se busca. Una vez localizado el producto, se informa de la descripción y el precio, se solicitan las unidades que se van a vender y, a continuación, se crea la nueva línea de detalle y se persiste.

Cada línea de detalle se añade al objeto *Factura* mediante el método *addDetalle()*.

Si has ejecutado el programa, puede que no haya funcionado y que hayas terminado con una excepción de tipo *NullPointerException*. No te preocupes, porque lo vamos a solucionar.

De nuevo es un problema que viene originado por cómo están implementadas las clases de entidad, y tendremos que hacer un pequeño retoque. Recuerda que un objeto *Factura* incluye una colección de tipo *List* de objetos *Detalle*.

```
@OneToMany(mappedBy="factura")
	private List<Detalle> detalles;
```

Cuando se construye un objeto *Factura* a partir de la consulta de una factura existente, la propiedad*detalles* es inyectada con la colección de detalles de dicha factura, y se podrían añadir más*detalle* mediante la llamada al método *addDetalle()*.

Pero cuando se trata de una factura añadida recientemente que todavía no tiene líneas de detalle, la propiedad *detalles* tiene el valor *null* y el método *addDetalle()* arroja una excepción, porque intenta añadir un elemento a una colección que no está construida.

La solución está en **añadir una línea al constructor de *Factura* para crear la colección**.

¡Ojo! Recuerda que *List* no es una clase sino una interfaz, y es necesario construir un objeto de una clase que implemente la interfaz *List*; por lo tanto, construiremos un objeto *Vector*.

```
public Factura() {
		detalles = new Vector<Detalle>();
	}
```

Constructor de *Factura* añade un nuevo objeto *Vector* vacío, listo para guardar nuevos objetos *Detalle*.

Podríamos tener el mismo problema para añadir nuevas facturas a un objeto *Cliente* si el cliente no tiene aún una factura. También deberíamos modificar el construir de *Cliente*, dejándolo así:

```
public Cliente() {
        facturas = new Vector<Factura>();
}
```

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- **JPA (*Java Persistence API*)** es el *framework* estándar proporcionado por *Java Enterprise Edition* (Java EE) para la capa de persistencia que implementa el concepto de ORM.
- Un ***framework* ORM** implementa la técnica de **Mapeo objeto-relacional,**proporcionando una estructura de objetos que representa la base de datos física.
- El Mapeo objeto-relacional se efectúa por medio de las **clases de entidad,** que representan la estructura de tablas de la base de datos real. Dentro de nuestra aplicación, trabajamos con una base de datos orientada a objetos que es una representación de la base de datos real.
- La clase principal con la que trabajamos en una aplicación JPA es el ***EntityManager***, administrador de los objetos de entidad (base de datos orientada a objetos virtual) que nos permite realizar las operaciones CRUD con la base de datos, a partir de los objetos de entidad.
- En un proyecto JPA, la configuración de conexión a una base de datos está dentro del archivo ***persistence.xml***.