



Professional Java Developer



**Desarrollando Soluciones
con Java**

Eric Gustavo Coronel Castillo

ISIL - 2014

Prologo

Ser un desarrollador competente hoy en día, implica en primer lugar tener los conceptos claros sobre la Programación Orientada a Objetos (POO), ya que es la base de los lenguajes de programación.

Otro aspecto que sin duda cada día se convierte en un estándar de todo desarrollador es el conocimiento y correcta aplicación de los patrones de diseño, si bien éste libro no es precisamente de patrones, si se aplica en los ejemplos, sobre todo en los últimos capítulos, donde usted encontrará como aplicar el patrón DAO y MVC.

Java es el lenguaje de programación más utilizado en el desarrollo de aplicaciones robustas que tengan un requerimiento de funcionamiento de 24x7x365, y con la ventaja de poder implementarlas en cualquier plataforma.

En este contexto, es necesario contar con un texto que sirva de guía para crear Soluciones Empresariales aplicando JEE, JDBC y patrones de diseño DAO y MVC.

Esta obra abarca estos temas de manera muy objetiva, donde el lector podrá comprender los conceptos ayudado mediante diagramas UML y luego implementados en Java.

El libro esta organizado en 4 partes y 18 capítulos, en cada uno de los capítulos se describen y explican los conceptos del tema planteado y se ilustra con ejemplos muy prácticos y de inmediata aplicación en un contexto real.

Parte 1: Programación Orientada a Objetos

Capítulo 01: Introducción al Desarrollo de Software

En este capítulo se desarrolla una visión general sobre el desarrollo de software.

Capítulo 02: Software a Utilizar

En este capítulo usted conocerá de donde obtener el software y su respectiva instalación.

Capítulo 03: Clases, Campos y Métodos

En este capítulo se desarrolla los primeros conceptos que el desarrollador debe conocer sobre la POO.

Capítulo 04: Constructores y Destructores

En este capítulo encontrará los conceptos de constructores y destructores, y sobre todo su aplicación práctica.

Capítulo 05: Relaciones entre Clases

Tema fundamental para entender los diagramas UML.

Capítulo 06: Java Collection Framework

Tema muy importante para el manejo de colecciones de datos, como por ejemplo, un conjunto de registros que provienen de una tabla.

Parte 2: Base de Datos MySQL

Capítulo 07: Introducción a MySQL

Primeros pasos para trabajar con MySQL.

Capítulo 08: Especificaciones del Proyecto Banco

Presentación del proyecto **EurekaBank** que es tomado como base para los ejemplos con base de datos en todo el libro.

Capítulo 09: Consulta de Datos

En este capítulo desarrollaremos el tema de consultas simples, básicamente las cláusulas SELECT, FROM y WHERE.

Capítulo 10: Consultas Avanzadas de Datos

Aprenderá a desarrollar consultas avanzadas, aplicando funciones de grupo, GROUP BY, HAVING, consultas multitablas y subconsultas.

Capítulo 11: Trabajando con Datos

En este capítulo se desarrolla el tema de modificación de datos, específicamente las instrucciones INSERT, UPDATE y DELETE, así como también el manejo de transacciones.

Capítulo 12: Programación

La potencia de un motor de base de datos está en la capacidad de poder ejecutar bloques de código en forma de función o procedimiento, este es el tema que desarrollado en este capítulo.

Parte 3: JDBC

Capítulo 13: Acceso a Base de Datos

En este capítulo conocerá todas las alternativas que provee JDBC para acceder a diferentes tipos de fuentes de datos.

Capítulo 14: Consultas

En este capítulo aprenderá a ejecutar consultas utilizando JDBC, desde las mas simples hasta las que utilizan parámetros con procedimientos almacenados.

Capítulo 15: Manejo de Transacciones

El tema mas delicado de todo sistemas, el manejo de transacciones; recuerde que las transacciones mal programadas puede decaer en una base de datos inconsistente y por lo tanto la información que proviene de ella ya no sería confiable.

Este es el tema que abordamos en este capítulo.

Parte 4: Desarrollo Web con JSP

Capítulo 16: Servlets

El servlets es la base de desarrollo Web con JSP, por lo tanto es fundamental conocer este tema con bastante detalle.

Este es el tema desarrollado en este capítulo.

Capítulo 17: Java Server Page

Las dificultades que se presentan en la programación son servlets son superadas con las JSP. Usted verá en este capítulo lo fácil que resulta la creación de soluciones aplicando JSP.

Capítulo 18: Uso de JSTL

Si las JSP le resultó fácil en el capítulo anterior, las JSTL le resulta aún mas fácil y elegante para construir soluciones empresariales.

Introducción

Java es el lenguaje de programación que más personas en el mundo lo utilizan, tanto en el ambiente académico, como para el desarrollo de aplicaciones empresariales.

Java es un Lenguaje Orientado a Objetos, desde el primer programa, por más simple que este sea, esta usted implementado una clase; entre sus principales características podemos mencionar:

- Multiplataforma, por que Java corre sobre cualquier sistema operativo.
- Multiproceso, por que un programa Java puede ejecutar varios procesos simultáneamente.
- Orientado a Objetos
- Seguro, por que elimina aspectos como los punteros, que pueden utilizarse para acceder a secciones de la memoria no permitida.

A esto tenemos que agregarle una gran cantidad de librerías y recursos para desarrollar todo tipo de aplicaciones, desde las más sencillas, como las de consola, aplicaciones de entorno grafico, y las más avanzadas, como componentes, aplicaciones Web, aplicaciones móviles, etc.

La madures que ha tenido Java a lo largo del tiempo, lo hacen el lenguaje preferido por empresas que realizan desarrollo serio, y en nuestro medio son cada vez más las empresas que lo utilizan.

Para hacer Desarrollo Empresarial se aplica JDBC, Servlets, JSP, JSTL y Patrones de Diseño, todos estos temas son abordados en este libro desde un enfoque práctico, con los conceptos muy precisos.

Finalmente, los ejemplos desarrollados se pueden descargar del siguiente repositorio:

https://github.com/gcoronelc/CD_Libros

El archivo es:

CD_Desarrollando_Soluciones_Con_Java.rar

Índice

Parte 01: Programación Orientada a Objetos

Capítulo 01: Introducción al Desarrollo de Software

- 1.1. Por que usar una Metodología de Software
- 1.2. Rational Unified Process (RUP)
- 1.3. Como Aplicar UML
 - 1.3.1. Requerimientos
 - 1.3.2. Diagramas UML
 - 1.3.3. Diagrama Caso de Uso
 - 1.3.4. Documentación de Caso de Uso
 - 1.3.5. Diagrama de clases
 - 1.3.5.1. Modelo de datos
 - 1.3.5.2. Modelo de sistemas
 - 1.3.6. Diagrama de Secuencia
 - 1.3.7. Diagrama de Estado
 - 1.3.8. Diagrama de Actividad
 - 1.3.9. Diagrama de Componentes
 - 1.3.10. Diagrama de Despliegue
 - 1.4. Trabajando con Patrones de Software
 - 1.4.1. MVC (Model View Controller)
 - 1.4.1.1. Problema
 - 1.4.1.2. Solución
 - 1.4.2. TO (Transfer Object)
 - 1.4.2.1. Problema
 - 1.4.2.2. Solución

Capítulo 02: Software a Utilizar

- 2.1. NetBeans
- 2.2. MySQL
 - 2.2.1. Software
 - 2.2.2. Instalación
 - 2.2.3. Trabajando con MySQL

Capítulo 03: Clases, Campos y Métodos

- 3.1. Definición de una Clase
 - 3.1.1. Definición de una Clase
 - 3.1.2. Representación UML de una Clase
 - 3.1.3. Declaración de Objetos
 - 3.1.4. Asignación de Objetos
- 3.2. Trabajando con Campos
 - 3.2.1. Definición
 - 3.2.2. Ocultando los Datos
- 3.3. Trabajando con Métodos
 - 3.3.1. Definición
 - 3.3.2. Sobrecarga de Métodos
- 3.4. Ejemplo Demostrativo
 - 3.4.1. Requerimientos del Software
 - 3.4.2. Abstracción
 - 3.4.3. Diagrama de Secuencia
 - 3.4.4. Diagrama de Clases
 - 3.4.5. Estructura del Proyecto en NetBeans
 - 3.4.5.1. Codificación de Clase Banco
 - 3.4.5.2. Codificación de la Interfaz de Usuario IUBanco

Capítulo 04: Constructores y Destructores

- 4.1. Constructores
- 4.2. Destructores
- 4.3. Alcance de Instancia y de Clase
 - 4.3.1. Alcance de Instancia
 - 4.3.2. Alcance de Clase
 - 4.3.3. Inicializador de Clase

Capítulo 05: Relaciones Entre Clases

- 5.1. Introducción
- 5.2. Dependencia
- 5.3. Generalización
- 5.4. Asociación
 - 5.4.1. Asociación Binaria
 - 5.4.1.1. Nombre de la asociación (association name)
 - 5.4.1.2. Nombre de rol (rolename)
 - 5.4.1.3. Multiplicidad (multiplicity)
 - 5.4.1.4. Ordenación (ordering)
 - 5.4.1.5. Modificabilidad (change ability)
 - 5.4.1.6. Navegabilidad (na vigability)
 - 5.4.1.7. Visibilidad (visibility)
 - 5.4.2. Agregación y Composición
 - 5.4.3. Asociación n-aria

Capítulo 06: Java Collection Framework

- 6.1. Introducción
- 6.2. Elementos de JCF
 - 6.2.1. Interfaces del core de JCF
 - 6.2.2. Interfaces de Soporte
 - 6.2.3. Clases de propósito general
 - 6.2.4. Interfaz Comparable y Comparator
- 6.3. Casos Prácticos
 - 6.3.1. Clase base
 - 6.3.2. Manejo de listas mediante ArrayList
 - 6.3.3. Manejo de listas mediante HashSet
 - 6.3.4. Manejo de listas mediante TreeSet
 - 6.3.5. Ordenar y buscar datos en una lista
 - 6.3.6. Manejo de datos de tipo clave/valor

Parte 2: Base de Datos MySQL

Capítulo 07: Introducción a MySQL

- 7.1. Verificación del Servicio de MySQL
- 7.2. Programa Cliente de Línea de Comando de MySQL
- 7.3. Ejecutando Comandos

Capítulo 08: Especificaciones del Proyecto Banco

- 8.1. Requerimiento de Software
- 8.2. Estándares Utilizados
 - 8.2.1. Nombre de las Tablas
 - 8.2.2. Nombre de las Columnas
 - 8.2.3. Nombre de las Restricciones
 - 8.2.3.1. Primary Key
 - 8.2.3.2. Foreign Key
 - 8.2.3.3. Check
 - 8.2.3.4. Unique
 - 8.2.4. Nombres de Índices
- 8.3. Diseño de la Base de Datos
 - 8.3.1. Tablas Generales
 - 8.3.1.1. Tabla: Moneda
 - 8.3.1.2. Tabla: CargoMantenimiento

- 8.3.1.3. Tabla: CargoMovimiento
 - 8.3.1.4. Tabla: InteresMensual
 - 8.3.1.5. Tabla: TipoMovimiento
 - 8.3.1.6. Tabla: Contador
 - 8.3.1.7. Tabla: Parámetro
 - 8.3.2. Sucursales
 - 8.3.2.1. Tabla: Sucursal
 - 8.3.2.2. Tabla: Empleado
 - 8.3.2.3. Tabla: Asignado
 - 8.3.3. Cuentas
 - 8.3.3.1. Tabla: Cliente
 - 8.3.3.2. Tabla: Cuenta
 - 8.3.3.3 Tabla: Movimiento
 - 8.3.4. Esquema Completo
 - 8.4. Creación de la Base de Datos
- Capítulo 09: Consulta de Datos**
- 9.1. Fundamentos
 - 9.1.1. SQL Fundamentos
 - 9.1.1.1. Lenguaje de Definición de Datos (DDL)
 - 9.1.1.2. Lenguaje de Manipulación de Datos (DML)
 - 9.1.1.3. Instrucciones de Control de Transacciones
 - 9.1.1.4. Instrucciones de Administración de la Base de Datos
 - 9.1.2. Operadores y Literales
 - 9.1.2.1. Operadores Aritméticos
 - 9.1.2.2. Precedencia de Operadores
 - 9.1.2.3. Literales
 - 9.1.2.4. Operadores de Comparación
 - 9.1.2.5. Operadores Lógicos
 - 9.1.2.6. Reglas de Comparación
 - 9.2. Escribiendo Consultas Simples
 - 9.2.1. Sintaxis Básica
 - 9.2.2. Usando la Sentencia SELECT
 - 9.2.2.1. Consulta del contenido de una Tabla
 - 9.2.2.2. Seleccionando Columnas
 - 9.2.2.3. Alias para Nombres de Columnas
 - 9.2.2.4. Asegurando Filas Únicas
 - 9.2.3. Filtrando Filas
 - 9.2.3.1. Operador de Igualdad (=)
 - 9.2.3.2. Operador Diferente (!=, <>)
 - 9.2.3.3. Operador Menor Que (<)
 - 9.2.3.4. Operador Mayor Que (>)
 - 9.2.3.5. Operador: IS [NOT] NULL
 - 9.2.3.6. Operador: [NOT] BETWEEN exp_min AND exp_max
 - 9.2.3.7. Operador: [NOT] IN
 - 9.2.3.8. Operador: NOT
 - 9.2.3.9. Operador: AND
 - 9.2.3.10. Operador: OR
 - 9.2.3.11. Operador: XOR
 - 9.2.3.12. Operador: [NOT] LIKE
 - 9.2.4. Ordenando Filas
 - 9.2.5. Uso de LIMIT
 - 9.3. Funciones Simples de Filas
 - 9.3.1. Funciones de Control de Flujo
 - 9.3.1.1. Función CASE
 - 9.3.1.2. Función IF()
 - 9.3.1.3. Función IFNULL()
 - 9.3.1.4. Función NULLIF()
 - 9.3.2. Funciones de Cadena
 - 9.3.3. Funciones Numéricas

- 9.3.4. Funciones de Fecha
 - 9.3.4.1. Formato de Fecha
 - 9.3.4.2. Formatos de Intervalos
 - 9.3.4.3. Función: ADDDATE()
 - 9.3.4.4. Funciones: CURDATE(), CURRENT_DATE, CURRENT_DATE()
 - 9.3.4.5. Funciones: CURTIME(), CURRENT_TIME, CURRENT_TIME()
 - 9.3.4.6. Funciones: NOW(), CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(), LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP(), SYSDATE()
 - 9.3.4.7. Función: DATE()
 - 9.3.4.8. Función: DATEDIFF()
 - 9.3.4.9. Funciones: DATE_ADD(), DATE_SUB()
 - 9.3.4.10. Función: DATE_FORMAT()
 - 9.3.4.11. Función: DAYOFMONTH(), DAY()
 - 9.3.4.12. Función: DAYNAME()
 - 9.3.4.13. Función: DAYOFWEEK()
 - 9.3.4.14. Función: DAYOFYEAR()
 - 9.3.4.15. Función: EXTRACT()
 - 9.3.4.16. Función: GET_FORMAT()
 - 9.3.4.17. Función: LAST_DAY()
 - 9.3.4.18. Función: MONTH()
 - 9.3.4.19. Función: MONTHNAME()
 - 9.3.4.20. Función: WEEK()
 - 9.3.4.21. Función: WEEKDAY()
 - 9.3.4.22. Función: WEEKOFYEAR()
 - 9.3.4.23. Función: YEAR()
- 9.3.5. Funciones de Conversión
 - 9.3.5.1. Tipos de conversión
 - 9.3.5.2. Función: Cast()
 - 9.3.5.3. Función: CONVERT()
- 9.3.6. Funciones de Encriptación
 - 9.3.6.1. Funciones: AES_ENCRYPT(), AES_DECRYPT
 - 9.3.6.2. Funciones: ENCODE(), DECODE()
- 9.3.6.3. Funciones: DES_ENCRYPT(), DES_DECRYPT()

Capítulo 10: Consulta Avanzada de Datos

- 10.1. Totalizando Datos y Funciones de Grupo
 - 10.1.1. Funciones de Grupo
 - 10.1.1.1. Función: AVG()
 - 10.1.1.2. Función: COUNT()
 - 10.1.1.3. Función: COUNT DISTINCT
 - 10.1.1.4. Función: GROUP_CONCAT()
 - 10.1.1.5. Función: MAX()
 - 10.1.1.6. Función: MIN()
 - 10.1.1.7. Función: SUM()
 - 10.1.2. GROUP BY
 - 10.1.3. HAVING
- 10.2. Consultas Multitablas
 - 10.2.1. ¿Qué es un Join?
 - 10.2.2. Consultas Simples
 - 10.2.3. Consultas Complejas
 - 10.2.3.1. Uso de Alias
 - 10.2.4. Usando Sintaxis ANSI
 - 10.2.4.1. NATURAL JOIN
 - 10.2.4.2. JOIN . . . USING
 - 10.2.4.3. JOIN ... ON
 - 10.2.4.4. STRAIGHT_JOIN
 - 10.2.5. Producto Cartesiano
 - 10.2.6. Joins Externos
 - 10.2.6.1. LEFT JOIN

- 10.2.6.2. RIGHT JOIN
- 10.2.7. Consultas Autoreferenciadas
- 10.2.8. Unión de Resultados
- 10.3. Subconsultas
 - 10.3.1. Subconsultas Como Tabla Derivada
 - 10.3.2. Subconsulta Como Expresión
 - 10.3.3. Subconsulta para Correlacionar Datos
 - 10.3.4. Operador: [NOT] EXISTS
 - 10.3.5. Operador: [NOT] IN
 - 10.3.6. Consultas de Referencias Cruzadas

Capítulo 11: Trabajando con Datos

- 11.1. Insertando Filas
 - 11.1.1. Caso 1
 - 11.1.1.1. Inserciones una Sola Fila
 - 11.1.1.2. Insertando Valores Nulos
 - 11.1.1.3. Insertando Varias Filas
 - 11.1.2. Caso 2
 - 11.1.3. Caso 3
 - 11.1.4. Cláusula: ON DUPLICATE KEY UPDATE
- 11.2. Modificando Datos
 - 11.2.1. Actualización Simple
 - 11.2.2. Seleccionando las Filas a Actualizar
 - 11.2.3. Actualizando Columnas con Subconsultas
 - 11.2.4. Error de Integridad Referencial
- 11.3. Eliminando Filas
 - 11.3.1. Introducción
 - 11.3.2. Eliminar Todas las Filas de una Tabla
 - 11.3.3. Seleccionando las Filas a Eliminar
 - 11.3.3.1. Eliminando una Sola Fila
 - 11.3.3.2. Eliminando un Grupo de Filas
 - 11.3.4. Uso de Subconsultas
 - 11.3.5. Error de Integridad Referencial
 - 11.3.6. Truncando una Tabla
- 11.4. Transacciones
 - 11.4.1. Introducción
 - 11.4.2. Propiedades de una Transacción
 - 11.4.2.1. Atomicity (Atomicidad)
 - 11.4.2.2. Consistency (Coherencia)
 - 11.4.2.3. Isolation (Aislamiento)
 - 11.4.2.4. Durability (Durabilidad)
 - 11.4.3. Operación de Transacciones
 - 11.4.3.1. Inicio de una transacción
 - 11.4.3.2. Confirmación de una transacción
 - 11.4.3.3. Cancelar una transacción

Capítulo 12: Programación

- 12.1. Funciones de usuario
 - 12.1.1. Crear nuevas funciones
 - 12.1.2. Eliminar una función
 - 12.1.3. Modificar una función
- 12.2. Procedimientos almacenados
 - 12.2.1. Creación de procedimientos
 - 12.2.2. Eliminar un procedimiento
 - 12.2.3. Modificar un procedimiento
- 12.3. Manejo de variables
 - 12.3.1. Sentencia DECLARE
 - 12.3.2. Sentencia SET
 - 12.3.3. Sentencia SELECT . . . INTO
- 12.4. Sentencias de control de flujo

- 12.4.1. Sentencia IF
- 12.4.2. Sentencia CASE
- 12.4.3. Sentencia LOOP
- 12.4.5. Sentencia LEAVE
- 12.4.6. Sentencia ITERATE
- 12.4.7. Sentencia REPEAT
- 12.4.8. Sentencia WHILE
- 12.5. Manejo de Errores
 - 12.5.1. Condición nombrada
 - 12.5.2. Manejador de error
- 12.6. Cursores
 - 12.6.1. Declaración de cursores
 - 12.6.2. Abrir un cursor
 - 12.6.3. Leer datos del cursor
 - 12.6.4. Cerrar el cursor

Parte 3: Programación con JDBC

Capítulo 13: Acceso a Bases de Datos

- 13.1. Introducción
- 13.2. ¿Qué es el API JDBC?
 - 13.2.1. ¿Qué hace el API JDBC?
 - 13.2.2. El API JDBC y ODBC frente a UDA
 - 13.2.3. Modelos de dos niveles y tres niveles
 - 13.2.4. Conformidad de SQL
 - 13.2.5. Productos JDBC
 - 13.2.6. Estructura de JDBC
 - 13.2.7. Tipos de Controladores JDBC
 - 13.2.7.1. Tipo 1: JDBC-ODBC bridge plus ODBC driver
 - 13.2.7.2. Tipo 2: Native-API partly-Java driver
 - 13.2.7.3. Tipo 3: 100% Pure Java, JDBC – Network
 - 13.2.7.4. Tipo 4: 100% Java
- 13.3. Conexión a una Fuente de Datos
 - 13.3.1. Driver y DriverManager
 - 13.3.1.1. Registrar Controladores JDBC
 - 13.3.1.2. Seleccionando y Desregistrando Controladores
 - 13.3.2. Trabajando con Objetos Connection
 - 13.3.2.1. Entendiendo JDBC URLs
 - 13.3.2.2. Abriendo Conexiones
 - 13.3.2.3. Cerrando las Conexiones de Base de Datos

Capítulo 14: Consultas

- 14.1. Introducción
- 14.2. Usando Statement
- 14.3. Usando PreparedStatement
- 14.4. Uso de CallableStatement
 - 14.4.1. Utilizando Parámetros
 - 14.4.2. Manejando Conjunto de Resultados

Capítulo 15: Manejo de Transacciones

- 15.1. Definiciones
 - 15.1.1. Transacción
 - 15.1.2. Propiedades de una Transacción
 - 15.1.3. Control de Transacciones
 - 15.1.3.1. Transacciones Controladas desde el cliente
 - 15.1.3.2. Transacciones Controladas en la Base de Datos.
 - 15.1.3.3. Transacciones Distribuidas
- 15.2. Programación de Transacciones
 - 15.2.1. Transacciones Controladas desde el cliente
 - 15.2.2. Transacciones de Base de Datos

Parte 3: Desarrollo Web con JSP

Capítulo 16: Servlets

- 16.1. Introducción General
- 16.2. Introducción a los Servlets
 - 16.2.1. ¿Qué es un Servlets?
 - 16.2.2. Ventajas de los Servlets
 - 16.2.3. Utilizar Servlets en lugar de Scripts CGI!
 - 16.2.4. Otros usos de los Servlets
- 16.3. Arquitectura del Paquete Servlet
 - 16.3.1. El Interfaz Servlet
 - 16.3.2. Interacción con el Cliente
 - 16.3.3. La Interfaz ServletRequest
 - 16.3.4. La Interfaz ServletResponse
 - 16.3.5. Capacidades Adicionales de los Servlets http
- 16.4.- Un Servlet Sencillo
 - 16.4.1. Creación de un Servlet Sencillo
 - 16.4.2. Descripción del Servlet
 - 16.4.3. Ejecución del Servlet
- 16.5.- Interacción con los Clientes
 - 16.5.1. Requerimientos y Respuestas
 - 16.5.2. Manejar Requerimientos GET y POST
 - 16.5.3. Problemas con los Threads
 - 16.5.4. Servlet Recursivo
- 16.6.- Programación de servlets
 - 16.6.1. Programación
 - 16.6.2. Esquema de funcionamiento
 - 16.6.2.1. Instanciación de un Servlet
 - 16.6.2.2. Nota sobre Inicialización de un Servlet
 - 16.6.2.3. Explicación Resumida del Proceso
 - 16.6.2.4. Respecto al Método destroy() del Ciclo de Vida
- 16.7. Servlets y JavaBeans
 - 16.7.1. ¿Qué es un JavaBeans?
 - 16.7.2. Propiedades
- 16.8. Interacción con un Servlet
 - 16.8.1. Consideraciones Previas
 - 16.8.2. Escribiendo la URL del servlet en un Navegador Web
 - 16.8.3. Llamar a un Servlet desde dentro de una página HTML
 - 16.8.4. Llamada a un Servlet desde otro Servlet
- 16.9. Sesiones
 - 16.9.1. Introducción
 - 16.9.2. La API de Seguimiento de Sesión
 - 16.9.2.1. Buscar el Objeto HttpSession de la Sesión Actual
 - 16.9.2.2. Buscar la Información Asociada con un Sesión.
 - 16.9.2.3. Asociar Información con una Sesión
 - 16.9.2.4. Finalizar una Sesión

Capítulo 17: Java Server Page

- 17.1. Definición
 - 17.1.1. ¿Qué es JSP?
 - 17.1.2. Características
 - 17.1.3. Ciclo de Vida de un Documento JSP
 - 17.1.3.1. A Nivel de Documento
 - 17.1.3.2. A nivel de Ejecución de Eventos
- 17.2. Elementos Básicos
 - 17.2.1. Declaraciones
 - 17.2.2. Expresiones JSP
 - 17.2.3. Scriptlets JSP
- 17.3. Directivas
 - 17.3.1. Introducción

- 17.3.2. Directiva: include
- 17.3.3.- Directiva: page
- 17.3.4. Directiva: taglib
- 17.4. Acciones
 - 17.4.1. Acción: <jsp:forward>
 - 17.4.2. Acción: <jsp:param>
 - 17.4.3. Acción: <jsp:include>
 - 17.4.4. Acción: <jsp:plugin>
 - 17.4.5. Acción: <jsp:useBean>
 - 17.4.6. Acción: <jsp:getProperty>
 - 17.4.7. Acción <jsp:setProperty>
- 17.5. Objetos Implicitos
 - 17.5.1. Objeto: request
 - 17.5.2. Objeto: response
 - 17.5.3. Objeto: out
 - 17.5.4 Objeto: session
 - 17.5.5. Objeto: application
 - 17.5.6. Objeto: config
 - 17.5.7. Objeto: pageContext
 - 17.5.8. Objeto: page

Capítulo 18: Uso de JSTL

- 18.1. Introducción
 - 18.1.1. ¿Qué es JSTL?
 - 18.1.2. ¿Cuál es el problema con los scriptlets JSP?
 - 18.1.3. ¿Como mejoran esta situación la librería JSTL?
 - 18.1.4. ¿Cuales son las desventajas de los JSTL?
 - 18.1.5. Librerías JSTL
 - 18.1.6. Instalación de JSTL
- 18.2. Lenguaje de Expresiones
 - 18.2.1. Introducción
 - 18.2.2. Operadores
 - 18.2.3. Acceso a datos
 - 18.2.4. Palabras Reservadas
 - 18.2.5. Objetos implícitos
 - 18.2.6. Objeto: pageContext
 - 18.2.7. Objeto: pageContext.errorData
- 18.3. Funciones EL
- 18.4. Etiquetas del Core
 - 18.4.1. Etiqueta: out
 - 18.4.2. Etiqueta: set
 - 18.4.3. Etiqueta: remove
 - 18.4.4. Etiqueta: if
 - 18.4.5. Etiquetas: choose, when, otherwise
 - 18.4.6. Etiqueta: forEach
 - 18.4.7. Etiqueta: forTokens
 - 18.4.8. Etiqueta: import
 - 18.4.9. Etiqueta: param
 - 18.4.10 Etiqueta: redirect
 - 18.4.11. Etiqueta: url
 - 18.4.12. Etiqueta: catch

Parte I



Programación Orientada a Objetos

Página en blanco.



Introducción al Desarrollo de software

Al iniciar un proyecto de construcción de software, nos encontramos con el Talón de Aquiles de la ingeniería de software, ¿Que metodología de software usar?. En la actualidad existen una variedad de metodologías, técnicas, recomendaciones, ninguna estándar, todas se combinan entre sí, las cuales nos ayudan a entender lo complejo que es crear un software.

Pero en toda esta lluvia de ideas y planteamientos por crear una metodología estándar, en la década de los 90 nace UML, que es estandarizado por OMG como notación para representar los planos de un software orientado a objetos, esta representación ayuda a los creadores de software a usar una simbología común y es aplicada para cualquier metodología de software.

Temas de desarrollar:

- 1.1. Por que usar una Metodología de Software
- 1.2. Rational Unified Process (RUP)
- 1.3. Como Aplicar UML
- 1.4. Trabajando con Patrones de Software

1.1. Por que usar una Metodología de Software

Antes de iniciar la creación de cualquier producto, es importante la planificación, esto involucra alcance, presupuesto, factibilidad, proyección de tiempo entre otros documentos de estudio. Nos preguntamos que hacemos primeros, como gestionar y tener éxito, que documentos debo presentar, como debo organizar mi equipo de trabajo y así muchas interrogantes.

Hace unas décadas se realizo un estudio estadístico de los proyectos de ingeniería de software, dando como resultado el famoso título "**La Crisis del Software**", titulada así por que muchos proyectos no terminaban dentro del plazo, no se ajustaban al presupuesto inicial, se desarrollaba un software de baja calidad y que no cumplían las especificaciones.

Para resolver todos estos inconvenientes es importante el uso de metodologías que le dará un conjunto de recomendación y mejores prácticas, asesorándolo en los pasos a seguir para gestionar un proyecto de desarrollo de software y generar un producto de calidad.

RUP, XP, MSF, CMMI, ITIL, COBIT y muchos más, son algunas de las metodologías más utilizadas en estos tiempos, pero el más resaltante y con mucha aceptación es RUP; hoy en día es un producto de IBM.

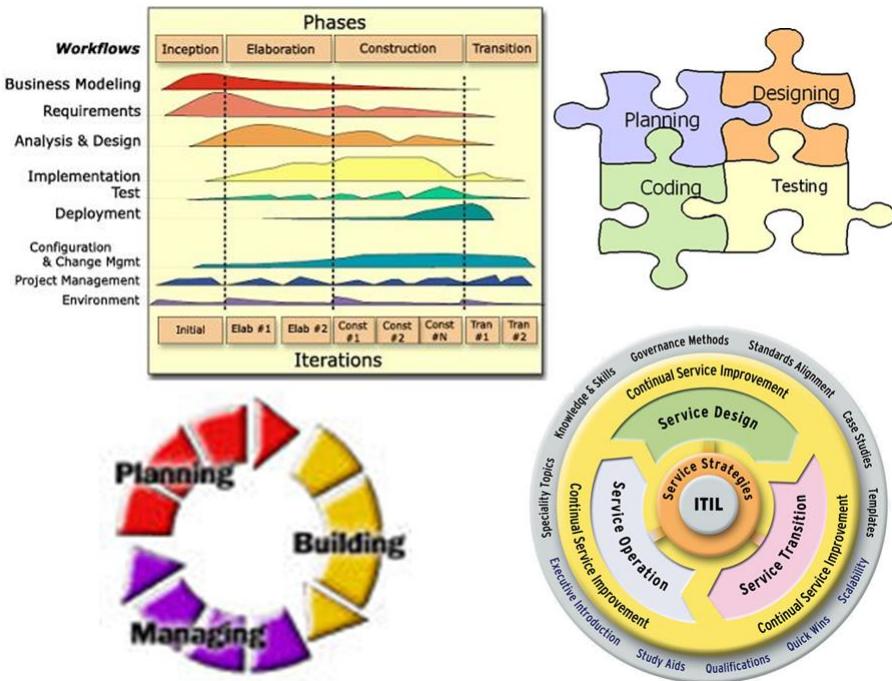


Figura 1 . 1 Metodologías de desarrollo de Software

1.2. Rational Unified Process (RUP)

Es la guía que te ofrece un conjunto de actividades, disciplinas y plantillas para documentar el software, ayudándote a generar un producto de calidad.

Figura 1 . 2 Guía Electrónica de RUP

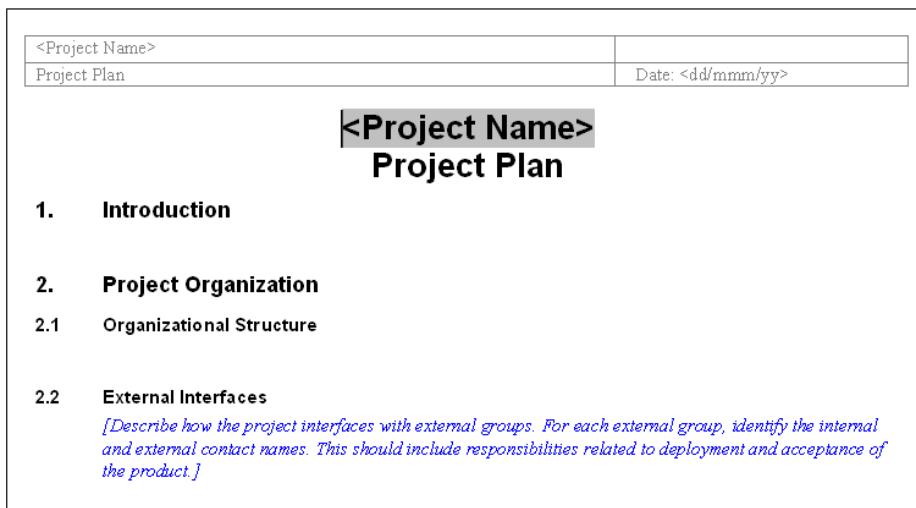


Figura 1 . 3 Plantillas que ofrece RUP.

1.3. Como Aplicar UML

En la web existe mucha crítica al Lenguaje de Modelado Unificado (UML), y existe muchos profesionales de TI, que hablan, enseñan, y documentan planos de software usando UML y en la mayoría de los casos no refleja lo implementado en el software, entonces podemos afirmar que tenemos problemas al aplicar UML. Muchos se preguntan que hago con tantos dibujitos y como se programa eso.

Unified Modeling Language - UML nace en la década de los 90 con el fin de unificar y crear una representación estándar, todo a consecuencia de un conflicto de metodologías llamada "**Guerra de los Métodos**", entonces se unen tres grandes personajes en metodología de diseño software llamados luego "**Los Tres Amigos**" (Booch, Rumbaugh y Jacobson) y crean UML que años mas tarde, en 1997, es considerado por la OMG como la representación estándar para los planos de software, esta notación puede ser utilizada o adaptada a muchas metodologías de software orientada a objetos, una metodología que nació con UML es RUP.

Tomando como analogía el plano de un edificio el cual representa en forma exacta la construcción realizada, los planos de software con UML deben ayudar al desarrollador en la fase de construcción para obtener un resultado equivalente.

UML te ayuda en todas las fases del ciclo de vida del software, desde el análisis representando los procesos del negocio y requisitos funcionales que tendrá el software (Diagrama de caso de uso), hasta la implementación de la aplicación en sí (Diagramas de secuencia, Diagramas de Actividades, Diagrama de estado, Diagrama de Clases, Diagrama de Componentes y Diagrama de despliegue)

1.3.1. Requerimientos

- Funcionalidad1
- Funcionalidad2

1.3.2. Diagramas UML

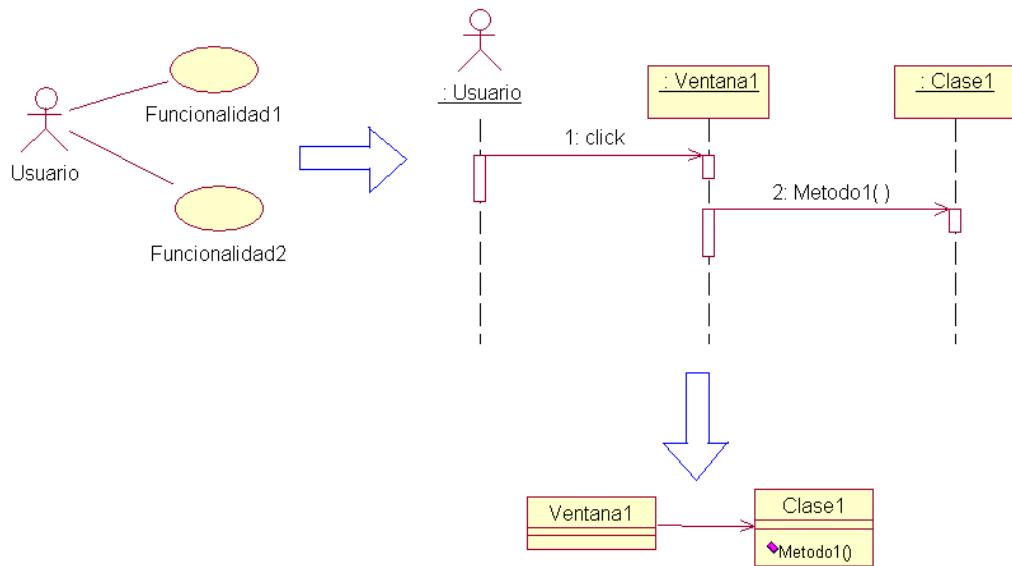


Figura 1 . 4 Diagramas UML

Este libro está enfocado a enseñarte a programar con uno de los mejores lenguajes de estos últimos tiempos y no podemos dejar de mencionar a UML, que se utilizará para representar muchos de los ejemplos a desarrollar en los diversos capítulos.

La importancia de UML ha llegado hoy en día a tal punto, que las herramientas de desarrollo, llamados IDE, incorporen diseñadores para los diferentes diagramas definidos en UML; podemos afirmar que ahora vienen con su propia herramienta CASE, pero no tienen la potencia y la productividad para un proyecto real de gran envergadura, sigo considerando que Rational Rose es la herramienta para UML usada en la mayoría de proyectos.

A continuación se muestra NetBeans y JDeveloper dos de los muchos IDEs que incorporan diseñadores de UML.

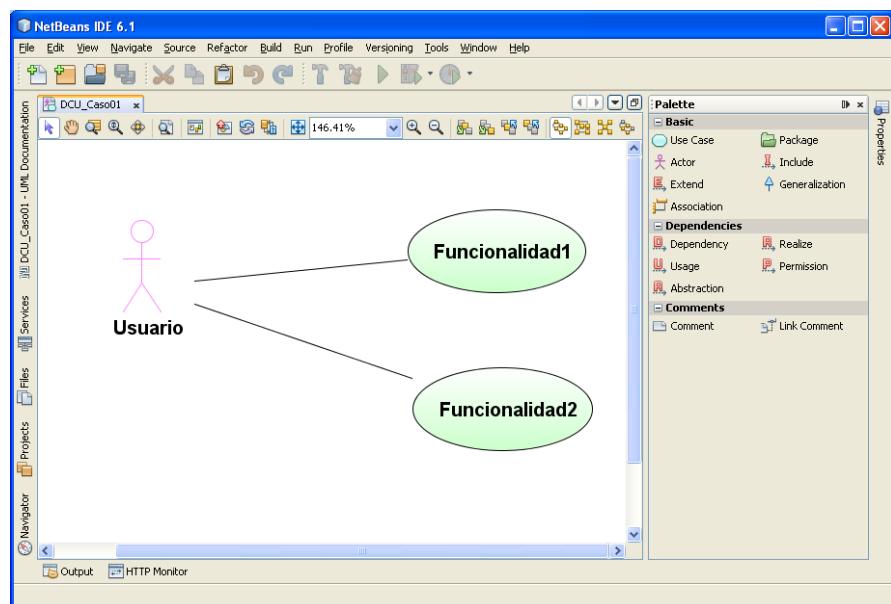


Figura 1 . 5 NetBeans incorpora un entorno de diseño aplicando UML.

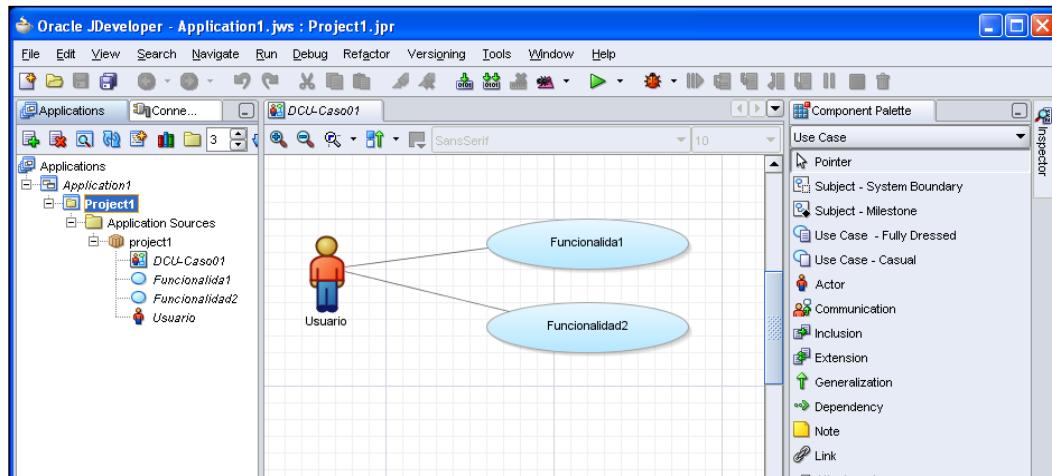


Figura 1 . 6 JDeveloper incorpora un entorno de diseño aplicando UML.

1.3.3. Diagrama Caso de Uso

Este es uno de los principales diagramas de UML y permite representar, analizar y documentar los requerimientos funcionales del software, esta compuesto por los siguientes elementos Actores, Caso de Uso y Relaciones (Comunicación, Extensión, Inclusión y Generalización).

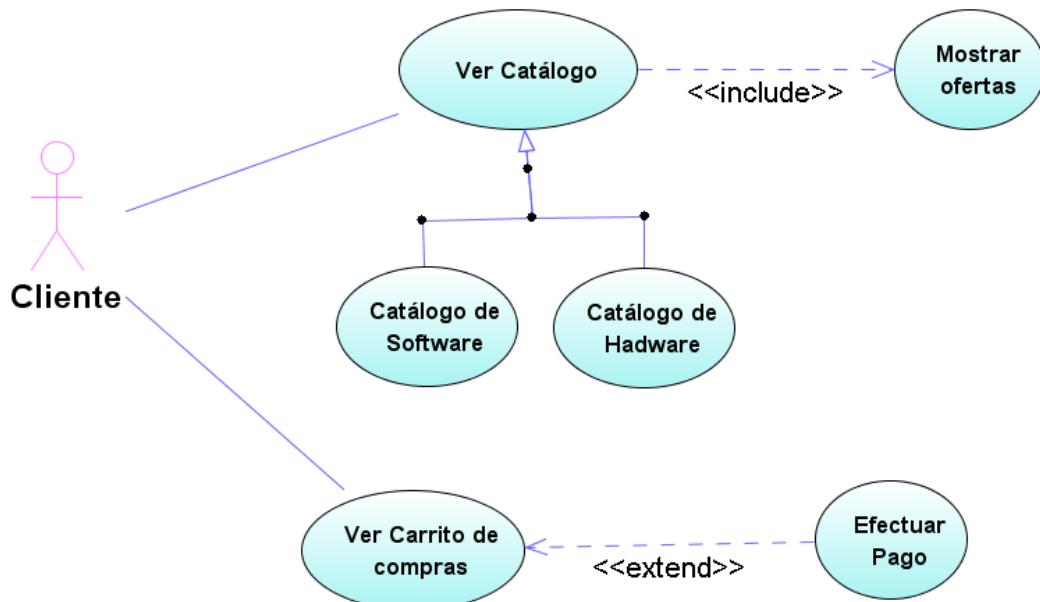


Figura 1 . 7 Diagrama de Caso de Uso en NetBeans.

1.3.4. Documentación de Caso de Uso

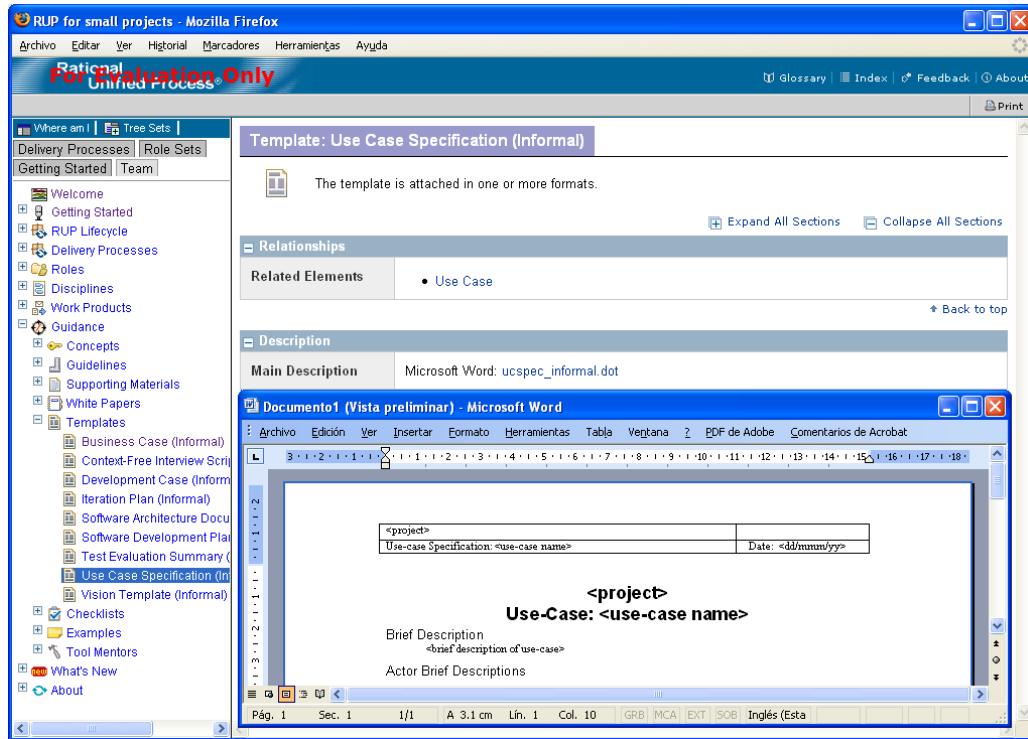


Figura 1 . 8 Plantilla de RUP para especificar Caso de Uso

Muchas de las metodologías recomiendan documentar el detalle de los pasos a realizar en cada una de estas funcionalidades (Caso de Uso), para esto existen varios modelos o plantillas que permiten escribir las especificaciones de los caso de uso, RUP incluye algunos modelos.

Por lo general la especificación de los casos de uso, adjuntan prototipos (interfaz de usuario) que describen los datos y acciones de la funcionalidad a desarrollar.

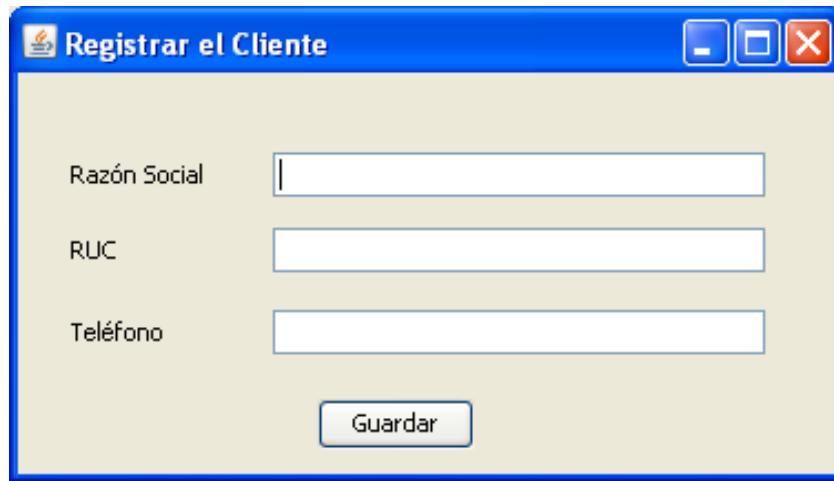


Figura 1 . 9 Prototipo basado en la especificación del caso de uso

1.3.5. Diagrama de clases

Los diagramas de clases representan la implementación que tendrá el software, podemos crear diagramas de clases enfocado al modelo de datos (implementación SQL) y modelo de sistemas (implementación de clases en un lenguaje de programación orientado a objetos).

Después de especificar los casos de uso, se realiza el proceso de abstracción para capturar las entidades que se usarán como base para los modelos de datos y modelo de sistemas.

RUP recomienda crear e incrementar estos modelos en cada iteración de la fase de Análisis y Diseño, con ayuda de las especificaciones de los casos de uso.

1.3.5.1. Modelo de datos

El siguiente diagrama de clases debe ser implementado con sentencia SQL para su creación en la base de datos.

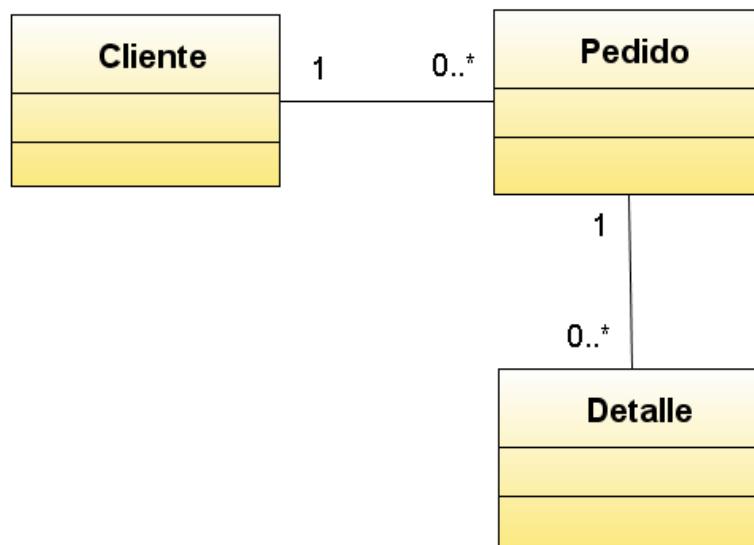


Figura 1 . 10 Modelo de datos

Implementación SQL

SQL es el lenguaje estándar para implementar sistemas gestores de base de datos relacional.

```

Create Table Cliente (
    . . . Primary Key,
    . . .
    . . .
)
Create Table Pedido (
    . . . Primary Key,
    . . .
    .
    . . .
    . . . Foreign Key . . . Cliente
)
Create Table Detalle (
    . . . Primary Key,
    . . .
    .
    . . .
    . . . Foreign Key . . . Pedido
)
  
```

1.3.5.2. Modelo de sistemas

El siguiente diagrama de clases debe ser implementado en un lenguaje de programación orientado a objetos.

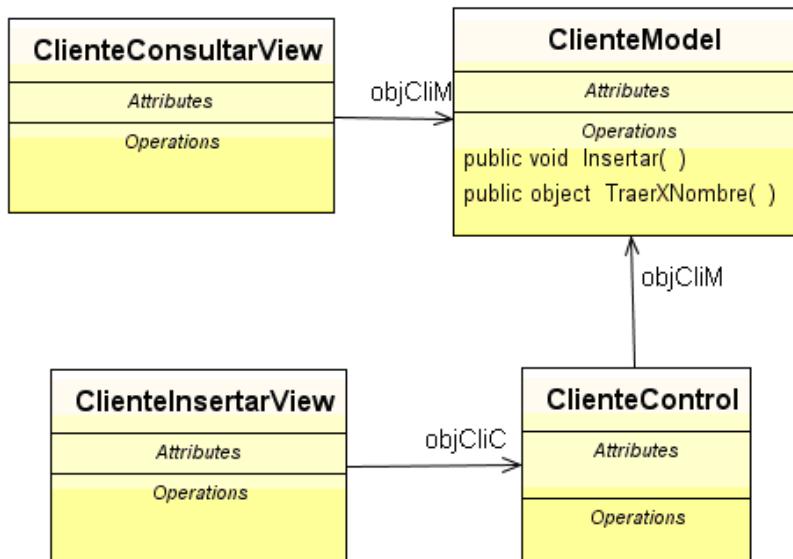


Figura 1 . 11 Modelo de sistema usando notación UML

También puede representar el diagrama usando los estereotipos Boundary, Control y Entity de RUP.

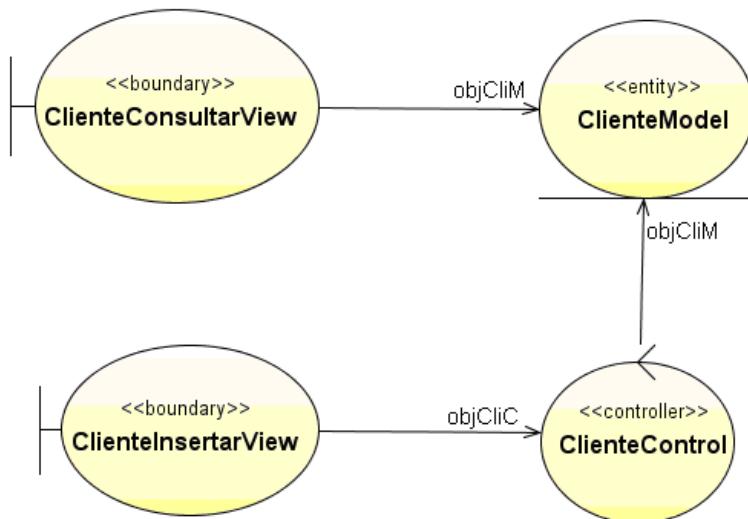


Figura 1 . 12 Modelo de sistema usando notación RUP

Implementación Java

Java es uno de los lenguajes de programación orientado a objetos considerado por muchos como el mejor en estos tiempos.

```

public class ClienteConsultarView extends JFrame {
    ClienteModel objCliM = new ClienteModel();
    ...
}
  
```

```
public class ClienteModel {  
    . . .  
}
```

1.3.6. Diagrama de Secuencia

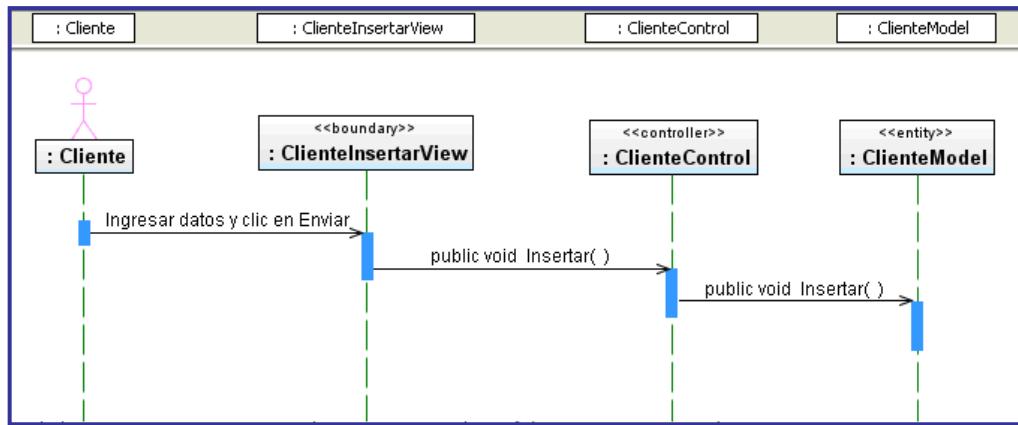


Figura 1 . 13 Envío de mensajes entre objetos (Diagrama de Secuencia)

Con los diagramas de secuencias representamos la interacción de los objetos (envío de mensajes), donde nos permite visualizar cual es la secuencia de operaciones (métodos) a realizar para resolver el conjunto de pasos indicados en el flujo normal y alternativo de la documentación de los casos de uso.

Tomando como referencia los diagramas de secuencia, se construyen las asociaciones de los diagramas de clase.

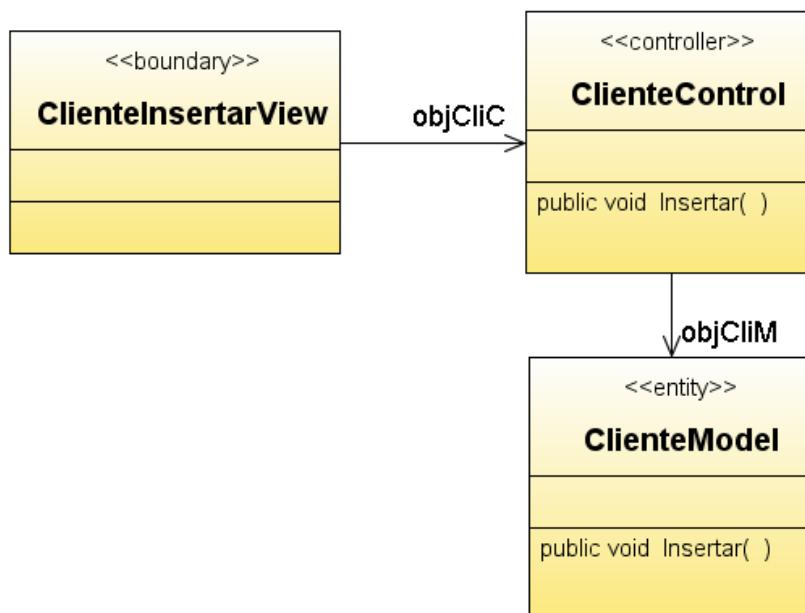


Figura 1 . 14 Diagrama de clases (Modelo de sistemas)

1.3.7. Diagrama de Estado

Por lo general este diagrama se usa para representar los estados que puede tener un objeto, cuyo valor se almacena en atributos. No todos los objetos manejan estados.

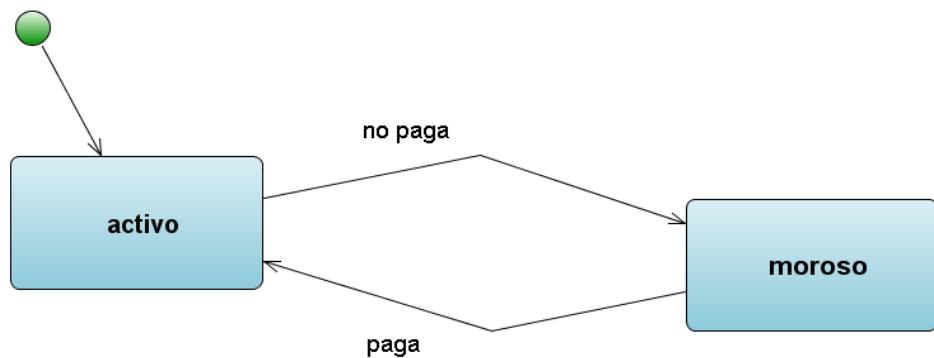


Figura 1 . 15 Diagrama de estado para el objeto cliente.

1.3.8. Diagrama de Actividad

Este diagrama se utiliza para representar el flujo de actividades de un negocio, caso de uso o una operación (método).

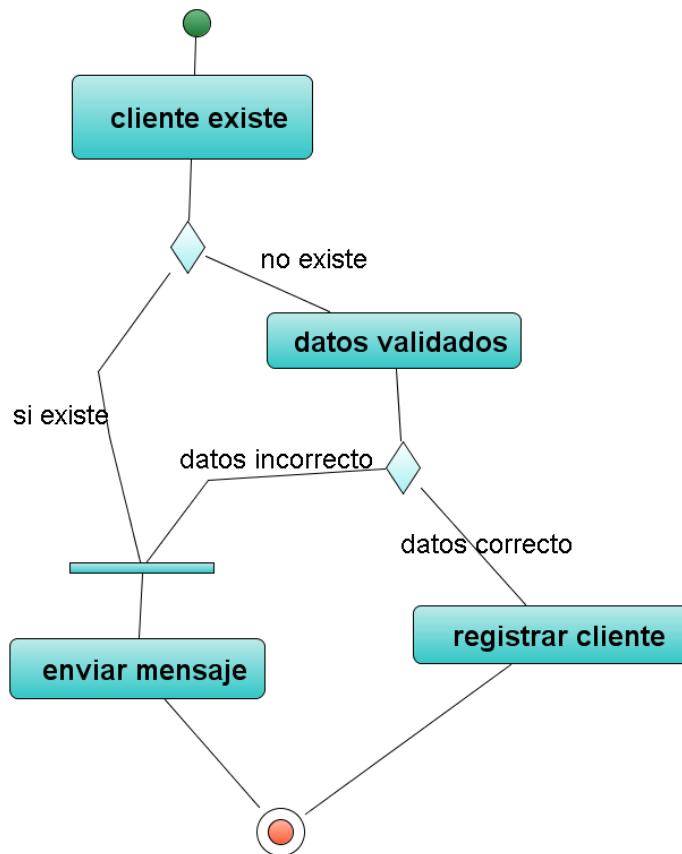


Figura 1 . 16 Diagrama de actividades de la operación Insertar.

1.3.9. Diagrama de Componentes

Los diagramas de componentes representan funcionalidad de software reutilizable, que muchas veces en un lenguaje de programación como Java están agrupados en archivos JAR o archivos DLL para .NET.

En la siguiente representación usamos un diagrama de componentes para la arquitectura en 3 capas.

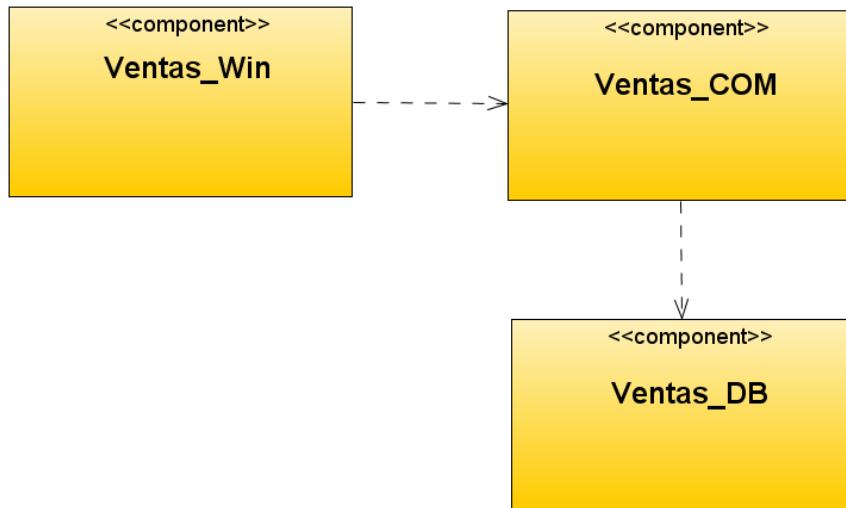


Figura 1 . 17 Diagrama de clases (Modelo de sistemas)

1.3.10. Diagrama de Despliegue

Con este diagrama representamos los dispositivos físicos que se requieren para el uso y la instalación de los componentes de software.

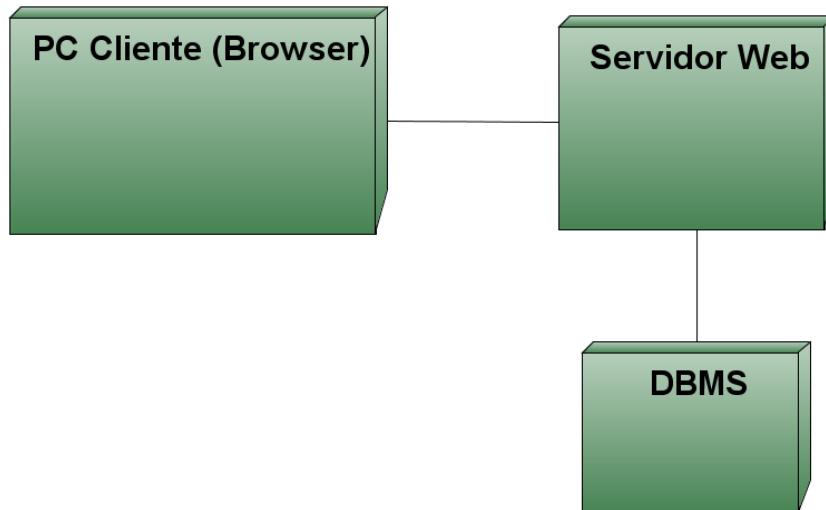


Figura 1 . 18 Diagrama de Despliegue

1.4. Trabajando con Patrones de Software

Los patrones son recomendaciones probadas y sirven como modelo para dar soluciones a problemas comunes en el desarrollo del software, son considerados en la actualidad como buenas prácticas en la ingeniería de software.

Un patrón define una posible solución correcta para un problema de diseño dentro de un contexto dado, describiendo las cualidades invariantes de todas las soluciones.

En 1979 el arquitecto **Christopher Alexander** aportó al mundo de la arquitectura el libro "**The Timeless Way of Building**"; en él proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad y manifestó la siguiente frase "**Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez.**"

No obstante, no fue hasta principios de los 90's cuando los **patrones de diseño de software** tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro "**Design Patterns**" escrito por **GOF** (Gang of Four) compuesto por **Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides**, en el que se recogían 23 patrones de diseño comunes (Factory, Singleton, Adapter, Facade, Command, Iterator entre otros)

Hoy en día existe una variedad de patrones que permite uniformizar y diseñar soluciones estándar de software, en el caso de Sun Microsystem ha publicado el "**Core J2EE Patterns**" para el desarrollo de aplicaciones empresariales.

Los patrones bastante usados en el desarrollo de aplicaciones web son **MVC** (Model View Controller) y **TO** (Transfer Object).

1.4.1. MVC (Model View Controller)

1.4.1.1. Problema

Es muy frecuente que se solicite cambios a la interfaz de usuario. Los cambios a la interfaz deberían ser fáciles y no tener consecuencias para el núcleo del código de la aplicación.

1.4.1.2. Solución

El sistema se divide en tres partes:

Modelo

Encapsula los datos y la funcionalidad de la aplicación (lógica que da solución al problema).

Vista

Interfaz de Usuario para el ingreso y presentación de datos generada por el modelo.

Controlador

Es el que controla el flujo de la aplicación y administra quien es el responsable de dar solución a un problema, este se comunica con el modelo y la vista.

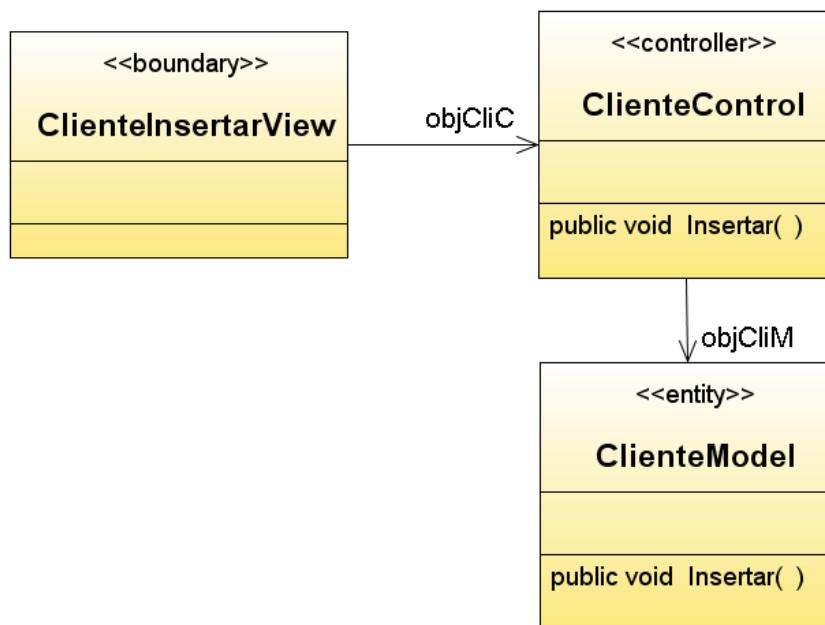


Figura 1 . 19 Modelo de un patrón MVC

1.4.2. TO (Transfer Object)

1.4.2.1. Problema

Se requiera enviar y recuperar un conjunto de datos entre un objeto y otro.

1.4.2.2. Solución

Crear una clase que agrupa un conjunto de datos que se desea transferir y por lo general implemente métodos **set** y **get** para enviar y recuperar los datos.

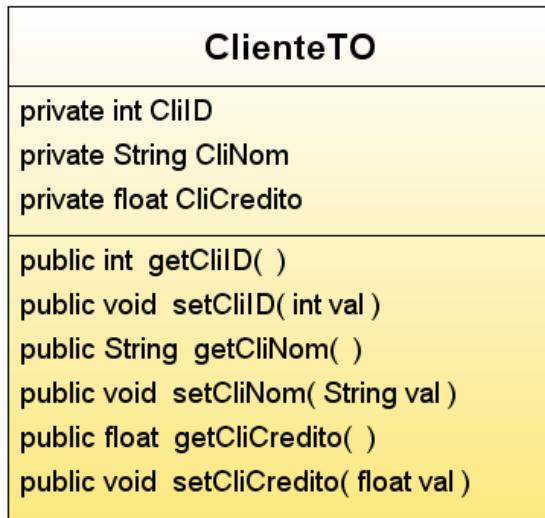


Figura 1 . 20 Clase Transfer Object (TO) antes llamado Value Object (VO)

Página en Blanco



Software a Utilizar

Para trabajar con Java tenemos varias alternativas, algunas de ellas libres y otras propietarias. En este capítulo explicaré el software a utilizar en el desarrollo del libro.

Temas a desarrollar:

- 2.1. NetBeans
- 2.2. MySQL

2.1. NetBeans

NetBeans es un IDE libre que permite desarrollar todo tipo de aplicaciones utilizando plataforma Java.

La comunidad NetBeans (<http://www.netbeans.org>) se encarga de desarrollar este IDE y plugins que permiten ampliar su funcionalidad para diversos requerimientos.

Para el desarrollo del libro utilizaré la versión 6.1¹; el archivo de instalación se puede obtener desde la página de la comunidad y sus características son:

Sitio Web	http://www.netbeans.org
Archivo	netbeans-6.1-windows.exe
Tamaño	182.27 MB

Su instalación es sencilla, solo hay que seguir los pasos del asistente y en unos cuantos minutos tendremos NetBeans instalado.

Esta versión de NetBeans instala Java y tiene incorporado el servidor GlassFish que es un contenedor de aplicaciones web con Servlets y JSP, por lo tanto no necesitamos de otro servidor para hacer nuestras pruebas.

En la Figura 2.1 podemos ver la interfaz de NetBeans, en la vista **Services** podemos apreciar dentro de la carpeta **Servers** que ya contamos con **GlassFish**, sin embargo podemos utilizar otros servidores como por ejemplo Tomcat, JBoss, etc.

¹ Es posible que en estos momentos que usted está leyendo éste texto exista una nueva versión.

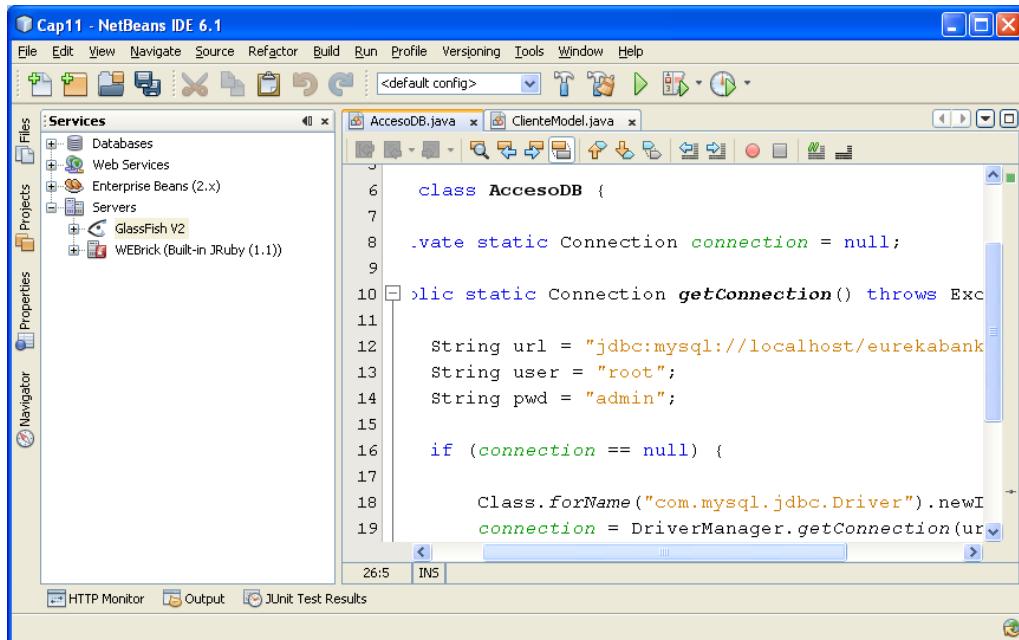


Figura 2 . 1 Interfaz de NetBeans

2.2. MySQL

2.2.1. Software

En los ejemplos donde necesitemos una base de datos utilizaremos MySQL, esta es una de las bases de datos más utilizadas en aplicaciones web, desarrolladas con Java y otros lenguajes como PHP.

Los datos para descargar el archivo instalador son:

Sitio Web	http://www.mysql.com
Archivo	MySQL-5.0.41-Win32-Setup.exe
Tamaño	81.3 MB

2.2.2. Instalación

- Como primer paso debe ejecutar el archivo MySQL-5.0.41-Win32-Setup.exe, y obtendrá la siguiente ventana de bienvenida:



Figura 2 . 2 Ventana inicial de instalación de MySQL

2. En la ventana de la Figura 2.2 debe hacer clic sobre el botón **Next**, y obtendrá la ventana donde seleccionará el tipo de instalación, tal como se muestra a continuación:



Figura 2 . 3 Ventana donde tiene que seleccionar el tipo de instalación.

Para nuestro caso seleccionaremos **Typical**, que es la opción por defecto seleccionada.

3. En la ventana de la Figura 2.3, después de seleccionar el tipo de instalación **Typical**, haga clic sobre el botón **Next**, y obtendrá la ventana donde se muestra información de los parámetros de instalación, tal como se ilustra a continuación:

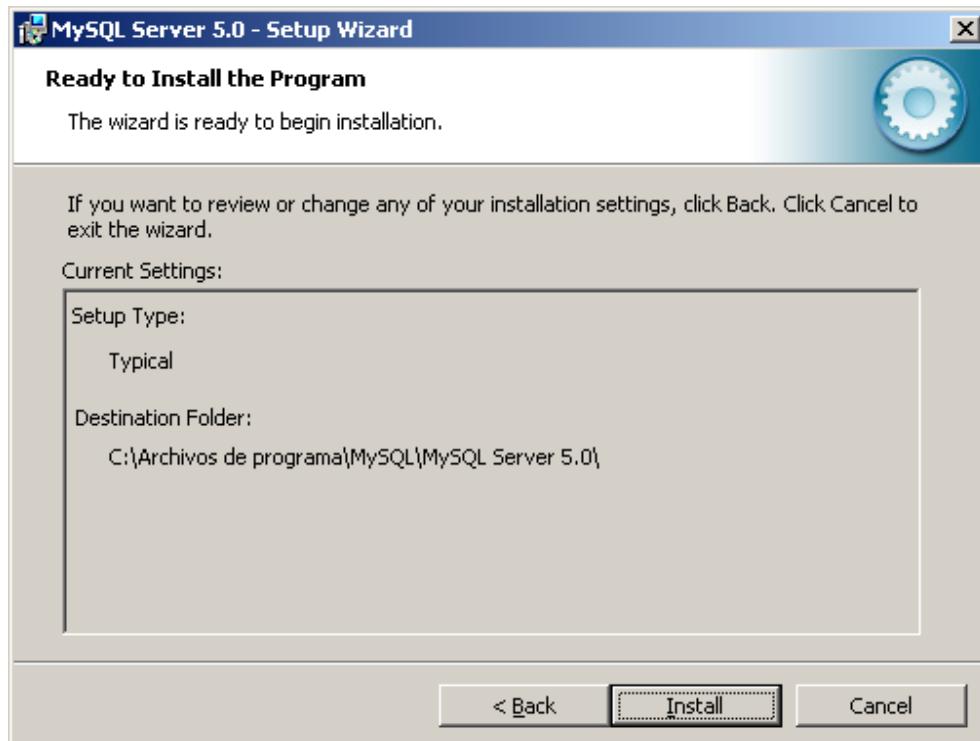


Figura 2 . 4 Ventana con información sobre los parámetros de instalación.

4. En la ventana de la Figura 2.4 debe hacer clic sobre el botón **Install** para iniciar la instalación, tal como se muestra a continuación:



Figura 2 . 5 Ventana que muestra el proceso de instalación.

5. Luego de terminar el proceso de instalación, se muestra la siguiente ventana indicándole que la instalación ha terminado.



Figura 2 . 6 Ventana que muestra la finalización de la instalación de MySQL.

En esta ventana tenemos la opción de finalizar la instalación y ejecutar el asistente para la configuración del servidor.

El asistente para la configuración del servidor se puede ejecutar en cualquier otro momento desde el menú:

Iniciar/Programas/MySQL/MySQL Server 5.0/MySQL Server Instance Config Wizard

6. En la ventana de la Figura 2.6 deje marcado el casilla **Configure the MySQL Server now** y haga clic sobre el botón **Finish**, se iniciará el asistente para configurar el servidor, tal como se ilustra en la siguiente ventana:

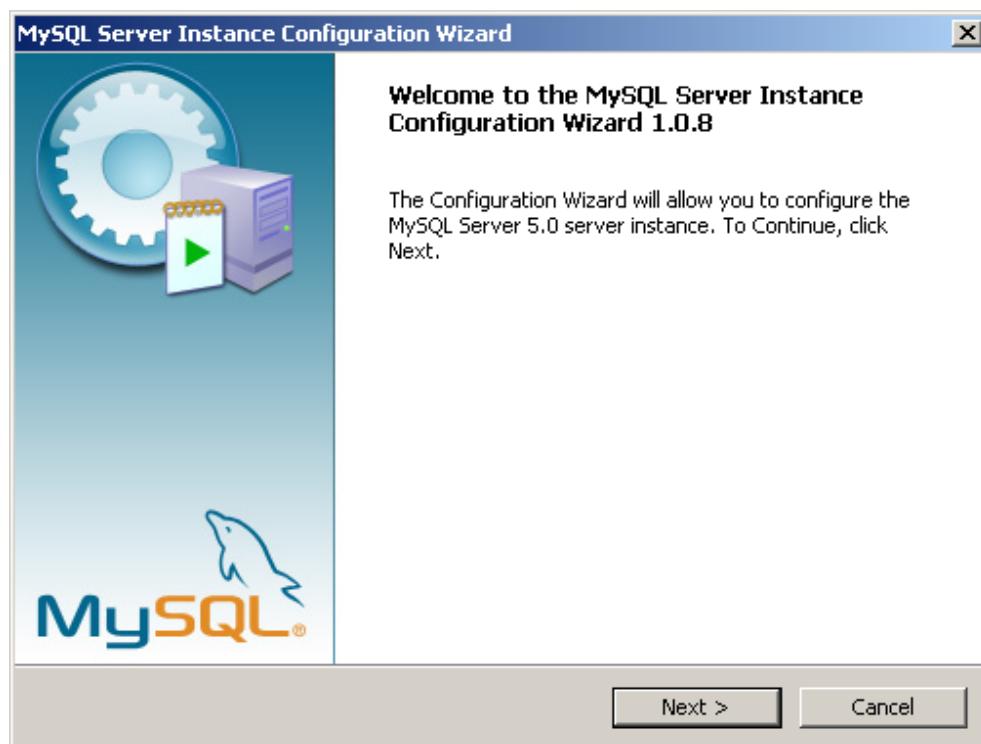


Figura 2 . 7 Inicio del asistente para configurar el servidor.

7. En la ventana de la Figura 2.7 haga clic sobre el botón **Next**, y obtendrá la siguiente ventana:

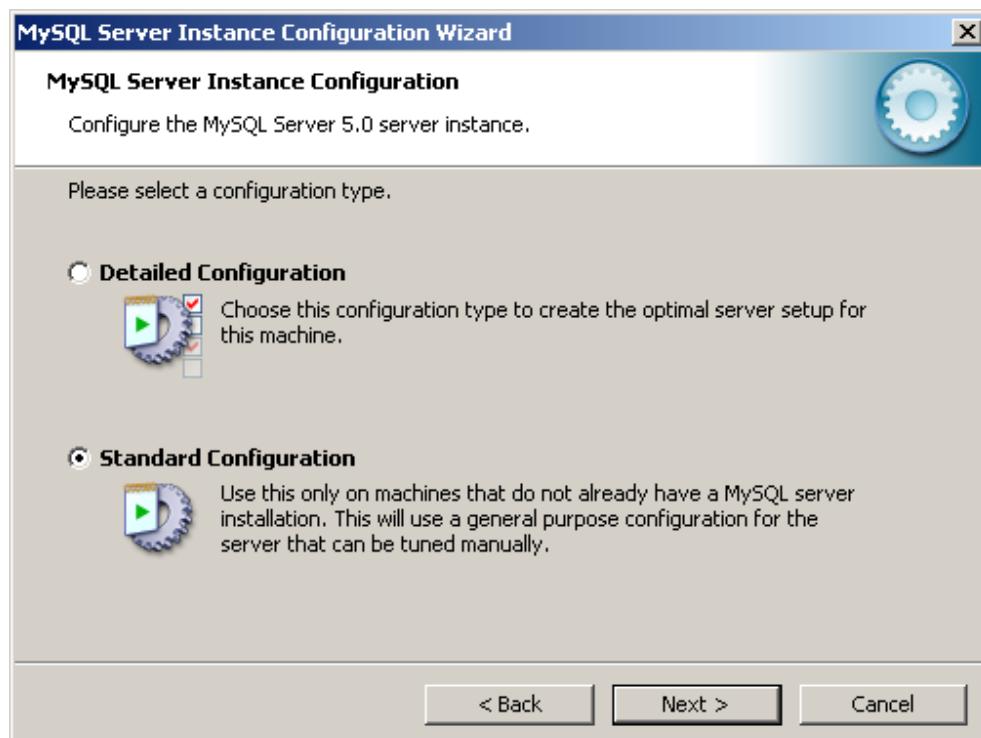


Figura 2 . 8 Ventana para seleccionar el tipo de configuración del MySQL.

En esta ventana debe seleccionar el tipo de configuración; la opción **Standard Configuration** se debe utilizar en equipos que no tienen instalado el servidor, en aquellos equipos donde ya está instalado y configurado MySQL debe utilizar la opción **Detailed Configuration**.

8. En la ventana de la Figura 2.8 seleccione la opción **Standar Configuration** y haga clic sobre el botón **Next**, y obtendrá la siguiente ventana:



Figura 2 . 9 Ventana de opciones de configuración de Windows.

En esta ventana tenemos las opciones de configuración de Windows, lo más importante es el nombre del servicio, que por defecto es **MySQL**.

9. En la ventana de la Figura 2.9 deje las opciones por defecto y haga clic sobre el botón **Next**, obtendrá la siguiente ventana:

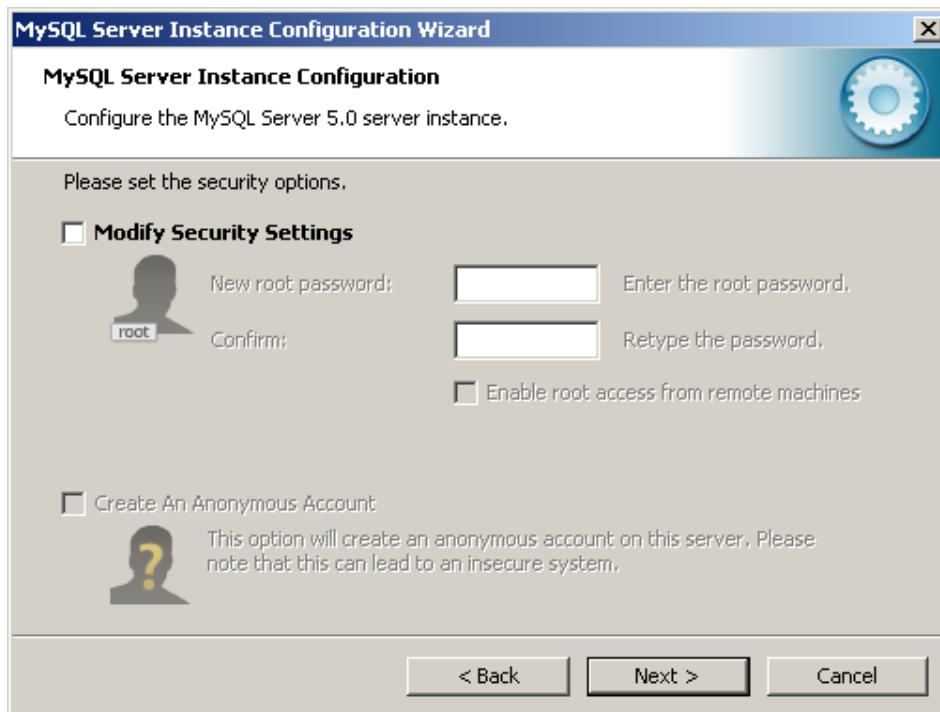


Figura 2 . 10 Ventana de configuración de opciones de seguridad.

MySQL tiene un usuario administrador (super usuario) de nombre **root**, por defecto no tiene contraseña, si queremos modificar la contraseña de root debemos habilitar la casilla **Modify Security Settings**.

10. En la Ventana de la Figura 2.10 deshabilite la casilla **Modify Security Settings** y haga clic sobre el botón **Next**, obtendrá la siguiente ventana:

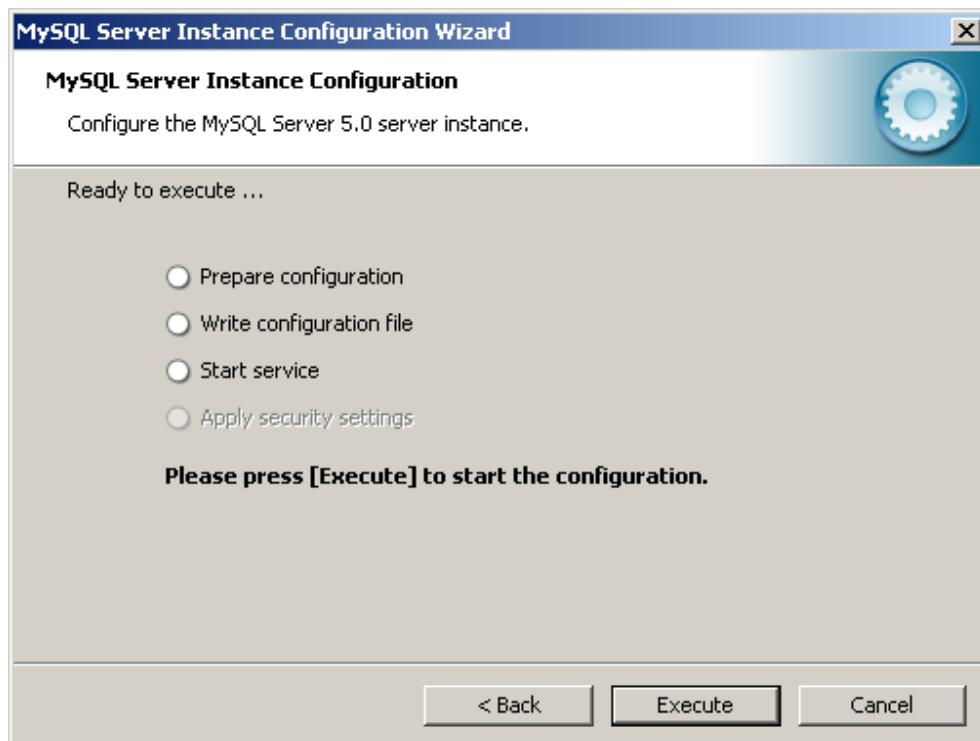


Figura 2 . 11 Ventana que muestra los pasos a seguir en la configuración del servidor.

Esta ventana muestra los pasos que se ejecutarán para configurar el servidor de MySQL.

11. En la ventana de la Figura 2.11 haga clic sobre el botón **Execute** para ejecutar la configuración del servidor MySQL; se supone que no debe tener problema alguno, por lo tanto obtendrá la siguiente ventana:

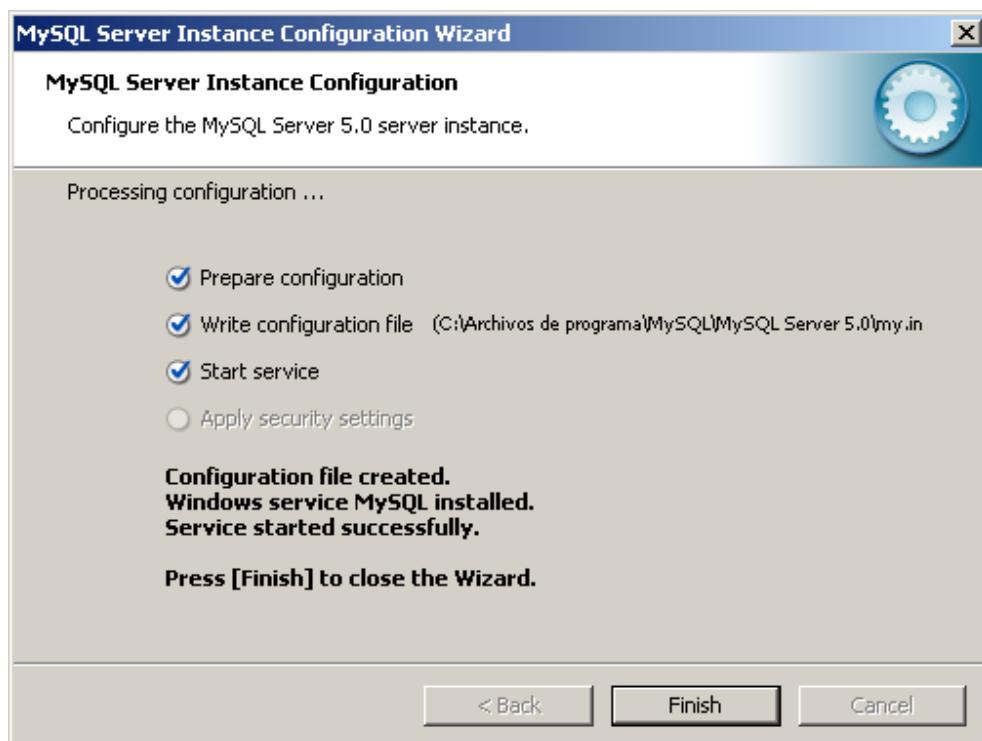


Figura 2 . 12 Ventana que muestra el estado de la configuración del servidor.

12. Para finalizar con la configuración de MySQL, haga clic sobre el botón **Finish** en la ventana de la Figura 2.12.

2.2.3. Trabajando con MySQL

Para empezar a trabajar con MySQL debe cargar la consola de trabajo, lo puede hacer desde la opción de menú:

Inicio/Programas/MySQL/MySQL Server 5.0/MySQL Command Line Client

Se muestra la siguiente ventana:

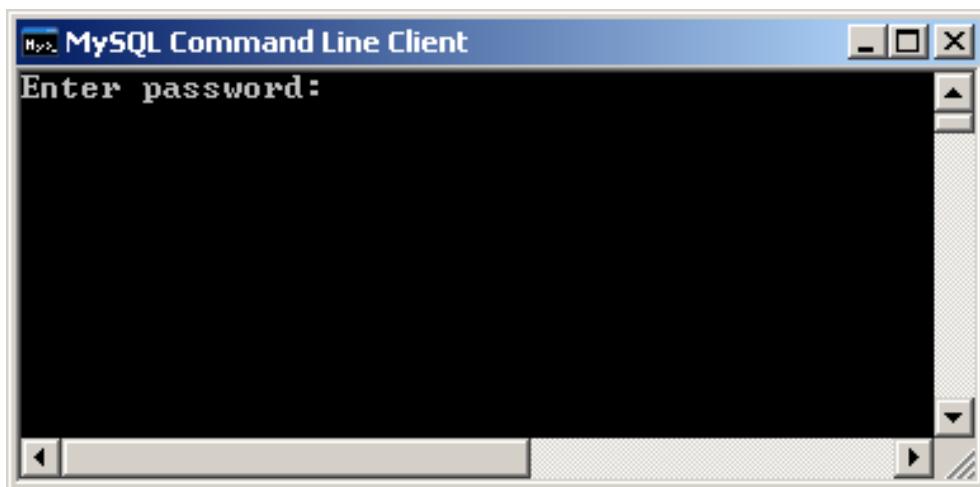


Figura 2 . 13 Consola de trabajo de MySQL.

Por defecto se inicia con el usuario **root**, por lo tanto, lo que se solicita en esta ventana es la contraseña del usuario **root**, pero como no le hemos asignado contraseña basta con presionar la tecla [Enter], se obtiene la siguiente ventana:

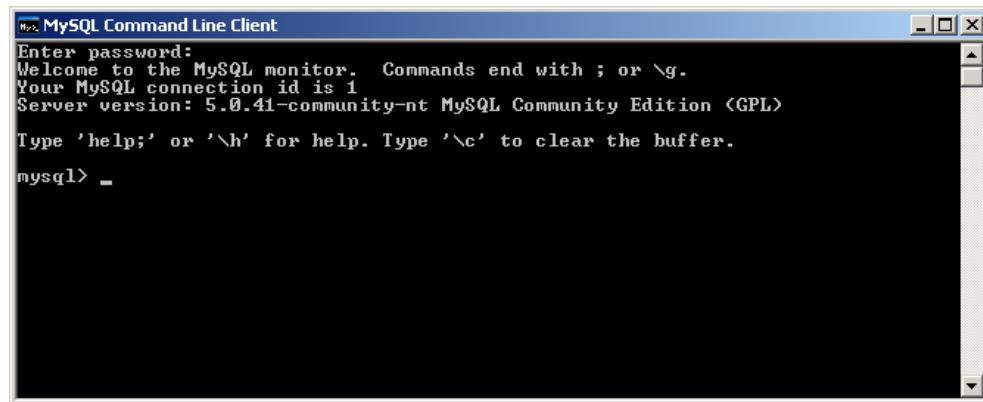


Figura 2 . 14 Consola de trabajo de MySQL.

Esta consola de comandos es la que debe utilizar para trabajar con el servidor de base de datos MySQL.



Clases, Campos y Métodos

En este capítulo desarrollaremos los primeros conceptos de la POO. Veremos los conceptos de clase, atributo y operación usando notación UML; luego su implementación utilizando Java, donde se conoce como clase, campo y método.

Los puntos a tratar son:

- 3.1. Definición de una Clase
- 3.2. Trabajando con Campos
- 3.3. Trabajando con Métodos
- 3.4. Ejemplo Demostrativo

3.1. Definición de una Clase

3.1.1. Definición de una Clase

Una clase define un tipo de objeto en particular.

Por ejemplo, en una empresa la clase `Factura` define a todas las facturas que la empresa emite.

Su sintaxis en Java es:

```
[ModificadorClase] class NombreClase {  
    // Campos  
    // Métodos  
}
```

El ModificadorClase se resume en el siguiente cuadro:

PALABRA CLAVE	DESCRIPCIÓN
public	Define una clase como pública; en caso contrario su visibilidad será solamente a nivel del paquete donde se encuentra.
abstract	Esta palabra clave determina que la clase no puede ser instanciada; es decir no podemos crear objetos de este tipo de clase. Este tipo de clase solo puede ser heredada.
final	Esta palabra clave determina que la clase no puede ser heredada, quiere decir que no podemos crear subclases de este tipo de clase. Este tipo de clase solo puede ser instanciada.

No es posible definir una clase como abstract y final a la vez.

Ejemplo 3 . 1

```
class Factura {

    // Campos
    private int numero;
    private double importe;

    // Métodos
    public void grabar( . . . ) {

        . . .
        . . .

    }
} // Factura
```

3.1.2. Representación UML de una Clase

Para tener una representación gráfica de una clase se utiliza UML (Unified Modeling Language), tal como se ilustra en la Figura 3.1.

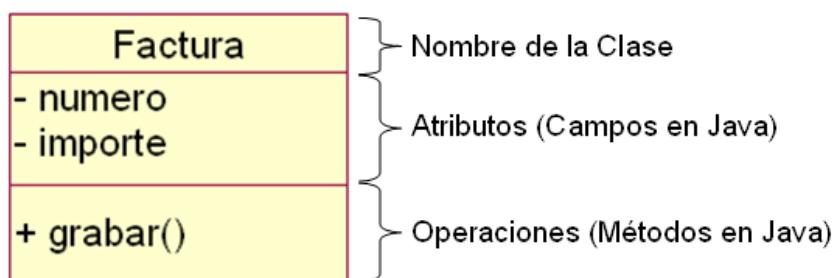


Figura 3 . 1 Representación UML de una clase.

3.1.3. Declaración de Objetos

El operador new se utiliza para crear un objeto de un tipo de clase específica, es decir, que asigne memoria para el objeto.

Para utilizar el operador new tenemos dos sintaxis.

Sintaxis 1

```
NombreDeClase nombreDeVariable;
nombreDeVariable = new NombreDeClase();
```

En la primera instrucción se define la variable que apuntará a un objeto que se crea en la segunda instrucción.

Sintaxis 2

```
NombreDeClase nombreDeVariable = new NombreDeClase();
```

En este caso tanto la creación de la variable y la creación del objeto se realizan en la misma instrucción.

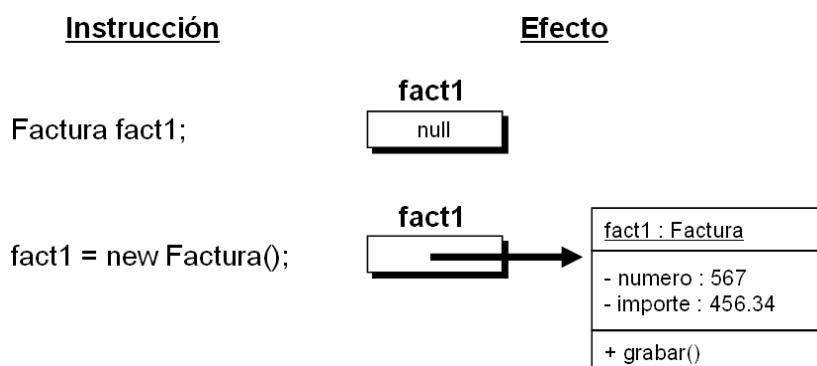


Figura 3 . 2 Instanciación de una clase (Creación de un objeto).

3.1.4. Asignación de Objetos

Cuando creamos un objeto internamente existe un puntero al que no tenemos acceso, pero cuando asignamos objetos, lo que realmente estamos asignando son las direcciones de memoria donde están definidos los objetos, como se explica en la Figura 3.3.

```
Factura fact1 = new Factura();
Factura fact2 = fact1;
```

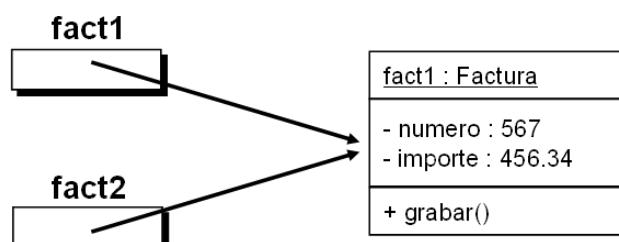


Figura 3 . 3 Asignación de objetos.

3.2. Trabajando con Campos

3.2.1. Definición

Los campos definen los datos de un objeto; cuando definimos una clase debemos definirlos y establecer su visibilidad:

La visibilidad determina desde qué parte del código se puede acceder al campo.

Sintaxis:

```
[TipoAcceso][ModificadorCampo] Tipo nombreCampo [= ValorInicial];
```

El TipoAcceso puede tomar los valores que se resume en el siguiente cuadro:

PALABRA CLAVE	DESCRIPCIÓN	UML
public	Indica que el campo es de acceso público. El acceso se realiza a través de una instancia del objeto.	+
protected	Indica que solo se puede acceder al campo desde métodos de la misma clase y clases derivadas (subclases)	#
sin especificar	Indica acceso de paquete ¹ , se puede acceder al campo a través de una instancia, pero sólo desde clases que se encuentren en el mismo paquete.	~
private	Indica que solo se puede acceder al campo desde métodos de la misma clase.	-

El ModificadorCampo puede tomar los valores que se resumen en el siguiente cuadro:

PALABRA CLAVE	DESCRIPCIÓN
static	El campo pertenece a la clase, no a los objetos creados a partir de ella.
final	El campo es una constante, en ese caso debe tener un valor inicial obligatoriamente. Por convenio en java las constantes se escriben en mayúsculas.
transient	Marca el campo como transitorio, para no ser serializado. Se usa cuando se trabaja con java beans.
volatile	Es un campo accedido de forma asíncrona mediante hilos, con este modificador se lo notificamos a java.

El ValorInicial permite inicializar la variable con un valor.

De acuerdo a la forma en que se especifica un atributo, objetos de otras clases tienen distintas posibilidades de accederlos, tal como se resume en el siguiente cuadro:

ACCESO DESDE:	PRIVATE	(PACKAGE)	PROTECTED	PUBLIC
La propia clase	Si	Si	Si	Si
Otras clases en el mismo paquete	No	Si	Si	Si
Subclase en el mismo paquete	No	Si	Si	Si
Subclases en otros paquetes	No	No	Si	Si
Otras clases en otros paquetes	No	No	No	Si

3.2.2. Ocultando los Datos

Uno de los fundamentos de la programación orientada a objetos es que el usuario solo debe tener acceso a los datos que le son de su interés, y del modo que le corresponde, por ejemplo solo lectura, solo escritura, ó ambos.

Para conseguir esto se debe declarar los atributos como privados, y se debe implementar métodos para acceder a ellos. Existe un estándar para definir estos métodos, por ejemplo, si el campo es nombre los métodos son:

¹ Los paquetes permiten organizar clases según su naturaleza. El tema de paquetes será desarrollado en un capítulo posterior.

- **setNombre** Este método permite asignar un valor al campo.
- **getNombre** Este método permite leer el valor del campo.

Como puede apreciar existen dos prefijos, el prefijo `set` que se utiliza para asignar un valor al campo y el prefijo `get` para leer el valor del campo; de esta manera podemos seleccionar si se implementa el método `set`, `get`, o ambos, y restringir el nivel de acceso a los datos.

Otra posibilidad que nos da la implementación de estos métodos es de poder agregar funcionalidad que puede servir para verificar, por ejemplo, si el dato que se está signando es correcto o no.

Ejemplo 3 . 2

En este ejemplo se ilustra el uso de los métodos `set` y `get`.

```
class Factura {

    // Campos
    private int numero;

    // Métodos set y get
    public void setNumero( int numero ){
        this.numero = numero;
    }
    public int getNumero(){
        return this.numero;
    }

    . . .
    . . .

} // Factura
```

Como puede apreciar en el código, el método `setNumero` se utiliza para asignar un valor al campo `numero`, mientras que el método `getNumero` se utiliza para leer su valor.

Es posible agregar lógica en cualquiera de los dos métodos, o en ambos si el caso lo amerita.

3.3. Trabajando con Métodos

3.3.1. Definición

Los métodos definen el comportamiento de un objeto de una clase concreta y tienen las siguientes características:

- Agrupan las responsabilidades del objeto (competencias).
- Describen sus acciones.

Las acciones realizadas por un objeto son consecuencia directa de un estímulo externo (un mensaje enviado desde otro objeto) y dependen del estado del objeto.

El estado y comportamiento de un objeto están relacionados. Por ejemplo, un avión no puede aterrizar (acción) sino está en vuelo (estado).

Sintaxis:

```
[TipoAcceso] [ModificadorMétodo]
Tipo NombreMétodo ( [ ListaDeParámetros ] ) {
```

```
// Cuerpo de método  
}
```

El TipoAcceso puede tomar los valores que se resumen en el siguiente cuadro:

PALABRA CLAVE	DESCRIPCIÓN	UML
public	Indica que el método es de acceso público. El acceso se realiza a través de una instancia del objeto.	+
protected	Indica que solo se puede acceder al método desde métodos de la misma clase y clases derivadas (subclases)	#
sin especificar	Indica visibilidad de paquete, se puede acceder al método a través de una instancia, pero sólo desde clases que se encuentren en el mismo paquete.	~
private	Indica que solo se puede acceder al método desde métodos de la misma clase.	-

El ModificadorMétodo puede tomar los valores que se resumen en el siguiente cuadro:

Palabra Clave	Descripción
static	El método pertenece a la clase, no a los objetos creados a partir de ella.
final	El método es definido como método definitivo, esto quiere decir que no se puede redefinir en una subclase.
abstract	Determina que el método no se implementa en la clase, su implementación será en las clases heredadas o subclases.
synchronized	Permiten sincronizar varios threads para el caso en que dos o más accedan concurrentemente a los mismos datos.

El Tipo determina el tipo de dato que debe retornar el método, puede ser cualquier tipo válido, incluso los tipos de clases creados por el programador. Si el método no retorna ningún valor se debe especificar void.

Los parámetros son opcionales, pero si son necesarios se deben especificar de la siguiente manera:

```
tipo1 parámetro1, tipo2 parametro2, ... ;
```

Los parámetros son variables que reciben valores de los argumentos que se le pasan al método cuando éste es invocado.

Los métodos que son de un tipo diferente de void, deben retornan un valor a la instrucción que realiza la llamada mediante la instrucción return, utilizando la siguiente sintaxis:

```
return expresión;
```

Donde expresión representa el valor que retorna el método.

3.3.2. Sobrecarga de Métodos

Un método se identifica por su nombre, la cantidad de parámetros y el tipo de sus parámetros; esto constituye la firma del método (signature).

La sobrecarga permite tener varios métodos con igual nombre pero con diferente firma, o sea, con diferente cantidad de parámetros y diferentes tipos de datos de sus parámetros.

Los métodos tendrán comportamientos diferentes según el tipo de dato, cantidad de parámetros o ambos. Al invocar un método, el compilador busca en la clase el método que corresponda a los argumentos de la invocación.

Un ejemplo ilustrativo lo tenemos con el método `valueOf` de la clase `String`, tal como se ilustra en la Figura 3.4.

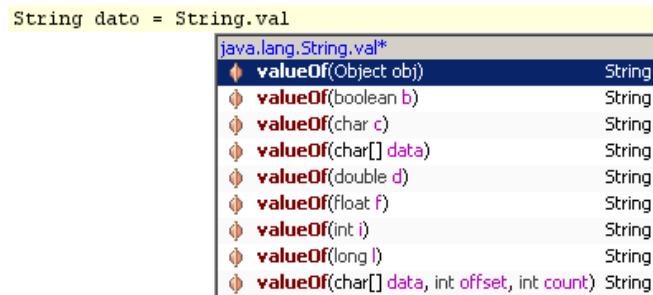


Figura 3 . 4 Método `valueOf` sobrecargado.

La Figura 3.5 muestra la representación gráfica de métodos sobrecargados de la clase `Clase1`.

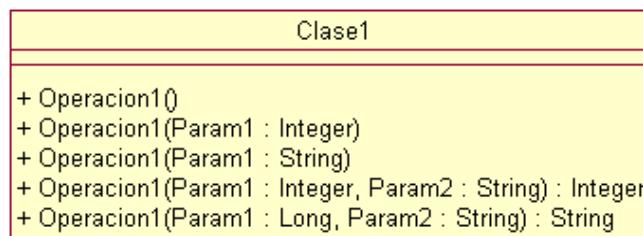


Figura 3 . 5 Representación UML de métodos sobrecargados.

A continuación tenemos su implementación en Java:

```
public class Clase1 {

    // Métodos sobrecargados
    public void Operacion1() {
        // Implementación
    }

    public void Operacion1(int Param1) {
        // Implementación
    }

    public void Operacion1(String Param1) {
        // Implementación
    }

    public int Operacion1(int Param1, String Param2) {
        // Implementación
        return 0;
    }

    public String Operacion1(long Param1, String Param2) {
        // Implementación
        return null;
    }
} // Clase1
```

Ejemplo 3 . 3

Supongamos que queremos un método que sume dos números, podrías necesitar que sume dos números de tipo `int` o `double`; entonces debemos implementar dos métodos con el mismo nombre por que se trata de la misma operación, podría ser `sumar`, pero con diferentes tipos de datos de sus parámetros, tal como se ilustra a continuación:

```
package egcc;

public class Matematica {

    public int sumar(int n1, int n2){
        int suma;
        suma = n1 + n2;
        return suma;
    }

    public double sumar(double n1, double n2){
        double suma;
        suma = n1 + n2;
        return suma;
    }

} // Matemática
```

Cuando usamos la clase `Matematica` el IDE son permite visualizar los diversos métodos que están sobrecargados, tal como se muestra en la siguiente figura:

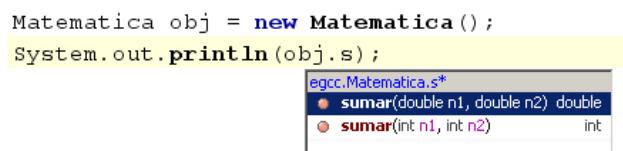


Figura 3 . 6 Visualización de la sobrecarga de métodos.

3.4. Ejemplo Demostrativo

3.4.1. Requerimientos del Software

Una institución financiera necesita de un programa que le permita encontrar el importe que deben pagar sus clientes por los préstamos que realiza, se sabe que se trata de un interés compuesto, capitalizable mensualmente.

La formula que debe aplicarse es:

$$M = C(1+i)^n$$

Donde:

- C : Capital
- i : Tasa de interés por periodo, por ejemplo puede ser mensual
- n : Número de periodos
- M : Importe acumulado en el número de periodos

3.4.2. Abstracción

A continuación tenemos algunas alternativas de abstraer el problema.

Caso 01

En este caso los atributos son públicos (visibilidad publica) por lo tanto se puede acceder de manera directa desde cualquier otra clase u objeto.

El método `calcularImporte` hace el cálculo en función al valor de los campos `capital`, `interes` y `periodo`.

Banco
+ capital : double
+ interes : double
+ periodo : int
+ calcularImporte()

Caso 02

Banco
+ calcularImporte(capital : double, interes : double, periodos : int) : Double

En este caso tenemos un solo método que tiene tres parámetros, a través de los cuales recibe los datos para hacer el cálculo y retorna el resultado correspondiente.

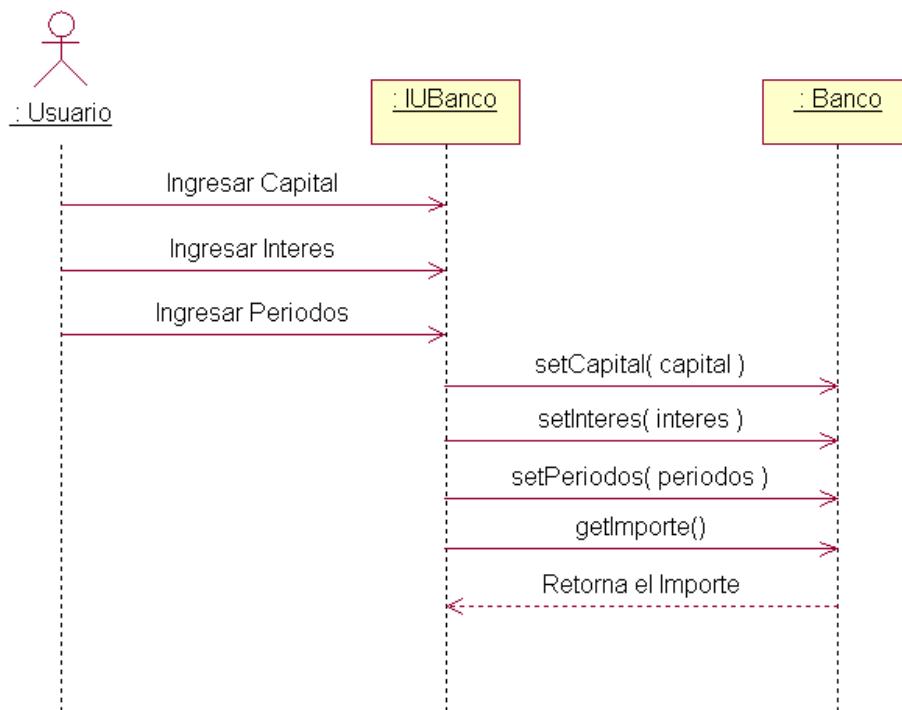
Caso 03

En este caso, los atributos son privados y se accede a ellos a través de los métodos `set` y `get`.

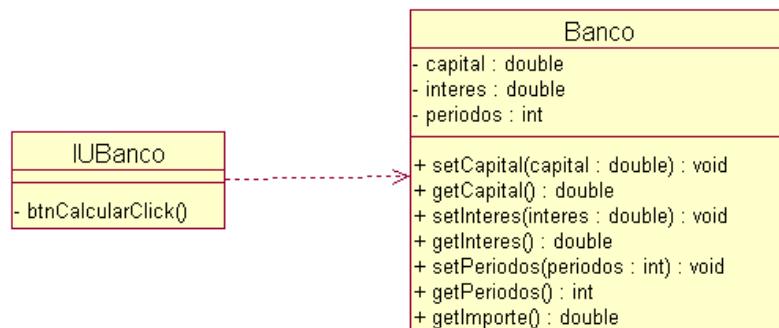
Para obtener el importe se debe utilizar el método `getImporte`, el cálculo se realiza en base a los campos respectivamente.

Banco
- capital : double
- interes : double
- periodos : int
+ setCapital(capital : double) : void
+ getCapital() : double
+ setInteres(interes : double) : void
+ getInteres() : double
+ setPeriodos(periodos : int) : void
+ getPeriodos() : int
+ getImporte() : double

3.4.3. Diagrama de Secuencia

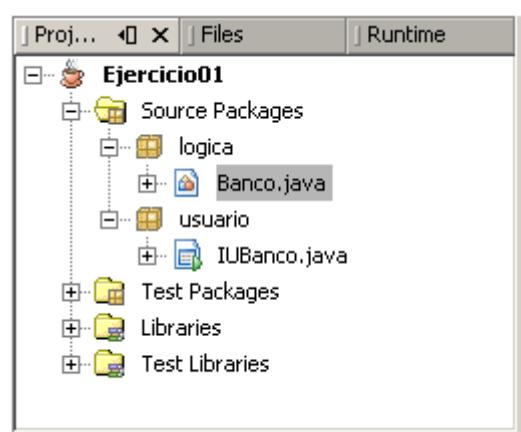


3.4.4. Diagrama de Clases



3.4.5. Estructura del Proyecto en NetBeans

Como se puede apreciar en el grafico de la derecha, tenemos dos paquetes, en el paquete **logica** ubicamos la clase **Banco**, y en el paquete **usuario** la clase **IUBanco**.



3.4.5.1. Codificación de Clase Banco

Banco
- capital : double - interes : double - periodos : int
+ setCapital(capital : double) : void + getCapital() : double + setInteres(interes : double) : void + getInteres() : double + setPeriodos(periodos : int) : void + getPeriodos() : int + getImporte() : double

```

package logica;

public class Banco {

    private double capital;
    private double interes;
    private int periodos;

    public void setCapital(double capital) {
        this.capital = capital;
    }

    public double getCapital() {
        return this.capital;
    }

    public void setInteres(double interes) {
        this.interes = interes;
    }

    public double getInteres() {
        return this.interes;
    }

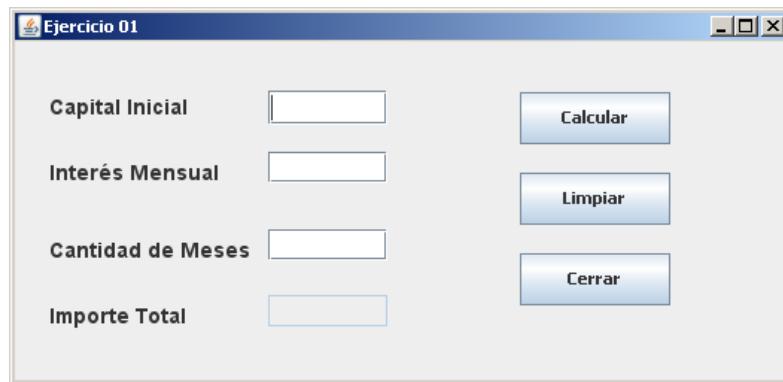
    public void setPeriodos(int periodos) {
        this.periodos = periodos;
    }

    public int getPeriodos() {
        return this.periodos;
    }

    public double getImporte() {
        double importe;
        importe = this.getCapital() *
            Math.pow(1 + this.getInteres(), this.getPeriodos());
        return importe;
    }
}
// Banco

```

3.4.5.2. Codificación de la Interfaz de Usuario IUBanco



Botón: btnCalcular

```
// Datos
double capital = Double.parseDouble(this.txtCapital.getText());
double interes = Double.parseDouble(this.txtInteres.getText());
int periodos = Integer.parseInt(this.txtPeriodos.getText());

// Instanciar la clase Banco
logica.Banco objBanco = new logica.Banco();

// Proceso
objBanco.setCapital(capital);
objBanco.setInteres(interes);
objBanco.setPeriodos(periodos);
double importe = objBanco.getImporte();

// Redondear el importe a dos decimales
importe = importe * 100;
importe = Math.round(importe);
importe = importe / 100;

// Presentar el resultado
this.txtImporte.setText(String.valueOf(importe));
```

Botón: btnLimpiar

```
this.txtCapital.setText("");
this.txtInteres.setText("");
this.txtPeriodos.setText("");
this.txtImporte.setText("");
```

Botón: btnCerrar

```
System.exit(0);
```



Constructores y Destructores

Toda clase está conformada por atributos y métodos; sin embargo, existen métodos especiales que permiten inicializar un objeto, y otro método especial que permite liberar los recursos que está utilizando un objeto.

Los puntos a tratar son:

- 4.1. Constructores
- 4.2. Destructores
- 4.3. Alcance de Instancia y de Clase

4.1. Constructores

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado. Esta inicialización se lleva a cabo invocando en un método especial denominado constructor. Esta invocación se realiza cuando se utiliza el operador `new`.

Los constructores tienen algunas características especiales:

- El nombre del constructor debe ser igual al de la clase.
- Puede recibir cualquier número de argumentos de cualquier tipo, como cualquier otro método.
- No se debe declarar ningún valor de retorno, ni siquiera `void`.

El constructor no es un miembro más de una clase. Sólo es invocado cuando se crea el objeto con el operador `new`. No puede invocarse explícitamente en ningún otro momento.

Al igual que los métodos, podemos sobrecargar el constructor de una clase, dependiendo de las necesidades de inicialización que se requieran para el objeto.

Si no creamos un constructor de manera explícita, se implementa automáticamente un constructor implícito sin parámetros.

Sintaxis

```
public NombreClase( [ lista_de_parámetros ] ) {  
    // Implementación  
}
```

En la Figura 3.1 tenemos la representación UML de una clase con dos constructores.

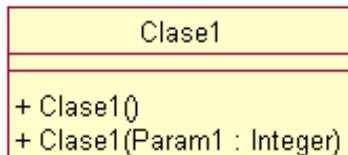


Figura 4 . 1 Representación UML de los constructores de una Clase.

A continuación tenemos su implementación en Java:

```
public class Clase1 {  
    // Constructores  
    public Clase1() {  
        // Implementación  
    }  
    public Clase1(int Param1) {  
        // Implementación  
    } // Clase1
```

Al crear un objeto debemos ejecutar un constructor, dependiendo de los parámetros que se le pasen se seleccionará y ejecutará el constructor que corresponda con la invocación, tal como se ilustra en la Figura 4.

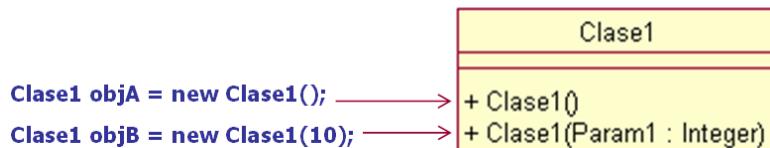


Figura 4 . 2 Ejecución de un constructor.

Ejemplo 4 . 1

Se necesita una clase para realizar operaciones con una base de datos Oracle; es requisito que en el instante de crear el objeto se conecte con la base de datos automáticamente.

La solución para este requerimiento es hacer una clase que en su constructor realice la conexión con la base de datos; la representación UML se muestra en la Figura 4.3.

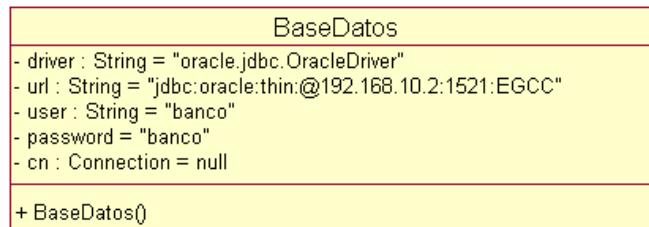


Figura 4 . 3 Representación de la clase BaseDatos.

A continuación tenemos su implementación en Java:

```
package logica;

import java.sql.Connection;
import java.sql.DriverManager;

public class BaseDatos {

    private String driver = "oracle.jdbc.OracleDriver";
    private String url = "jdbc:oracle:thin:@localhost:1521:EGCC";
    private String user = "banco";
    private String password = "banco";
    private Connection cn = null;

    public BaseDatos() throws Exception {
        try {
            Class.forName(driver);
            cn = DriverManager.getConnection(url, user, password);
        } catch (Exception e) {
            cn = null;
            throw e;
        }
    } // Constructor
} // BaseDatos
```

4.2. Destructores

Un destructor es un método especial que es ejecutado cada vez que se destruye (se elimina de la RAM) un objeto.

Es de visibilidad protegida (`protected`), no tiene parámetros, ni valor de retorno y se utiliza para liberar recursos utilizados por el propio objeto.

Java incorpora un método de nombre `finalize` que realiza la labor de destructor; está definido en la clase `Object`, si queremos implementarlo lo que realmente hacemos es sobre-escribirlo. Su sintaxis es:

```
protected void finalize() throws Throwable {
    // Implementación
}
```

Java implementa un **Recolector de Basura** (en inglés, **Garbage Collection**), que se encarga de eliminar los objetos de la memoria. Antes de que el Garbage Collection elimine el objeto de memoria ejecuta el destructor, o sea, el método `finalize`.

El Garbage Collection de la máquina virtual Java es el mecanismo que se encarga de reasignar la memoria de objetos que ya no estén siendo referenciados. Es transparente al programador, aunque existe `System.gc()`, pero no es más que una recomendación a la máquina virtual para que ejecute el Garbage Collection. Además, tiene un coste computacional importante; por lo general se recomienda dejar al sistema que decida cuando liberar memoria.

Esperar a que el Garbage Collection ejecute el método `finalize` para que se liberen los recursos utilizados por el objeto puede resultar muy costoso. Por ejemplo, en la clase `BaseDatos` del Ejemplo 4.1 podrías tener la conexión con la base de datos activa por mucho tiempo causando problemas de acceso a otros usuarios.

Es posible implementar un método público para que sea invocado cada vez que desea liberar recursos utilizados por el objeto; muchas veces este método es llamado `dispose`.

A continuación tenemos la representación UML de los métodos `finalize` y `dispose` para una clase:

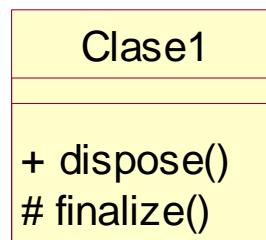


Figura 4 . 4 Representación UML de los métodos `finalize` y `dispose`.

Su implementación en Java podría ser así:

```
public class Clase1 {  
  
    private boolean liberar = true;  
  
    // Método para liberar recursos  
    public void dispose() {  
  
        // Implementación Liberar recursos  
  
        Liberar = false;  
    } // dispose  
  
    // Destructor  
    protected void finalize() throws Throwable {  
        if (liberar == false) return;  
        try {  
            this.dispose();  
        } catch (Exception e) {  
        } finally {  
            super.finalize();  
        }  
    } // finalize  
  
} // Clase1
```

Ejemplo 4 . 2

Se necesita implementar los métodos `finalize` y `dispose` de la clase `BaseDatos` del Ejemplo 4.1.

La representación UML se muestra en la Figura 4.5.

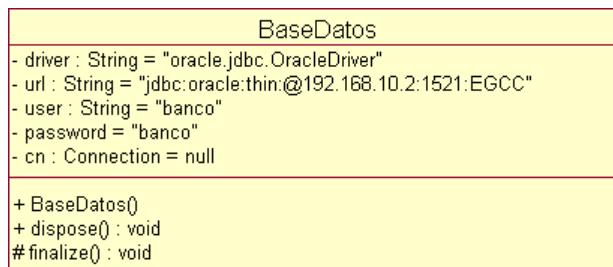


Figura 4 . 5 Representación UML de la clase BaseDatos.

A continuación tenemos su codificación en Java.

```

package logica;

import java.sql.Connection;
import java.sql.DriverManager;

public class BaseDatos {

    private String driver = "oracle.jdbc.OracleDriver";
    private String url = "jdbc:oracle:thin:@localhost:1521:EGCC";
    private String user = "banco";
    private String password = "banco";
    private Connection cn = null;

    public BaseDatos() throws Exception {
        try {
            Class.forName(driver);
            cn = DriverManager.getConnection(url, user, password);
        } catch (Exception e) {
            cn = null;
            throw e;
        }
    } // Constructor

    public void dispose() {
        try {
            if (!cn.isClosed())
                cn.close();
            cn = null;
        } catch (Exception e) {
        }
    } // dispose

    protected void finalize() throws Throwable {
        try {
            this.dispose();
        } catch (Exception e) {
        } finally {
            super.finalize();
        }
    } // finalize
}

} // BaseDatos

```

En la clase `BaseDatos` podemos implementar otros métodos para realizar procesos como consultas y transacciones, que se utilizarían de esta manera:

```
BaseDatos obj1 = new BaseDatos();
//
// Proceso
//
obj1.dispose();
```

Otra alternativa es usar la clase BaseDatos como base para crear otras clases especializadas en, por ejemplo, consultas y transacciones, tal como se ilustra en la Figura 4.6.

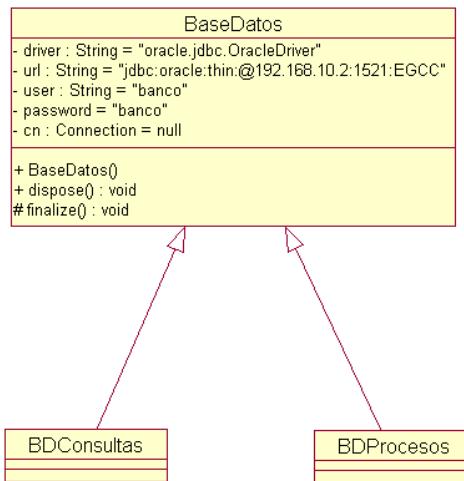


Figura 4 . 6 Especialización de la clase BaseDatos.

A continuación tenemos su implementación en Java:

Clase: BDConsulta

```
package logica;

public class BDConsulta extends BaseDatos {

    // Implementación
}
```

Clase: BDProceso

```
package logica;

public class BDProceso extends BaseDatos {

    // Implementación
}
```

4.3. Alcance de Instancia y de Clase

4.3.1. Alcance de Instancia

Son campos y métodos que necesitan ser invocados por una instancia, es decir, por un objeto, ya que se crean en el objeto.

```
nombreObjeto.nombreCampo  
nombreObjeto.nombreMétodo( ... )
```

4.3.2. Alcance de Clase

Son campos y métodos que no requieren crear una instancia (objeto) para ser invocados, basta con anteponer el nombre de la clase, ya que se encuentran en la clase y no en el objeto.

```
NombreClase.nombreCampo  
NombreClase.nombreMétodo( ... )
```

En Java esto es llamado campo y método estático (`static`), en el caso de ser un campo estático mantiene su valor para cada instancia de la clase; de ser público el campo podemos acceder incluso desde cualquier otra clase.

Su sintaxis es para los campos es:

```
[private|public|protected] static tipo nombreCampo [= valor];
```

Su sintaxis para los campos ed:

```
[private|public|protected] static tipo nombreMétodo( [parámetros] ) {  
    // Implementación  
}
```

Cuando crea una instancia de la clase, no se crean copias de las variables estáticas; en vez de ello, todas las instancias comparten las mismas variables estáticas, es decir el valor es compartido.

Los métodos de clase solo pueden acceder a campos y métodos de clase y no a campos y métodos de instancia, y no se puede usar la palabra `this` y `super` en un método estático. Un caso especial es el método estático `main` que se invoca automáticamente.

Un ejemplo de métodos de clase lo tenemos en la clase `Math`; no necesitamos instanciar la clase para utilizar sus métodos.

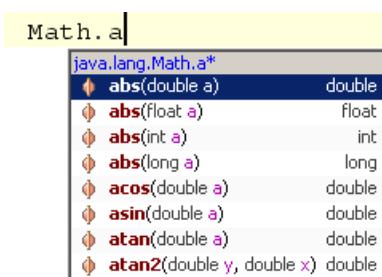


Figura 4 . 7 Acceso a métodos de clase de la clase Math.

A continuación tenemos la representación UML de una clase con campos y métodos de instancia y clase.

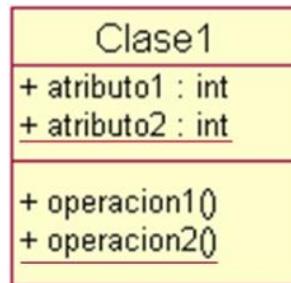


Figura 4 . 8 Alcance de Instancia y de Clase.

A continuación tenemos la implementación en Java:

```
public class Clase1 {  
  
    public int atributo1; // Alcance de instancia  
    public static int atributo2; // Alcance de clase  
  
    // Alcance de instancia  
    public void operacion1() {  
  
        //Implementación  
  
    } //  
    // Alcance de clase  
    public static void operacion2() {  
  
        //Implementación  
  
    }  
} // Clase1
```

4.3.3. Inicializador de Clase

También llamado inicializador estático, es similar a los constructores, pero para clases. Se diferencia del constructor en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los tipos primitivos pueden inicializarse directamente con asignaciones en la clase o en el constructor, pero para inicializar objetos o elementos más complicados es bueno utilizar un inicializador, ya que permite gestionar excepciones (situaciones de error) con try... catch.

Los inicializadores de clase se crean dentro de la clase, como métodos sin nombre, sin argumentos y sin valor de retorno; tan solo es la palabra `static` y el código entre llaves `{ ... }`, tal como se ilustra a continuación:

```
public class Clase1 {  
  
    . . .  
    . . .  
  
    static {  
  
        // Inicialización de campos estáticos
```

```

    }
} // Clase1

```

Ejemplo 4 . 3

Se requiere una clase con métodos de clase para las cuatro operaciones básicas, estas operaciones deben hacerse con dos números de tipo int.

La solución a este requerimiento se ilustra en la Figura 4.9.

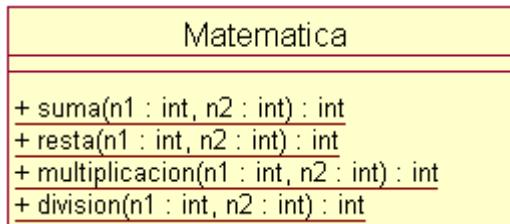


Figura 4 . 9 Clase con las cuatro operaciones básicas.

Su implementación en Java es:

```

package logica;

public class Matematica {

    public static int suma(int n1, int n2) {
        return (n1 + n2);
    }

    public static int resta(int n1, int n2) {
        return (n1 - n2);
    }

    public static int multiplicacion(int n1, int n2) {
        return (n1 * n2);
    }

    public static int division(int n1, int n2) {
        return (n1 / n2);
    }
} // Matematica

```

Para utilizar los métodos de la clase Matematica no será necesario instanciarla, solo debemos escribir el nombre de la clase, el operador punto y el método a utilizar, tal como se ilustra a continuación:

```
System.out.println( Matematica.suma(15,17) );
```

Página en blanco



Relaciones entre Clases

Todos los sistemas se construyen a partir de muchas clases y objetos cuya colaboración permite lograr el comportamiento deseado en el sistema.

La colaboración entre objetos impone la existencia de relaciones entre ellos: dependencia, generalización y asociaciones.

Los puntos a tratar son:

- 5.1. Introducción
- 5.2. Dependencia
- 5.3. Generalización
- 5.4. Asociación

5.1. Introducción

Las relaciones existentes entre las distintas clases de un sistema nos indican cómo se comunican los objetos de estas clases entre sí.

Los mensajes "navegan" por las relaciones existentes entre las distintas clases.

Existen 3 tipos de relaciones:

- Dependencia
- Generalización
- Asociación

Para hacer una representación gráfica de las clases se utilizan los diagramas de clases; donde, las dependencias se representan mediante una línea discontinua terminada en una punta de flecha, la generalización está representada por una línea continua terminada en un triángulo blanco, y la asociación es representada por una línea continua simple.

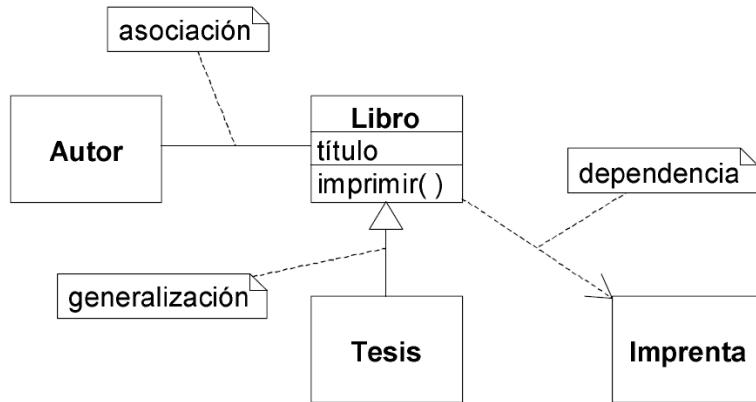


Figura 5 . 1 Diagrama de clases con sus tres relaciones básicas.

En la Figura 5.1 podemos ver un sencillo diagrama de clases con cuatro clases y tres relaciones. La clase **Libro** contiene el atributo **título** y la operación **imprimir()**, y tiene relaciones de asociación, generalización y dependencia con las clases **Autor**, **Tesis** e **Imprenta** respectivamente. La relación de asociación significa que las instancias de las clases **Autor** y **Libro** están relacionadas; la relación de generalización (o especialización, si se lee en sentido inverso) significa que el conjunto de instancias de la clase **Tesis** es un subconjunto del conjunto de instancias de la clase **Libro**; la relación de dependencia significa que la clase **Libro** depende de alguna manera de la clase **Imprenta**, por ejemplo, por que la operación **imprimir** requiere la especificación de una instancia de **Imprenta** como parámetro.

Un objeto representa una instancia particular de una clase, y tiene identidad y valores concretos para los atributos de su clase. La notación de los objetos se deriva de la notación de las clases, mediante el empleo de subrayado. Un objeto se dibuja como un rectángulo con dos secciones. La primera división contiene el nombre del objeto y de su clase, separados por el símbolo ":" y subrayados ambos, mientras que la segunda división, opcional, contiene la lista de atributos de la clase con los valores concretos que tiene en ese objeto.

Un enlace es una instancia de una asociación, del mismo modo que un objeto es una instancia de una clase. Un enlace se representa mediante una línea continua que une los objetos enlazados. Un **diagrama de objetos** es un grafo de instancias, es decir, objetos, valores de datos y enlaces. Un diagrama de objetos es una instancia de un diagrama de clases; muestra una especie de fotografía del estado del sistema en un sencillo instante de tiempo. En la Figura 5.2 podemos ver un sencillo ejemplo de diagrama de objetos correspondiente al diagrama de clases de la Figura 5.1.

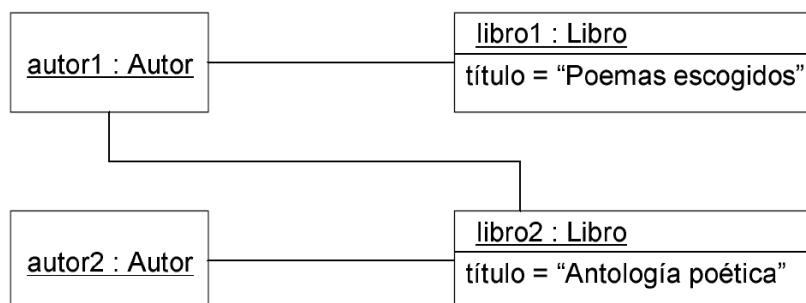


Figura 5 . 2 Diagrama de Objetos que muestra objetos, valores y enlaces.

En este diagrama podemos ver dos instancias de la clase **Autor**, dos instancias de la clase **Libro**, y tres instancias (es decir, enlaces) de la asociación **Autor-Libro**; estos enlaces significan que el primer autor es autor de dos libros, mientras que el segundo autor sólo es autor de un libro; a su vez, el primer libro tiene un autor y el segundo libro tiene dos autores.

Después de esta introducción pasaremos a ver cada uno de los tipos de relación.

5.2. Dependencia

Es una relación de uso, es decir una clase (dependiente) usa a otra que la necesita clase (independiente) para ejecutar algún proceso. Se representa con una flecha discontinua va desde la clase dependiente a la clase independiente.

v Con la dependencia mostramos que un cambio en la clase independiente puede afectar al funcionamiento de la clase dependiente, pero no al contrario.

Ejemplo 5 . 1

En este ejemplo haremos el diseño de las clases para resolver una ecuación de segundo grado:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

En la Figura 5.3 se muestra la relación de dependencia entre las clases Ecuación y Math.

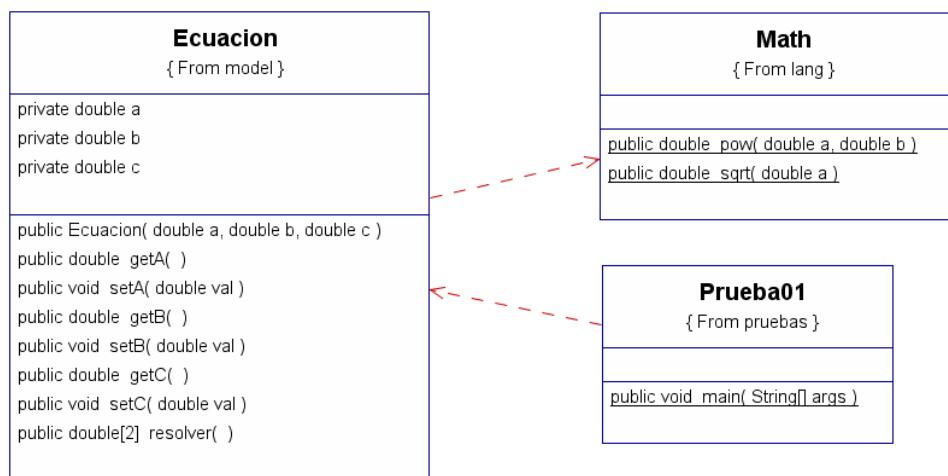


Figura 5 . 3 Relación de Dependencia.

La clase Ecuación depende de la clase Math porque necesita de los métodos pow y sqrt para realizar el cálculo de las raíces de la ecuación, por lo tanto la clase Math es la clase independiente y la clase Ecuación es la clase dependiente. También existe una dependencia de la clase Prueba01 y la clase Ecuación, para este caso la clase Ecuación es la clase independiente y la clase Prueba01 es la clase dependiente.

La clase Math se encuentra en el paquete java.lang, por lo tanto no necesita ser importada para poder utilizarla.

La codificación de la clase Ecuación es el siguiente:

```

package egcc.model;

public class Ecuacion {

    private double c;
    private double b;
    private double a;

    public Ecuacion (double a, double b, double c) {
        this.setA(a);
    }
  
```

```
        this.setB(b);
        this.setC(c);
    }

    public double getA () {
    return a;
    }

    public void setA (double val) {
this.a = val;
    }

    public double getB () {
    return b;
    }

    public void setB (double val) {
this.b = val;
    }

    public double getC () {
    return c;
    }

    public void setC (double val) {
this.c = val;
    }

    public double[] resolver () {
        double raiz[] = new double[2];
        double temp = Math.sqrt( Math.pow(this.getB(), 2) - 4 * this.getA() * this.getC() );
        raiz[0] = ( - this.getB() + temp ) / ( 2 * this.getA() );
        raiz[1] = ( - this.getB() - temp ) / ( 2 * this.getA() );
        return raiz;
    }

} // Ecuacion
```

La codificación de la clase Prueba01 es el siguiente:

```
package egcc.pruebas;

import egcc.model.Ecuacion;

public class Prueba01 {

    public static void main (String[] args) {

        // Valor de los coeficientes a, b, c
        Ecuacion obj = new Ecuacion( 1, -4, 3 );

        // Se obtienen las dos raíces
        double rpta[] = obj.resolver();

        // Se imprime el resultado
        System.out.println("X1 = " + rpta[0]);
        System.out.println("X2 = " + rpta[1]);
```

```

} // main

} // Prueba01

```

Si ejecutamos la clase Prueba01 obtenemos el resultado que se muestra en la Figura 5.4.

```

Output - Cap05 (run-single)
init:
deps-jar:
Created dir: F:\JavaN200\Cap05\build\classes
Compiling 1 source file to F:\JavaN200\Cap05\build\classes
compile-single:
run-single:
X1 = 3.0
X2 = 1.0
BUILD SUCCESSFUL (total time: 2 seconds)

```

Figura 5 . 4 Ejecución de la clase Prueba01.

5.3. Generalización

Muchos autores prefieren denominarla como **Generalización/Especialización**, por que se refieren a dos técnicas que nos llevan a obtener el mismo resultado, la herencia.

La Generalización consiste en factorizar las propiedades comunes de un conjunto de clases (clases hijas) en una clase más general (clase padre).

La Especialización es el proceso inverso a la Generalización; a partir de una clase denominada superclase se crean subclases que representan refinamientos de la superclase. Se añaden subclases para especializar el comportamiento de clases ya existentes.

En la Figura 5.5 tenemos la representación UML para representar la herencia. El mecanismo usado para implementar la generalización o especialización es la herencia, y los nombres usados para nombrar las clases son: superclase - subclase, clase padre - clase hija, clase base - clase derivada.

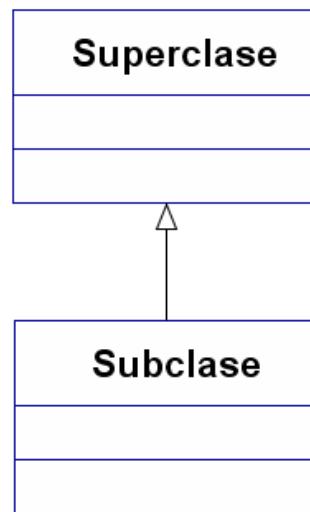


Figura 5 . 5 Notación UML para representar la herencia.

Las clases hijas heredan atributos y operaciones que están disponibles en sus clases padres.

En general, la herencia (Generalización o Especialización) es una técnica muy eficaz para la extensión y reutilización de código.

La implementación en Java se realiza utilizando la palabra clave `extends`, a continuación tenemos el script de la implementación del gráfico que se muestra en la Figura 5.5.

```
public class Superclase {  
}  
  
public class Subclase extends Superclase {  
}
```

Es importante recordar que toda clase en Java tiene que heredar de una clase padre, cuando no se indica explícitamente de que clase hereda, está heredando de la clase `Object` que se encuentra en el paquete `java.lang`.

Para nuestro caso la clase `Superclase` esta heredando de la clase `java.lang.Object`.

Ejemplo 5 . 2

En este ejemplo veremos un caso práctico, muy ilustrativo de como se puede aplicar la herencia. Tenemos una clase base de nombre `CalculadoraBase`, y una subclase de nombre `Calculadora1` que hereda de `CalculadoraBase`.

En la Figura 5.6 se puede apreciar que la subclase `Calculadora1` hereda los métodos `suma` y `resta`, y implementa los métodos `factorial` y `esPrimo`. Un aspecto importante es que los métodos están sobrecargados.

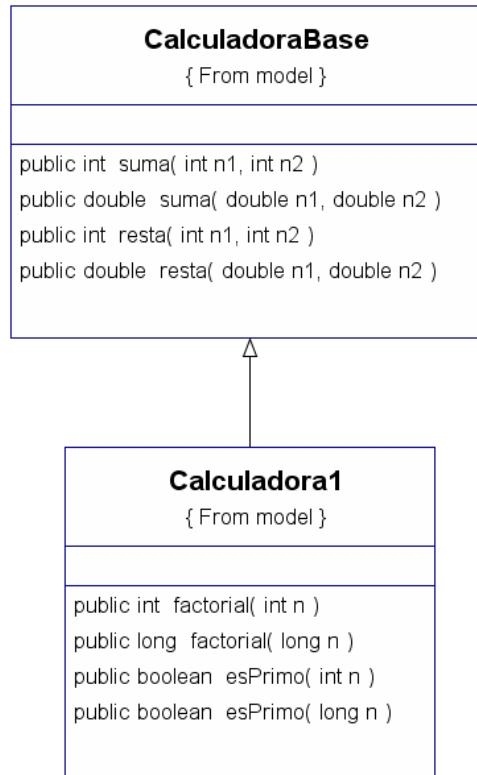


Figura 5 . 6 Representación UML de Herencia entre Clases.

Una instancia de la clase `CalculadoraBase` solo tendrá los métodos `suma` y `resta`, mientras que una instancia de la clase `Calculadora1` tendrá los métodos heredados `suma` y `resta`, y además sus métodos propios `factorial` y `esPrimo`.

A continuación tenemos la implementación de la clase CalculadoraBase:

```
package egcc.model;

public class CalculadoraBase {

    public int suma (int n1, int n2) {
        int rpta = n1 + n2;
        return rpta;
    }

    public double suma (double n1, double n2) {
        double rpta = n1 + n2;
        return rpta;
    }

    public int resta (int n1, int n2) {
        int rpta = n1 - n2;
        return rpta;
    }

    public double resta (double n1, double n2) {
        double rpta = n1 - n2;
        return rpta;
    }

} // CalculadoraBase
```

A continuación tenemos la implementación de la clase Calculadora1, la palabra extends en la definición de la clase indica que esta heredando de la clase CalculadoraBase, y puede usted verificar que solo se está implementando los nuevos métodos.

```
package egcc.model;

public class Calculadora1 extends CalculadoraBase {

    public int factorial(int n) {
        int f = 1;
        while (n > 1) {
            f = f * n--;
        }
        return f;
    }

    public long factorial(long n) {
        long f = 1;
        while (n > 1) {
            f = f * n--;
        }
        return f;
    }

    public boolean esPrimo(int n) {
        boolean primo = true;
        int k = 1;
        while (++k < n) {
            if (n % k == 0) {
                primo = false;
                break;
            }
        }
        return primo;
    }
}
```

```
        }
    }
    return primo;
}

public boolean esPrimo(long n) {
    Boolean primo = true;
    int k = 1;
    while (++k < n) {
        if (n % k == 0) {
            primo = false;
            break;
        }
    }
    return primo;
}

} // Calculadora1
```

A continuación tenemos la clase Prueba02 que nos permite probar la clase Calculadora1:

```
package egcc.pruebas;

import egcc.model.Calculadora1;

public class Prueba02 {

    public static void main(String[] args) {

        // Datos
        int n1 = 5;
        int n2 = 8;

        // Proceso
        Calculadora1 obj = new Calculadora1();
        int suma = obj.suma(n1, n2);
        int f1 = obj.factorial(n1);
        int f2 = obj.factorial(n2);

        // Reporte
        System.out.println("La suma es: " + suma);
        System.out.println("Factorial de " + n1 + " es: " + f1);
        System.out.println("Factorial de " + n2 + " es: " + f2);

    }

} // Prueba02
```

Note usted que desde un objeto de la clase Calculadora1 podemos acceder a los métodos definidos en la clase CalculadoraBase y en los suyos propios.

En la Figura 5.7 podemos ver el resultado de la ejecución de la clase Prueba02.

```
: Output - Cap05 (run-single)
init:
deps-jar:
compile-single:
run-single:
La suma es: 13
Factorial de 5 es: 120
Factorial de 8 es: 40320
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 5 . 7 Resultado de la ejecución de la clase Prueba02.

5.4. Asociación

La asociación se define como “una relación semántica entre dos o más clases que especifica conexiones entre las instancias de estas clases”.

Una instancia, a menudo es usada como sinónimo de objeto, es “una entidad que tiene identidad única, un conjunto de operaciones que se le pueden aplicar, y un estado que almacena los efectos de las operaciones”, y una clase es la “descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica”.

En un sistema informático orientado a objetos, los objetos existentes en un momento dado están conectados entre si y estas conexiones son descritas en el nivel abstracto de las clases mediante asociaciones.

5.4.1. Asociación Binaria

Una asociación binaria es una asociación entre dos clases. Se representa mediante una línea continua que conecta las dos clases asociadas: la línea puede ser horizontal, vertical, oblicua, curva, o estar formada por dos o más trazos unidos. El cruce de dos asociaciones que no están conectadas se puede mostrar con un pequeño semicírculo (como en los diagramas de circuitos eléctricos). La asociación en sí misma puede distinguirse de sus extremos (*association ends*), mediante los cuales se conecta a las dos clases asociadas. Las propiedades relevantes de una asociación pueden pertenecer a la asociación como tal, o a uno de sus extremos. En cada caso estas propiedades se representan mediante adornos gráficos (*graphical adornments*) situados en el centro de la asociación o en el extremo correspondiente (de tal forma que al mover o modificar la forma de una asociación en una herramienta CASE, el adorno debe desplazarse solidariamente con la asociación). Los adornos que podemos encontrar en una asociación binaria son:

5.4.1.1. Nombre de la asociación (association name)

Es opcional, y se representa mediante una cadena de caracteres situada junto al centro de la asociación, suficientemente separada de los extremos como para no ser confundida con un nombre de rol.

El nombre de la asociación puede contener un pequeño triángulo negro, denominado “dirección del nombre” (*name direction*), que apunta en la dirección en la que se debe leer la asociación, tal como se aprecia en la Figura 5.8.

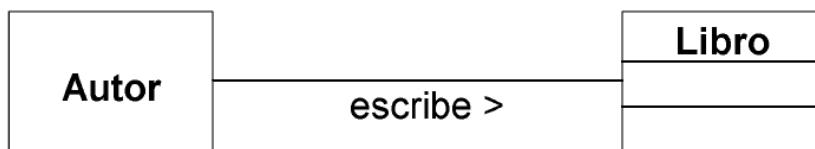


Figura 5 . 8 Ejemplo de asociación con nombre y dirección de nombre.

5.4.1.2. Nombre de rol (rolename)

Se representa mediante una cadena de caracteres situada junto a un extremo de la asociación, como se puede ver en la Figura 5.9. Es opcional, pero si se especifica en el modelo ya no puede ser omitido en las vistas. Índica el rol que juega en la asociación la clase unida a ese extremo.

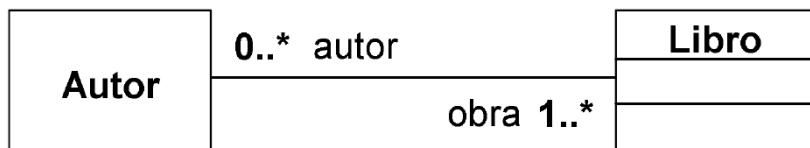


Figura 5 . 9 Ejemplo de asociación con nombres de rol y multiplicidades

Tanto el nombre de rol como la multiplicidad deben situarse cerca del extremo de la asociación, para que no se confundan con otra asociación distinta. Se pueden poner en cualquier lado de la línea: es tentador especificar que siempre se sitúen en el mismo lado (a mano derecha o a mano izquierda, según se mira desde la clase hacia la asociación), pero en un diagrama repleto de símbolos debe prevalecer la claridad. El nombre de rol y la multiplicidad pueden situarse en lados contrarios del mismo extremo de asociación, o juntos en el mismo lado.

5.4.1.3. Multiplicidad (multiplicity)

La multiplicidad de una relación determina cuántos objetos de cada clase (tipo de clase) intervienen en la asociación.

Cada asociación tiene dos multiplicidades, una a cada extremo de la asociación.

Para indicar la multiplicidad de una asociación hay que indicar la multiplicidad mínima y máxima utilizando el siguiente formato:

limite_inferior..limite_superior

Donde **limite_inferior** y **limite_superior** son valores enteros literales que especifican un intervalo cerrado de enteros. Además, el asterisco (*) se puede usar como límite superior para denotar un valor ilimitado (muchos).

Si se especifica un único valor, entonces el rango contiene sólo ese valor. Si la multiplicidad consiste en solo un asterisco, entonces denota el rango completo de los enteros no negativos, es decir, equivale a 0..* (cero o más). La multiplicidad 0..0 no tiene sentido, ya que indicaría que no puede haber ninguna instancia.

El siguiente cuadro muestra los diferentes tipos de multiplicidad de una asociación:

Multiplicidad	Significado
0..1	Cero o uno.
1	Uno y sólo uno.
0..*	Cero o muchos.
*	Cero o muchos.
1..*	Uno o muchos (al menos uno)
N..M	De N hasta M.

En la Figura 5.10 tenemos dos ejemplos de cardinalidad.



Un cliente tiene por lo menos una factura.
Una factura pertenece sólo a un cliente.



Una facultad tiene muchos profesores.
Un profesor pertenece a sólo una facultad.

Figura 5 . 10 Ejemplos de multiplicidad de una asociación.

5.4.1.4. Ordenación (ordering)

Si la multiplicidad es mayor que uno, entonces el conjunto de elementos relacionados puede estar ordenado o no ordenado (que es la opción por defecto si no se indica nada). La ordenación se especifica mediante la propiedad `{ordered}` en el extremo correspondiente.

La declaración de que un conjunto es ordenado no especifica cómo se establece o mantiene la ordenación, y en todo caso no se permiten elementos duplicados. Las operaciones que insertan nuevos elementos son responsables de especificar su posición, ya sea implícitamente (por ejemplo, al final) o explícitamente. La estructura de datos y el algoritmo empleados para la ordenación son detalles de implementación y decisiones de diseño que no añaden nueva semántica a la asociación.

La Figura 5.11 ilustra un ejemplo.

5.4.1.5. Modificabilidad (change ability)

Si los enlaces son modificables en un extremo (pueden ser añadidos, borrados o cambiados), entonces no se necesita ninguna indicación especial. Por el contrario, la propiedad `{frozen}` indica que no se pueden añadir, borrar ni cambiar enlaces de un objeto una vez que ha sido creado e inicializado. La propiedad `{addOnly}` indica que se pueden añadir enlaces, pero no se pueden borrar ni modificar los existentes. La Figura 5.11 ilustra un ejemplo estas propiedades.



Figura 5 . 11 Ejemplo de asociación con especificación de ordenación y modificabilidad.

5.4.1.6. Navegabilidad (navigability)

Una punta de flecha en el extremo de una asociación indica que es posible la navegación hacia la clase unida a dicho extremo (ver Figura 5.12). Se pueden poner flechas en uno, dos o ninguno de los extremos. Para ser totalmente explícito, se puede mostrar la flecha en todos los lugares donde la navegación es posible: en la práctica, no obstante, a menudo conviene suprimir algunas de las flechas y mostrar sólo las situaciones excepcionales. Si no se muestra la flecha en un extremo, no se puede inferir que la asociación no sea navegable en esa dirección: se trata simplemente de información omitida.

Conviene adoptar un estilo uniforme en la presentación de las flechas de navegabilidad en los diagramas: el estilo adoptado seguramente variará a lo largo del tiempo en función del tipo de diagrama que se trate. Algunos estilos posibles son:

- **Primer estilo:** mostrar todas las flechas. La ausencia de una flecha indica que la navegación no es posible.
- **Segundo estilo:** suprimir todas las flechas. No se puede inferir nada sobre la navegabilidad de las asociaciones, de igual modo que en otras situaciones hay determinada información que se suprime en una vista por conveniencia, no porque la información no exista.
- **Tercer estilo:** suprimir las flechas con navegabilidad en ambas direcciones, mostrar las flechas sólo para asociaciones con navegabilidad en una dirección. En este caso, no se puede distinguir la asociación navegable en las dos direcciones de la asociación que no es navegable en ninguna dirección, aunque este último caso es ciertamente muy raro en la práctica.



Figura 5 . 12 Ejemplo de asociación con especificación de navegabilidad y visibilidad

2.4.1.7. Visibilidad (visibility)

La visibilidad de un extremo de la asociación se indica mediante un signo especial o con un nombre de propiedad explícito delante del nombre de rol (ver Figura 5.12), y especifica la visibilidad de la asociación al transitarla en dirección hacia ese nombre de rol. Las posibles visibilidades son:

- (+) public
- (~) package
- (#) protected
- (-) private

La indicación de visibilidad puede ser suprimida, sin que eso signifique que la visibilidad está indefinida o es public, simplemente, no se desea mostrar en la vista actual.

Ejemplo 5 . 3

En el siguiente ejemplo ilustraremos el uso de asociación, para este caso contamos con dos clases: Article y Articles, tal como se muestra en el diagrama de clases de la Figura 5.13.

Un objeto de la clase Articles contiene una lista de objetos de la clase Article,

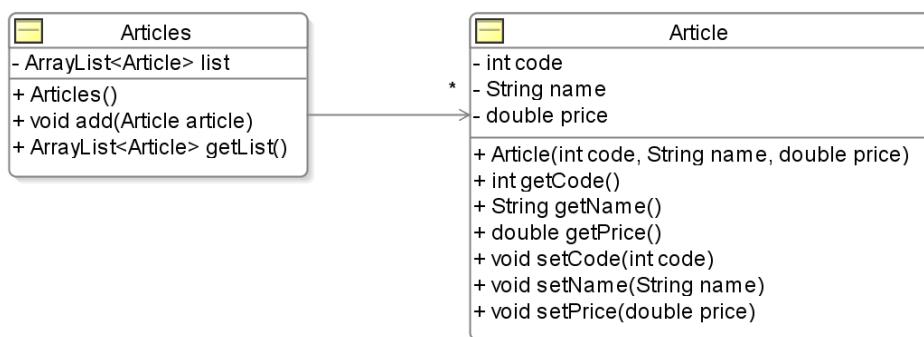


Figura 5 . 13 Diagrama de clases de la asociación entre Articles y Article.

A continuación tenemos la implementación de la clase Article:

```

package asociaciones;

public class Article {

    private int code;
  
```

```

private String name;
private double price;

public Article(int code, String name, double price) {
    this.setCode(code);
    this.setName(name);
    this.setPrice(price);
}

public int getCode() {
    return code;
}

public void setCode(int code) {
    this.code = code;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
}

} // Article

```

En la clase `Articles` estamos usando un `ArrayList`. La clase `ArrayList` puede contener una lista de objetos de cualquier tipo de clase, pero en esta oportunidad está parametrizado para que los objetos sean de tipo `Article`. A continuación tenemos la implementación de la clase `Articles`:

```

package asociaciones;

import java.util.ArrayList;

public class Articles {

    private ArrayList<Article> list;

    public Articles() {
        this.list = new ArrayList<Article>();
    }

    public ArrayList<Article> getList() {
        return this.list;
    }

    public void add(Article article) {
        this.list.add(article);
    }
}

```

```
} // Articles
```

Para ilustrar el uso de la clase `Articles` estamos creando una clase `Prueba`. El script de la clase `Prueba` es el siguiente:

```
package asociaciones;

import java.util.Iterator;

public class Prueba {

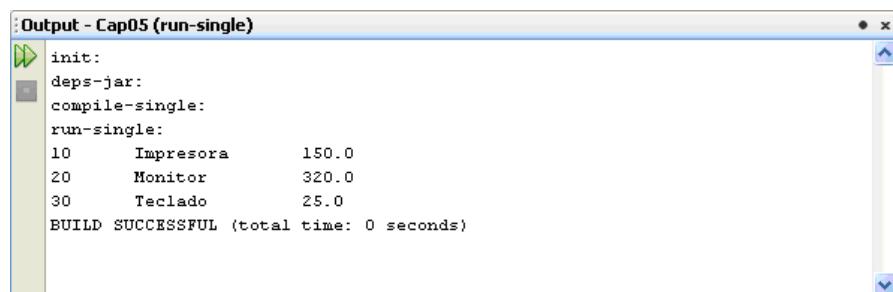
    public static void main(String[] args) {

        // Creamos la lista
        Articles lista = new Articles();
        Iterator it;
        Article art;

        // Le agregamos elementos
        lista.add( new Article(10,"Impresora",150.0) );
        lista.add( new Article(20,"Monitor",320.0) );
        lista.add( new Article(30,"Teclado",25.0) );

        // Listar la lista de elementos
        it = lista.getList().iterator();
        while( it.hasNext() ){
            art = (Article) it.next();
            System.out.println(art.getCode() + "\t" +
                art.getName() + "\t" + art.getPrice());
        }
    }
} // Prueba
```

El resultado de su ejecución es:



The screenshot shows the 'Output' window of an IDE. The title bar says 'Output - Cap05 (run-single)'. The window contains the following text:
init:
deps-jar:
compile-single:
run-single:
10 Impresora 150.0
20 Monitor 320.0
30 Teclado 25.0
BUILD SUCCESSFUL (total time: 0 seconds)

Figura 5 . 14 Resultado de la ejecución de la clase `Prueba`.

5.4.2. Agregación y Composición

La agregación es un tipo especial de asociación que se emplea para representar la relación entre un todo y sus partes. Se indica mediante un rombo blanco en el extremo de la asociación unido a la clase que representa el “todo”, también llamado “agregado” (aggregate) (ver Figura 5.14). Evidentemente, no se puede poner el rombo de agregación en los dos extremos de una asociación, ya que se trata de una relación antisimétrica. Si una asociación se define como agregación en el modelo, el símbolo de la agregación no se puede omitir en las vistas.

Dos o más agregaciones con el mismo “todo” se pueden dibujar en forma de árbol, juntando los extremos en uno solo, cuando coinciden las demás propiedades en el extremo agregado (multiplicidad, navegabilidad, etc.). La representación como árbol es meramente una opción de presentación, y no conlleva ninguna semántica especial.

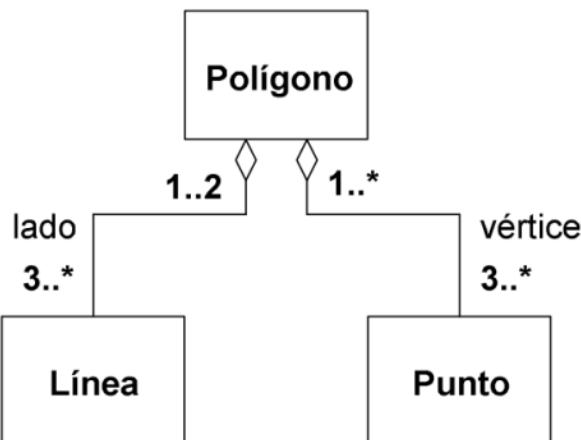


Figura 5 . 15 Ejemplo de dos agregaciones que no pueden dibujarse como un árbol porque no tienen la misma multiplicidad en el extremo agregado.

La composición es un tipo de agregación más fuerte, representada por un rombo negro en lugar de blanco, que se sitúa igualmente en el extremo unido al todo o “compuesto” (composite) (ver Figura 5.16). La composición implica que cada instancia-parte está incluida en un momento dado como máximo en una instancia-todo (compuesto), y que el compuesto tiene la responsabilidad exclusiva de la disposición de sus partes. La multiplicidad en el extremo compuesto no puede exceder de uno, es decir, las partes no se comparten entre distintos todos.

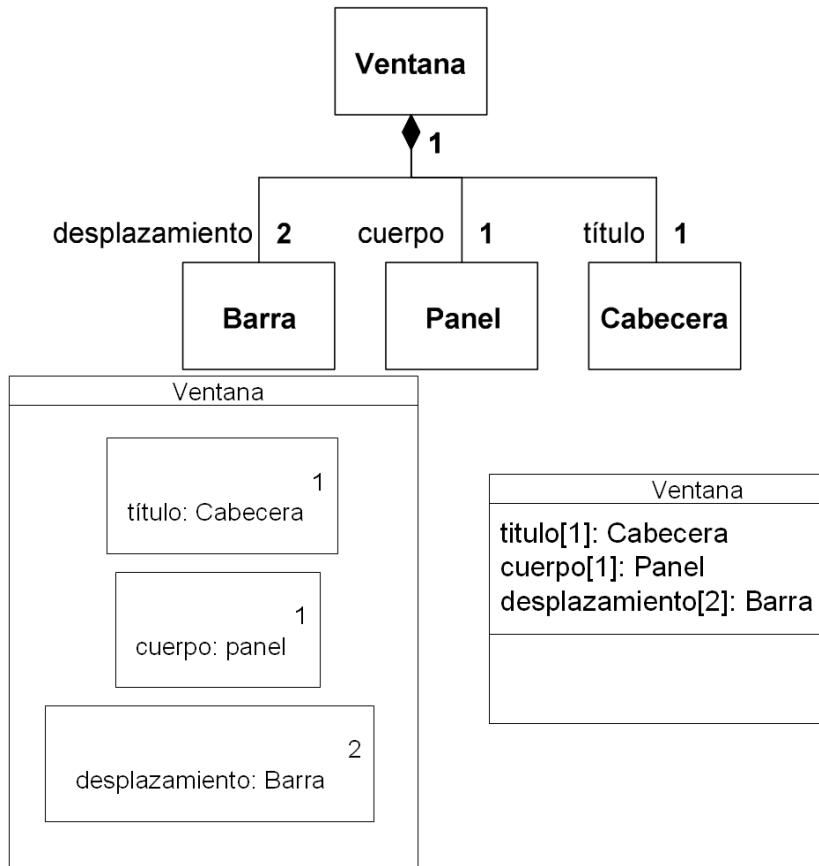


Figura 5 . 16 Ejemplo de composición representada en forma convencional (mediante asociaciones), en forma anidada, y en forma de atributos.

En lugar de usar líneas terminadas en rombos negros en la forma convencional, la composición se puede representar mediante el anidamiento de los símbolos de las partes dentro del símbolo del todo. La multiplicidad de la parte respecto al compuesto se muestra en la esquina superior derecha de la clase anidada que representa la parte, si se omite, se asume que es "muchos" por defecto. El nombre de rol de la parte respecto al todo se escribe delante del nombre de la clase anidada, separado por ":" (ver Figura 5.16). Nótese que la relación entre una clase y sus atributos es en la práctica una relación de composición.

Las partes de una composición pueden ser tanto clases como asociaciones. Una asociación como parte de un compuesto significa que tanto los objetos conectados como el propio enlace pertenecen todos al mismo compuesto. En la notación convencional, será necesario representar la asociación-parte como clase-asociación, para poder dibujar la composición con el todo mediante una línea terminada en rombo negro. Usando la notación anidada, una asociación entre dos partes dibujada dentro de los límites de la clase que representa el compuesto se considera que es parte de la composición. Por el contrario, si la asociación cruza el borde del compuesto, no será parte de la composición, y sus enlaces pueden establecerse entre partes de diferentes objetos compuestos.

5.4.3. Asociación n-aria

Una asociación n-aria es una asociación entre tres o más clases, alguna de las clases puede participar más de una vez, con roles distintos, en la asociación. Cada instancia de la asociación es una n-tupla de valores de las respectivas clases. Una asociación binaria se puede considerar que es un caso particular de asociación n-aria, con su propia notación.

Una asociación n-aria se representa mediante un rombo unido con una línea a cada clase participante (ver Figura 5.17). El nombre de la asociación se muestra junto al rombo. Cada extremo de la asociación puede tener adornos, como en una asociación binaria. En particular, se puede indicar la multiplicidad, pero ningún extremo puede tener cualificación ni agregación.

Se puede unir un símbolo de clase al rombo mediante una línea discontinua para indicar que se trata de una clase-asociación con atributos, operaciones o asociaciones con otras clases.

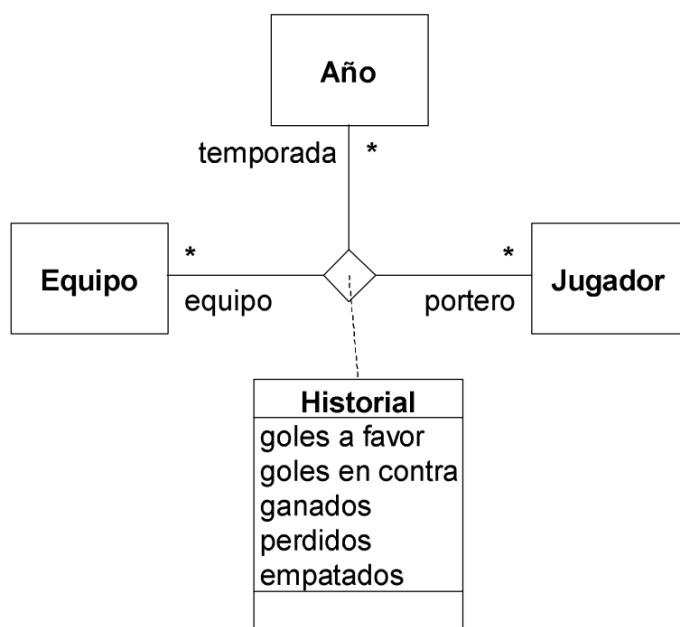


Figura 5 . 17 Una asociación ternaria que también es clase-asociación. En este ejemplo se muestra el historial de un equipo en cada temporada con un portero concreto. Se supone que el portero puede ser intercambiado durante la temporada y por tanto puede aparecer en varios equipos

Un enlace n-ario (instancia de asociación n-aria) se representa del mismo modo, mediante un rombo unido a cada una de las instancias participantes.

La multiplicidad se puede especificar en las asociaciones n-arias, pero su significado es menos obvio que la multiplicidad binaria. La multiplicidad de un rol representa el potencial número de instancias en la asociación cuando se fijan los otros N-1 valores.

Ejemplo 5 . 4

En este ejemplo veremos como implementar la agregación entre dos clases, para tal ilustración tomaremos como ejemplo las clases Banco y Cliente cuya relación se puede ver en la Figura 5.18.

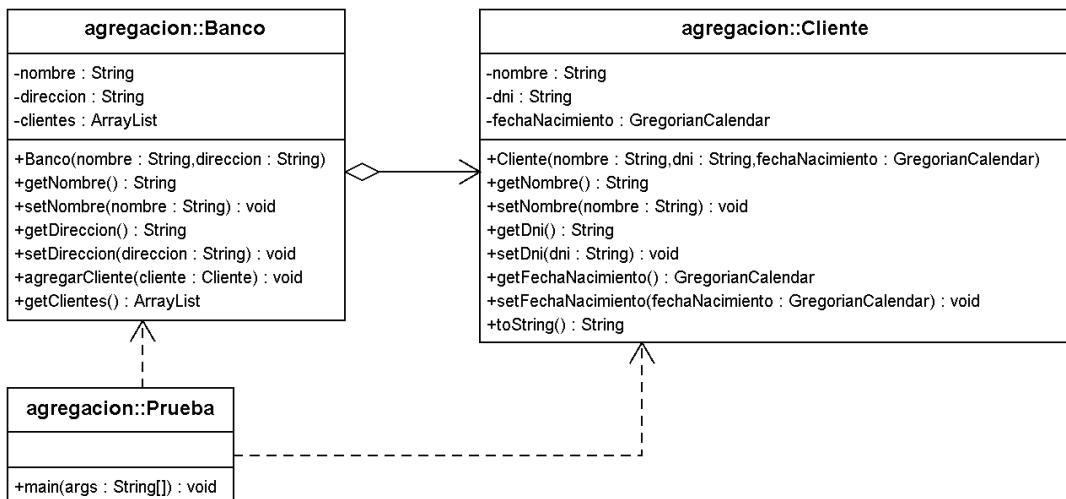


Figura 5 . 18 Diagrama de clases donde se ilustra la agregación.

Tanto el banco como el cliente tienen existencia independiente, pero el banco esta conformado por cliente. Un cliente puede serlo no solo de un banco, sino de varios bancos, y si un objeto de tipo Banco deja de existir los objetos de tipo Cliente no tienen que hacerlo, en eso radica su independencia.

A continuación tenemos el script de la clase Cliente:

```

package agregacion;

import java.util.GregorianCalendar;

public class Cliente {

    private String nombre = null;
    private String dni = null;
    private GregorianCalendar fechaNacimiento = null;

    public Cliente(String nombre, String dni, GregorianCalendar fechaNacimiento) {
        this.nombre = nombre;
        this.dni = dni;
        this.fechaNacimiento = fechaNacimiento;
    }

    public String getNombre() {
        return nombre;
    }
}
  
```

```
public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getDni() {
    return dni;
}

public void setDni(String dni) {
    this.dni = dni;
}

public GregorianCalendar getFechaNacimiento() {
    return fechaNacimiento;
}

public void setFechaNacimiento(GregorianCalendar fechaNacimiento) {
    this.fechaNacimiento = fechaNacimiento;
}

@Override
public String toString() {
    return this.getNombre() + "\t" +
        this.getDni() + "\t" + this.getFechaNacimiento().getTime();
}

} // Cliente
```

A continuación tenemos el script de la clase Banco:

```
package agregacion;

import java.util.ArrayList;

public class Banco {

    private String nombre = null;
    private String direccion = null;
    private ArrayList<Cliente> clientes = null;

    public Banco(String nombre, String direccion) {
        this.nombre = nombre;
        this.direccion = direccion;
        this.clientes = new ArrayList<Cliente>();
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDireccion() {
        return direccion;
    }
```

```

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }

    public void agregarCliente(Cliente cliente){
        this.clientes.add(cliente);
    }

    public ArrayList<Cliente> getClientes(){
        return this.clientes;
    }
}

} // Banco

```

A continuación tenemos el script de la clase Prueba:

```

package agregacion;

import java.util.GregorianCalendar;
import java.util.Iterator;

public class Prueba {

    public static void main(String[] args) {

        // Creación de Objetos
        Banco banco01 = new Banco("Santader", "España");
        Banco banco02 = new Banco("Credito", "Perú");
        Cliente clie01 = new Cliente("Gustavo", "06914897",
            new GregorianCalendar(1964, 3, 6) );
        Cliente clie02 = new Cliente("Claudia", "10156345",
            new GregorianCalendar(1970, 6, 15) );
        Cliente clie03 = new Cliente("Sergio", "11548227",
            new GregorianCalendar(1960, 2, 3) );

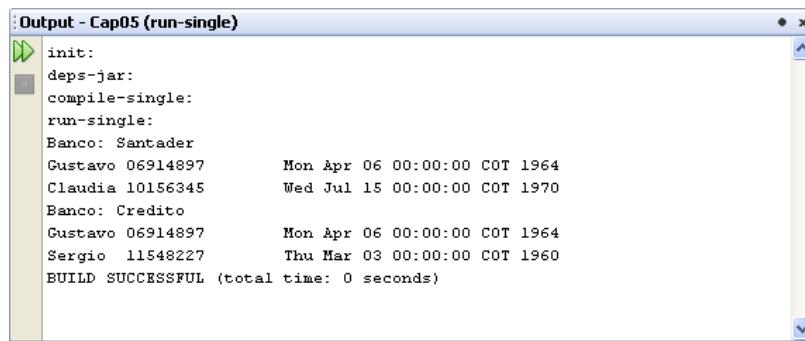
        // Agregamos los clientes al Banco
        banco01.agregarCliente(clie01);
        banco01.agregarCliente(clie02);
        banco02.agregarCliente(clie01);
        banco02.agregarCliente(clie03);

        // Imprimimos los clientes
        System.out.println("Banco: " + banco01.getNombre());
        Iterator it = banco01.getClientes().iterator();
        while( it.hasNext() ){
            Cliente obj = (Cliente) it.next();
            System.out.println(obj.toString());
        }
        System.out.println("Banco: " + banco02.getNombre());
        it = banco02.getClientes().iterator();
        while( it.hasNext() ){
            Cliente obj = (Cliente) it.next();
            System.out.println(obj.toString());
        }
    }
}

} // Prueba

```

El resultado de su ejecución se puede ver en la Figura 5.19.



```
:Output - Cap05 (run-single)
init:
deps-jar:
compile-single:
run-single:
Banco: Santader
Gustavo 06914897      Mon Apr 06 00:00:00 COT 1964
Claudia 10156345     Wed Jul 15 00:00:00 COT 1970
Banco: Credito
Gustavo 06914897      Mon Apr 06 00:00:00 COT 1964
Sergio 11548227       Thu Mar 03 00:00:00 COT 1960
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 5 . 19 Resultado de la ejecución de la clase Prueba.

Puede usted comprobar que el objeto `clie01` forma parte del objeto `banco01` y `banco02`, si el objeto `banco01` dejase de existir los objetos de tipo `Cliente` no lo tienen que hacer, por que el tiempo de vida de los objetos incluidos es independiente de objeto que lo incluye.

Si quisieramos imprimir los datos de todos los clientes podríamos utilizar el siguiente script, sin necesidad de pasar por algún objeto de tipo `Banco`:

```
System.out.println(clie01.toString());
System.out.println(clie02.toString());
System.out.println(clie03.toString());
```

Finalmente, si la relación fuese de tipo **composición** las cosas cambiarían, por que en una relación de este tipo el tiempo de vida del objeto incluido va a depender del tiempo de vida del objeto que lo incluye.



Java Collections Framework

Una tarea básica en el desarrollo de aplicaciones es la manipulación de grupos de objetos, como por ejemplo una lista de artículos (catalogo) recuperados de una base de datos; para este tipo de tareas Java proporciona un framework llamado **Java Collections Framework**.

Este framework esta compuesto por Interfaces, implementaciones y algoritmos que facilitan la manipulación de grupos de objetos.

Los puntos a tratar son:

- 6.1. Introducción
- 6.2. Elementos de JCF
- 6.3. Casos Prácticos

6.1. Introducción

Un marco de trabajo es un conjunto de interfaces y clases proporcionados para resolver un problema determinado. La intención es utilizar las clases proporcionadas, extenderlas o implementar las interfaces.

En la versión 1.2 del JDK se introdujo el Java Framework Collections o “estructura de colecciones de Java” (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la programación orientada a objetos. Dada la amplitud de Java en éste y en otros aspectos se va a optar por hacer una descripción general, dejando al lector la tarea de buscar las características concretas de los distintos métodos en la documentación de Java. En este capítulo se va a utilizar una forma sencilla de informar sobre los métodos disponibles en una clase o interfaz.

La colección de interfaces del core encapsulan diferentes tipos de colecciones, las cuáles se muestran en la Figura 6.1. Estas interfaces permiten colecciones que son manipuladas independientemente de los detalles que representan. La colección de interfaces del core son la base de JCF, y como usted puede ver en la Figura 6.1, forman una jerarquía de interfaces.

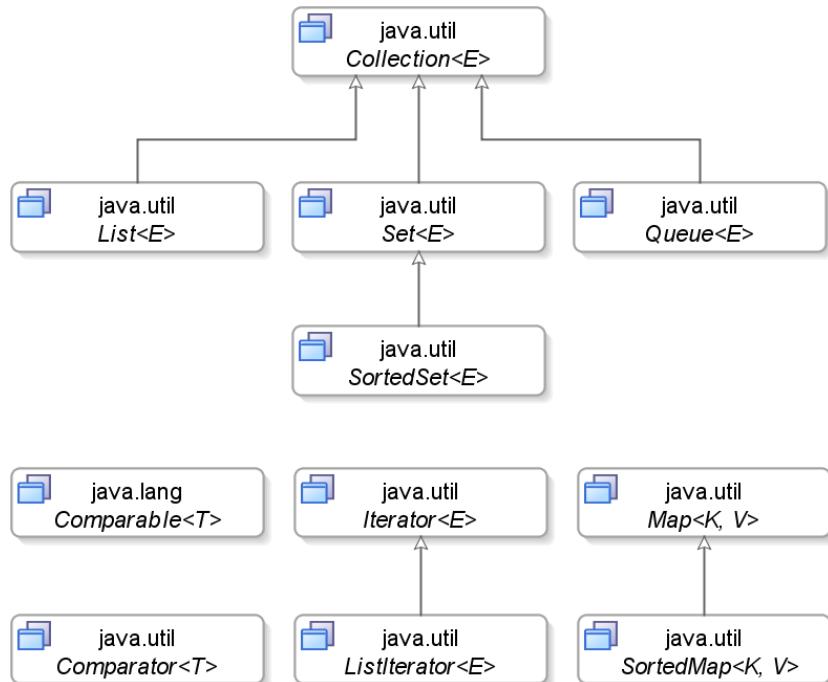


Figura 6 . 1 Colección de Interfaces del core de JCF.

En el diseño de JCF las interfaces son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interfaz se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases `ArrayList` y `LinkedList` disponen exactamente de los mismos métodos (Ver Figura 6.2) y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que `ArrayList` almacena los objetos en un array, la clase `LinkedList` los almacena en una lista vinculada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

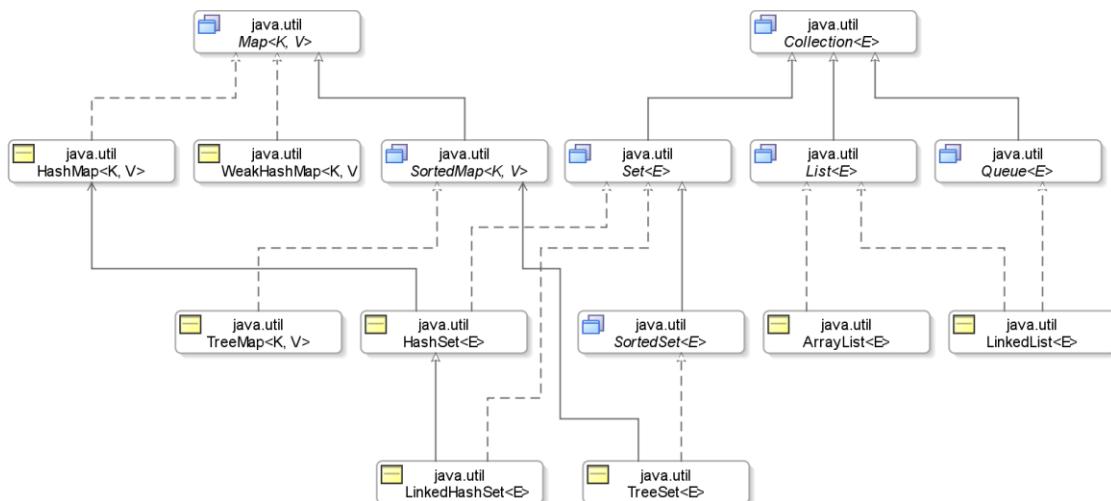


Figura 6 . 2 Clases que implementan las interfaces en JCF.

La Figura 6.3 muestra la jerarquía de clases de JCF. Puede usted observar que se muestran las clases abstractas y las clases concretas (clases de las que se pueden crear objetos), para tener un panorama mas amplio de como esta constituido en JCF.

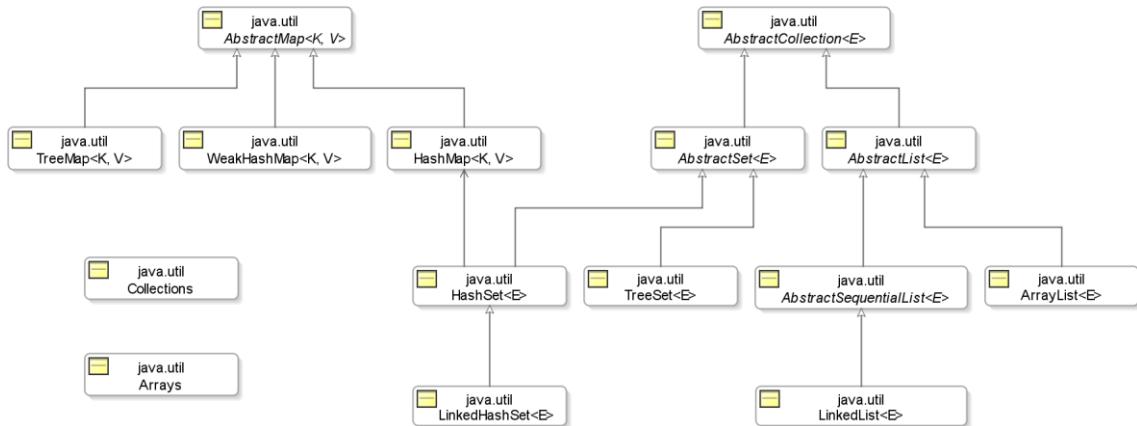


Figura 6 . 3 Diagrama de clases de JCF.

Las clases `Collections` y `Arrays` son especiales, no son abstractas, pero no tienen constructores públicos con los que se puedan crear objetos. Fundamentalmente contienen métodos de clase (`static`) para realizar ciertas operaciones como: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.

6.2. Elementos de JCF

6.2.1. Interfaces del core de JCF

En el siguiente cuadro tenemos un breve descripción de las interfaces fundamentales que constituyen el core del JCF.

INTERFAZ	DESCRIPCIÓN
Collection	Representa un grupo de objetos sin implementaciones directas, agrupa la funcionalidad general que todas las colecciones ofrecen.
Set	Colección que no puede tener objetos duplicados.
SortedSet	Set que mantiene los elementos ordenados
List	Colección ordenada que puede tener objetos duplicados
Map	Colección que enlaza claves y valores; no puede tener claves duplicadas y cada clave debe tener al menos un valor.
SortedMap	Map que mantiene las claves ordenadas.
Queue	Colección que maneja la prioridad para procesar los elementos

6.2.2. Interfaces de Soporte

En el siguiente cuadro tenemos una breve descripción de la interfaces que constituyen el soporte del core del JCF.

INTERFAZ	DESCRIPCIÓN
Iterator	Sustituye a la interfaz <code>Enumeration</code> . Dispone de métodos para recorrer una colección y para borrar elementos.
ListIterator	Deriva de <code>Iterator</code> y permite recorrer <code>lists</code> en ambos sentidos.
Comparable	Declara el método <code>compareTo()</code> que permite ordenar las distintas colecciones según un orden natural (<code>String</code> , <code>Date</code> , <code>Integer</code> , <code>Double</code> , ...).
Comparator	Declara el método <code>compare()</code> y se utiliza en lugar de <code>Comparable</code> cuando se desea ordenar objetos no estándar o sustituir a dicha interfaz.

6.2.3. Clases de propósito general

En el siguiente cuadro tenemos una breve descripción de las clases de propósito general del JCF.

CLASE	DESCRIPCIÓN
HashSet	Implementación de la interfaz <code>Set</code> mediante una hash table.
TreeSet	Implementación de la interfaz <code>SortedSet</code> mediante un árbol binario ordenado.
ArrayList	Implementación de la interfaz <code>List</code> mediante un array.
LinkedList	Implementación de la interfaz <code>List</code> mediante una lista vinculada.
HashMap	Implementa la interfaz <code>Map</code> mediante una hash table.
WeakHashMap	Implementa la interfaz <code>Map</code> de modo que la memoria de los pares clave/valor pueda ser liberada cuando las claves no tengan referencia desde el exterior de la <code>WeakHashMap</code> .
TreeMap	Implementa la interfaz <code>SortedMap</code> mediante un árbol binario.

6.2.4. Interfaz Comparable y Comparator

Interfaz Comparable.

Las clases que implementan esta interfaz cuentan con un “orden natural”. Este orden es total, es decir, siempre han de poder ordenarse dos objetos cualesquiera de la clase que implementa este interfaz. La interfaz `Comparable` declara el método `compareTo()` de la siguiente forma:

```
public int compareTo(Object obj)
```

Este método compara su argumento implícito (`this`) con el objeto que se le pasa como parámetro (`obj`), retorna un entero negativo, cero o positivo según que el argumento implícito (`this`) sea anterior, igual o posterior al objeto `obj`.

Las estructuras cuyos objetos tengan implementado el interfaz `Comparable` podrán ejecutar ciertos métodos basados en el orden tales como `sort` o `binarySearch`.

Si queremos programar el método `compareTo()` debemos hacerlo con cuidado, ha de ser coherente con el método `equals()` y ha de cumplir la propiedad transitiva.

Interfaz Comparator.

Si una clase ya tiene una ordenación natural y se desea realizar una ordenación diferente, por ejemplo descendente, dependiente de otros campos o simplemente requerimos varias formas de ordenar una clase, haremos que una clase distinta de la que va a ser ordenada implemente este interfaz.

Su principal método se declara en la forma:

```
public int compare(Object o1, Object o2)
```

El método `compare()` devuelve un entero negativo, cero o positivo según su primer argumento sea anterior, igual o posterior al segundo (Así asegura un orden ascendente).

Es muy importante que `compare()` sea compatible con el método `equals()` de los objetos que hay que mantener ordenados.

Los objetos que implementa esta interfaz pueden ser utilizados en las siguientes situaciones (especificando un orden distinto al natural):

- Como argumento a un constructor TreeSet o TreeMap, con la idea de que las mantengan ordenadas de acuerdo con dicho Comparator.
- Collections.sort(List, Comparator), Arrays.sort(Object[], Comparator)
- Collections.binarySearch(List, Object, Comparator),
Arrays.binarySearch(Object[] v, Object key, Comparator c)
Object también debe ser implementarlo.

¿Cuándo usar cada uno?

En una colección de objetos, éstos pueden ser ordenados por diferentes criterios. Dependiendo de la clase que realiza la comparación, se implementará la interfaz Comparator o Comparable.

Usaremos Comparable para definir el orden natural de una clase **C**, entendiendo por orden natural aquel que se utilizará normalmente o simplemente por convenio. Así, diremos que los objetos de clase **C** son comparables.

Por otro lado, implementaremos nuevas clases (**C1 ... Cn**) que extiendan el interfaz Comparator por cada ordenación nueva que necesitemos distinta a la natural para la clase **C**. Así tendremos una “librería de comparadores” (**C1 ... Cn**) para la clase **C**.

6.3. Casos Prácticos

6.3.1. Clase base

Para los casos que veremos utilizaremos la clase Producto como base construir la lista de objetos. La Figura 6.4 muestra la representación gráfica de la clase Producto.

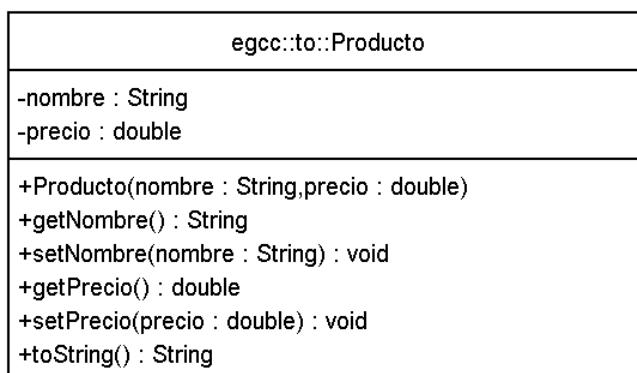


Figura 6 . 4 Clase Producto.

A continuación tenemos el script que implementa la clase Producto. Puede usted ver que se está sobre-escribiendo en método `toString()`.

El método `toString()` nos va ha permitir imprimir los datos de un artículo de una manera simple y rápida.

```
package egcc.to;

public class Producto {

    // Campos

    private String nombre = null;
    private double precio;
```

```
// Constructor

public Producto( String nombre, double precio ) {
    this.setNombre(nombre);
    this.setPrecio(precio);
} // Producto

// Métodos

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public double getPrecio() {
    return precio;
}

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

@Override
public String toString() {
    return this.getNombre() + " - " + this.getPrecio();
}

} // Producto
```

6.3.2. Manejo de listas mediante ArrayList

En este caso práctico vamos a ilustrar como manejar una lista de objetos mediante un `ArrayList`, el diagrama de clases se muestra en la Figura 6.5.

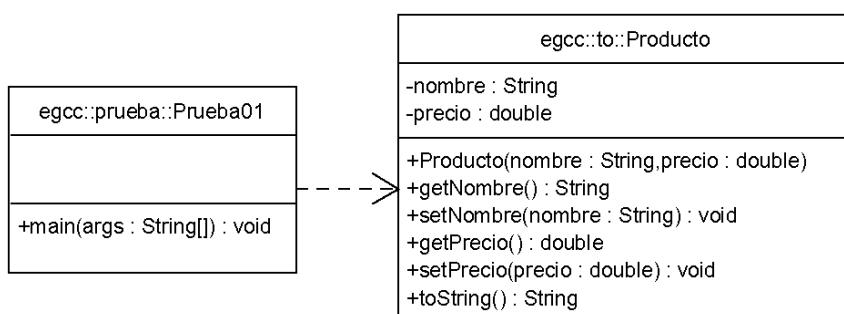


Figura 6 . 5 Diagrama de clases para el manejo de una lista de objetos mediante un `ArrayList`.

El script de la clase `Prueba01` es el siguiente:

```
package egcc.prueba;

import egcc.to.Producto;
import java.util.ArrayList;

public class Prueba01 {
```

```

public static void main(String[] args) {

    // Creación de la lista
    ArrayList lista = new ArrayList();

    // Creación de los productos
    Producto prod01 = new Producto( "Televisor", 650.0 );
    Producto prod02 = new Producto( "Impresora", 187.0 );
    Producto prod03 = new Producto( "Computadora", 2356.0 );
    Producto prod04 = new Producto( "Lavadora", 1348.0 );
    Producto prod05 = new Producto( "Cocina", 582.0 );

    // Agregar productos a la lista
    lista.add(prod01);
    lista.add(prod02);
    lista.add(prod03);
    lista.add(prod04);
    lista.add(prod05);
    lista.add(prod05);

    // Mostrar tamaño de la lista
    System.out.println("Tamaño: " + lista.size());

    // Mostrar elementos de la lista
    Producto objProd = null;
    for( int k = 0; k < lista.size(); k++ ){
        objProd = (Producto) lista.get(k);
        System.out.println( objProd.toString() );
    }

} // main
} // Prueba01

```

La creación de la lista se puede mejorar indicándole en el constructor el tipo de objeto que queremos almacenar en la lista, el script sería de la siguiente manera:

```
// Creación de la lista
ArrayList<Producto> lista = new ArrayList<Producto>();
```

Para mostrar los elementos de la lista también podemos utilizar el bucle `foreach`, el script quedaría de la siguiente manera:

```
// Mostrar elementos de la lista
for (Producto producto : lista) {
    System.out.println( producto.toString() );
}
```

También podemos utilizar un objeto de tipo `Iterator` para recorrer los elementos de una colección, en este caso del `ArrayList`, el script quedaría de la siguiente manera:

```
// Mostrar elementos de la lista
Iterator<Producto> it = lista.iterator();
while( it.hasNext() ){
    System.out.println(it.next().toString());
}
```

Cualquiera de los casos expuestos puede ser utilizado para recorrer los elementos de una colección.

La desventaja de la colección `ArrayList` es que permite elementos duplicados, tal como lo puede apreciar en la ejecución mostrada en la Figura 6.6.

```
Output - Cap06 (run-single)
init:
deps-jar:
Compiling 1 source file to F:\JavaN200\Cap06\build\classes
compile-single:
run-single:
Tamaño: 6
Televisor - 650.0
Impresora - 187.0
Computadora - 2356.0
Lavadora - 1348.0
Cocina - 582.0
Cocina - 582.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 6 . 6 Resultado de la ejecución de la clase `Prueba01`.

6.3.3. Manejo de listas mediante `HashSet`

Para aplicar `HashSet` debemos implementar los métodos `equals` y `hashCode` en la clase `Producto`.

En el método `equals` debemos desarrollar la forma como se deben comparar los objetos.

El método `hashCode` retornará un identificador único para cada instancia que se cree a partir de la clase `Producto`.

La Figura 6.7 muestra el diagrama de clases a utilizar en el presente ejemplo.

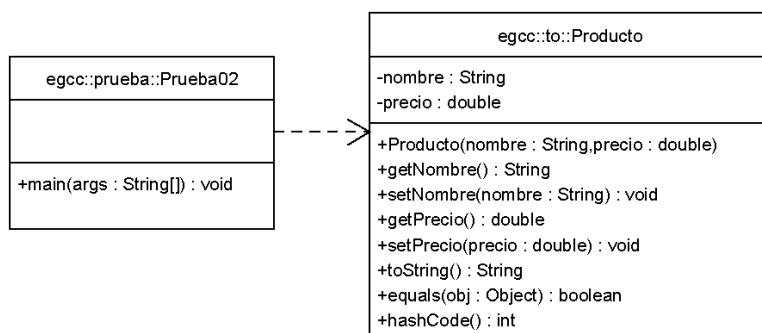


Figura 6 . 7 Diagrama de clases para el manejo de listas mediante `HashSet`.

Como primer paso debemos modificar la clase `Producto` agregándole los métodos `equals` y `hashCode`.

El nuevo script de la clase `Producto` es el siguiente:

```
package egcc.to;

public class Producto {

    // Campos

    private String nombre = null;
    private double precio;

    // Constructor

    public Producto( String nombre, double precio ) {
```

```

        this.setNombre(nombre);
        this.setPrecio(precio);
    } // Producto

    // Métodos

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public double getPrecio() {
        return precio;
    }

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

    @Override
    public String toString() {
        return this.getNombre() + " - " + this.getPrecio();
    }

    @Override
    public boolean equals(Object obj) {
        if( obj == null ){
            return false;
        }
        if (obj instanceof Producto) {
            Producto producto = (Producto) obj;
            return this.getNombre().equals(producto.getNombre());
        } else {
            return false;
        }
    }

    @Override
    public int hashCode() {
        return this.getNombre().hashCode();
    }
} // Producto

```

El script de la clase Prueba02 es el siguiente:

```

package egcc.prueba;

import egcc.to.Producto;
import java.util.HashSet;
import java.util.Iterator;

public class Prueba02 {

    public static void main(String[] args) {

```

```
// Creación de la lista
HashSet<Producto> lista = new HashSet<Producto>();

// Creación de los productos
Producto prod01 = new Producto( "Televisor", 650.0 );
Producto prod02 = new Producto( "Impresora", 187.0 );
Producto prod03 = new Producto( "Computadora", 2356.0 );
Producto prod04 = new Producto( "Lavadora", 1348.0 );
Producto prod05 = new Producto( "Cocina", 582.0 );
Producto prod06 = new Producto( "Cocina", 582.0 );

// Agregar productos a la lista
lista.add(prod01);
lista.add(prod02);
lista.add(prod03);
lista.add(prod04);
lista.add(prod05);
lista.add(prod06);

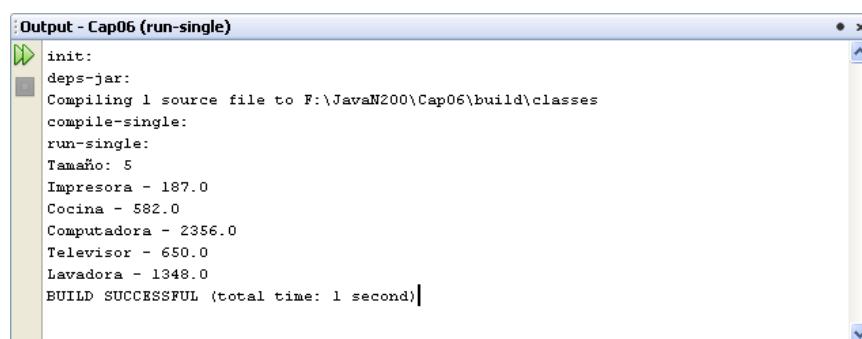
// Mostrar tamaño de la lista
System.out.println("Tamaño: " + lista.size());

// Mostrar elementos de la lista
Iterator<Producto> it = lista.iterator();
while( it.hasNext() ){
    System.out.println(it.next().toString());
}

} // main

} // Prueba02
```

En la Figura 6.8 puede comprobar el resultado de la ejecución de la clase Prueba02.



The screenshot shows the 'Output' window of an IDE. The title bar says 'Output - Cap06 (run-single)'. The window contains the following text:

```
init:
deps-jar:
Compiling 1 source file to F:\JavaN200\Cap06\build\classes
compile-single:
run-single:
Tamaño: 5
Impresora - 187.0
Cocina - 582.0
Computadora - 2356.0
Televisor - 650.0
Lavadora - 1348.0
BUILD SUCCESSFUL (total time: 1 second)
```

Figura 6 . 8 Resultado de la ejecución de la clase Prueba02.

Como seguramente ha podido comprobar se están agregando 6 objetos a la lista, sin embargo el reporte solo muestra 5, y esto es por que el quinto y sexto objeto contienen los mismos datos; lo mismo hubiera pasado si solamente el nombre del producto fuese el mismo y el precio diferente, y eso es por la manera como se ha implementado el método equals y hashCode en la clase Producto; queda como tarea para usted hacer la comprobación respectiva.

6.3.4. Manejo de listas mediante TreeSet

Para manejar listas con TreeSet debemos implementar la interfaz Comparable, esta interfaz define el método `compareTo()` en el cual debemos programar la forma como se debe comparar los objetos.

También debemos tener en cuenta que la implementación de listas con TreeSet no admite valores duplicados.

La Figura 6.9 muestra el diagrama de clases de la implementación de listas con TreeSet.

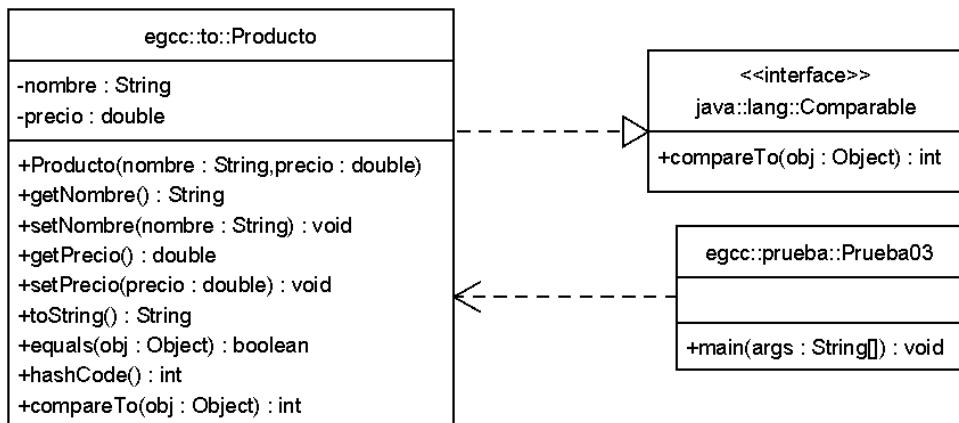


Figura 6 . 9 Diagrama de clases de la implementación de listas mediante TreeSet.

El script de la clase Producto quedaría de la siguiente manera:

```

package egcc.to;

public class Producto implements Comparable{

    // Campos

    private String nombre = null;
    private double precio;

    // Constructor

    public Producto( String nombre, double precio ) {
        this.setNombre(nombre);
        this.setPrecio(precio);
    } // Producto

    // Métodos

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public double getPrecio() {
        return precio;
    }
}

```

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

@Override
public String toString() {
    return this.getNombre() + " - " + this.getPrecio();
}

@Override
public boolean equals(Object obj) {
    if( obj == null ){
        return false;
    }
    if (obj instanceof Producto) {
        Producto producto = (Producto) obj;
        return this.getNombre().equals(producto.getNombre());
    } else {
        return false;
    }
}

@Override
public int hashCode() {
    return this.getNombre().hashCode();
}

public int compareTo(Object obj) {
    // Indica en base a que atributos se compara el objeto
    // Devuelve +1 si this es > que objeto
    // Devuelve -1 si this es < que objeto
    // Devuelve 0 si son iguales
    Producto producto = (Producto)obj;
    String nombreObj = producto.getNombre().toLowerCase();
    String nombreThis = this.getNombre().toLowerCase();
    return( nombreThis.compareTo( nombreObj ) );
}

} // Producto
```

El siguiente es el script de la clase Prueba03:

```
package egcc.prueba;

import egcc.to.Producto;
import java.util.Iterator;
import java.util.TreeSet;

public class Prueba03 {

    public static void main(String[] args) {

        // Creación de la lista
        TreeSet<Producto> lista = new TreeSet<Producto>();

        // Creación de los productos
        Producto prod01 = new Producto( "Televisor", 650.0 );
        Producto prod02 = new Producto( "Impresora", 187.0 );
        Producto prod03 = new Producto( "Computadora", 2356.0 );
```

```

        Producto prod04 = new Producto( "Lavadora", 1348.0 );
        Producto prod05 = new Producto( "Cocina", 582.0 );
        Producto prod06 = new Producto( "Cocina", 620.0 );

        // Agregar productos a la lista
        lista.add(prod01);
        lista.add(prod02);
        lista.add(prod03);
        lista.add(prod04);
        lista.add(prod05);
        lista.add(prod06);

        // Mostrar tamaño de la lista
        System.out.println("Tamaño: " + lista.size());

        // Mostrar elementos de la lista
        Iterator<Producto> it = lista.iterator();
        while( it.hasNext() ){
            System.out.println(it.next().toString());
        }

    } // main

} // Prueba03

```

En la Figura 6.10 tenemos el resultado de la ejecución de la clase Prueba03.

```

:Output - Cap06 (run-single)
init:
deps-jar:
compile-single:
run-single:
Tamaño: 5
Cocina - 582.0
Computadora - 2356.0
Impresora - 187.0
Lavadora - 1348.0
Televisor - 650.0
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figura 6 . 10 Resultado de la ejecución de la clase Prueba03.

Como puede comprobar se agregan 6 objetos a la lista pero solo se muestran 5, esto es debido a que la lista TreeSet no admite objetos duplicados, y ahora los productos se muestran ordenados según el criterio de comparación implementado en el método `compareTo()` de la clase `Producto`.

6.3.5. Ordenar y buscar datos en una lista

La clase `Collections` (que no es la interfaz `Collection`) nos permite ordenar y buscar elementos en listas.

Para ordenar los elementos de la lista se utiliza el método `sort()` y para buscar un elemento se utiliza el método `binarySearch()`.

Los objetos de la lista deben tener métodos `equals()`, `hashCode()` y `compareTo()` adecuados.

La Figura 6.11 muestra el diagrama de clases de la implementación de ordenamiento y búsqueda utilizando la clase `Colllections`.

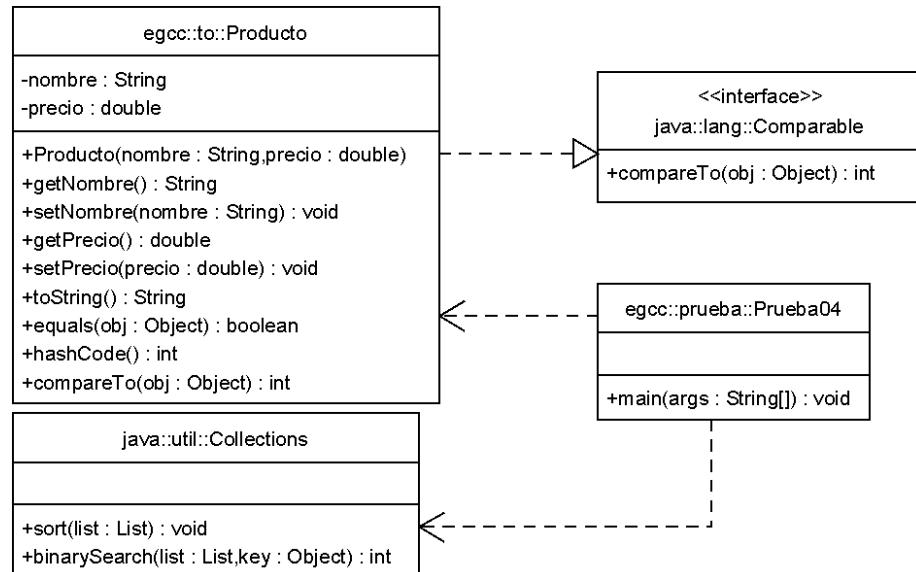


Figura 6 . 11 Diagrama de clases de la implementación de ordenamiento y búsqueda.

La clase `Producto` no ha sido modificada con respecto al caso presentado en el punto 6.3.3.

A continuación se lista el script de la clase `Prueba04`:

```

package egcc.prueba;

import egcc.to.Producto;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class Prueba04 {

    public static void main(String[] args) {

        // Creación de la lista
        ArrayList lista = new ArrayList();

        // Creación de los productos
        Producto prod01 = new Producto( "Televisor", 650.0 );
        Producto prod02 = new Producto( "Impresora", 187.0 );
        Producto prod03 = new Producto( "Computadora", 2356.0 );
        Producto prod04 = new Producto( "Lavadora", 1348.0 );
        Producto prod05 = new Producto( "Cocina", 582.0 );

        // Agregar productos a la lista
        lista.add(prod01);
        lista.add(prod02);
        lista.add(prod03);
        lista.add(prod04);
        lista.add(prod05);

        // Ordenar lista
        Collections.sort(lista);

        // Mostrar tamaño de la lista
        System.out.println("Tamaño: " + lista.size());
    }
}
  
```

```

// Mostrar lista
Iterator it = lista.iterator();
while( it.hasNext() ){
    System.out.println(it.next().toString());
}

// Buscar en la lista
Producto dato = new Producto("impresora", 0);
int posicion = Collections.binarySearch(lista, dato);
System.out.println("Posicion = " + posicion);

} // main

} // Prueba04

```

El resultado de la ejecución de esta clase se muestra en la Figura 6.12.

```

init:
deps-jar:
compile-single:
run-single:
Tamaño: 5
Cocina - 582.0
Computadora - 2356.0
Impresora - 187.0
Lavadora - 1348.0
Televisor - 650.0
Posicion = 2
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figura 6 . 12 Resultado de la ejecución de la clase Prueba04.

Para ordenar la lista se está utilizando el método `sort()` de la clase `Collections`, y para la búsqueda de un elemento se está utilizando el método `binarySearch()` de la misma clase.

Al método `binarySearch()` se le pasa como parámetro la lista y un objeto de tipo `Producto` con el nombre del artículo que se quiere buscar, en este caso no interesa el valor del campo `precio`, y esto por la forma como está implementado el método `compareTo()` en la clase `Producto`, donde la comparación solo se realiza en base al nombre del producto y descarta el valor del precio.

6.3.6. Manejo de datos de tipo clave/valor

La clase `HashMap` permite manejar pares de datos de tipo clave/valor.

En este ejemplo ilustraremos como crear una lista de tipo clave/valor y luego hacer su recorrido utilizando un objeto de tipo `Iterator`.

La Figura 6.13 muestra el diagrama de clases de la implementación de listas de tipo clave/valor.

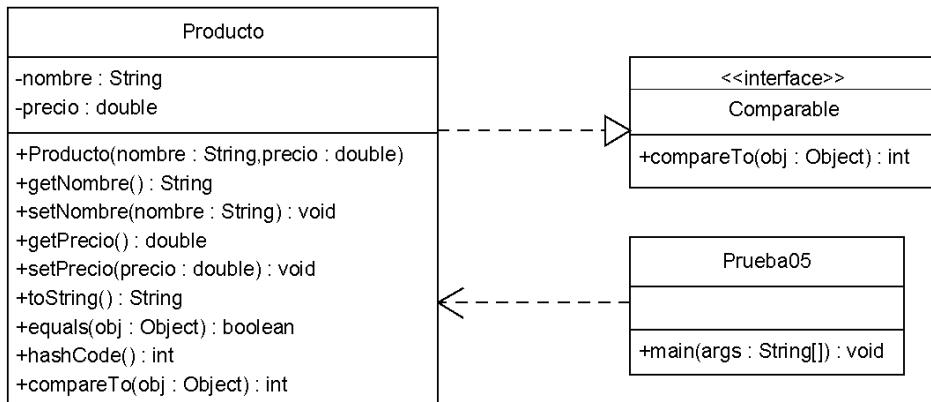


Figura 6 . 13 Diagrama de clases de la implementación de datos de tipo clave/valor.

La clase `Producto` no ha sido modificada con respecto al caso presentado en el punto 6.3.3.

A continuación se lista el script de la clase `Prueba05`:

```

package egcc.prueba;

import egcc.to.Producto;
import java.util.HashMap;
import java.util.Iterator;

public class Prueba05 {

    public static void main(String[] args) {

        // Creación de la lista
        HashMap lista = new HashMap();

        // Creación de los productos
        Producto prod01 = new Producto( "Televisor", 650.0 );
        Producto prod02 = new Producto( "Impresora", 187.0 );
        Producto prod03 = new Producto( "Computadora", 2356.0 );
        Producto prod04 = new Producto( "Lavadora", 1348.0 );
        Producto prod05 = new Producto( "Cocina", 582.0 );

        // Agregar productos a la lista
        lista.put("01", prod01);
        lista.put("02", prod02);
        lista.put("03", prod03);
        lista.put("04", prod04);
        lista.put("05", prod05);

        // Mostrar tamaño de la lista
        System.out.println("Tamaño: " + lista.size());

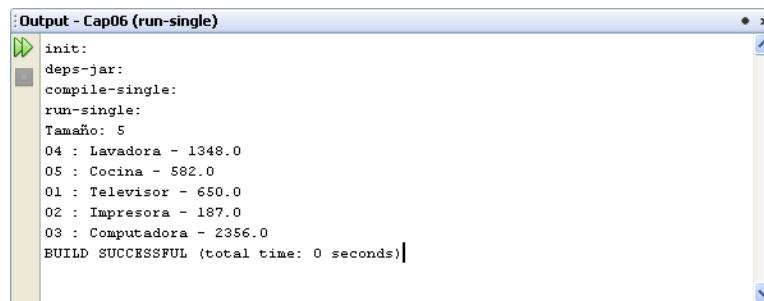
        // Mostrar lista
        Iterator it = lista.keySet().iterator();
        while( it.hasNext() ){
            String key = it.next().toString();
            Producto producto = (Producto)lista.get(key);
            System.out.println(key + " : " + producto.toString());
        }

    } // main
}
  
```

```
} // Prueba05
```

Para mostrar la lista se itera a través de las claves, luego esta clave se utiliza para acceder a cada uno de los objetos.

En la Figura 6.14 tenemos el resultado de la ejecución de la clase Prueba05.



```
init:  
deps-jar:  
compile-single:  
run-single:  
Tamaño: 5  
04 : Lavadora - 1348.0  
05 : Cocina - 582.0  
01 : Televisor - 650.0  
02 : Impresora - 187.0  
03 : Computadora - 2356.0  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 6 . 14 Resultado de la ejecución de la clase Prueba05.

Página en Blanco

Parte II



Base de Datos MySQL

Página en blanco.



Introducción a MySQL

MySQL es uno de los servidores de bases de datos más populares en el mundo; es utilizado por pequeñas, medianas y grandes empresas. Entre sus ventajas podemos mencionar que se trata de un motor robusto, fácil de administrar y multiplataforma.

Los puntos a tratar son:

- 7.1. Verificación del Servicio de MySQL
- 7.2. Programa Cliente de Línea de Comando de MySQL
- 7.3. Ejecutando Comandos

7.1. Verificación del Servicio de MySQL

Es importante que antes de empezar a trabajar con MySQL verifique que el servicio se encuentre ejecutándose; el nombre del servicio se define durante el proceso de instalación¹ de MySQL.

La verificación del servicio se puede realizar desde la ventana de servicios, que se obtiene ejecutando el comando `services.msc` en la ventana Ejecutar².

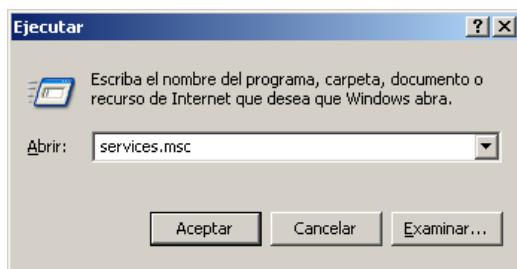
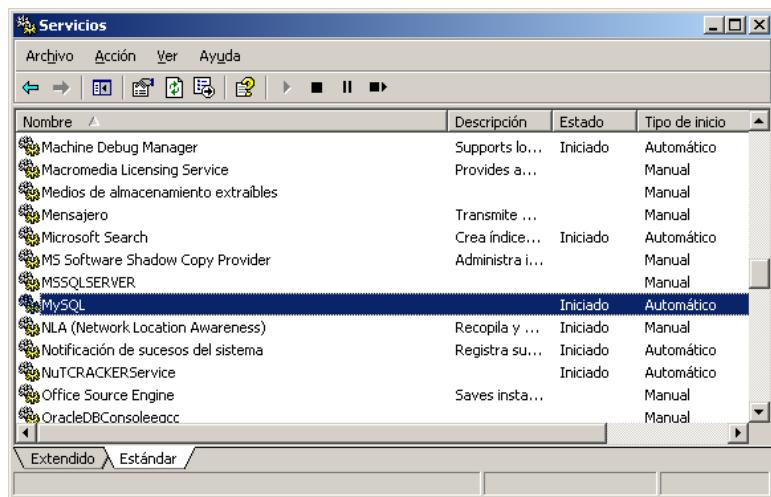


Figura 7 . 1 Cargando la ventana de servicios.

A continuación tenemos la ventana de servicios:

¹ La instalación de MySQL se explica en el Capítulo 2.

² Puede usar las teclas de acceso rápido [Tecla Win] + [R] para cargar la ventana Ejecutar.

**Figura 7 . 2** Ventana de Servicios.

Desde esta ventana de servicios no solo podemos verificar el estado del servicio de MySQL (de cualquier otro servicio); también podemos detenerlo, reiniciarlo, o si está detenido iniciarlo. Compruebe que el nombre del servicio sea MySQL o el que usted le asignó durante la instalación.

También podemos usar los comandos Net para manipular los servicios. Básicamente tenemos tres comandos Net para este propósito; tal como se explica a continuación:

- net start

Este comando muestra un listado de los servicios que actualmente están ejecutándose.

**Figura 7 . 3** Lista de servicios activos.

- net stop *nombre_servicio*

Este comando permite detener un servicio.

```
C:\>net stop mysql
El servicio de MySQL está deteniéndose.
El servicio de MySQL fue detenido con éxito.

C:\>
```

Figura 7 . 4 Detener un servicio.

- `net start nombre_servicio`

Este comando permite iniciar un servicio.

```
C:\>net start mysql
El servicio de MySQL se ha iniciado con éxito.

C:\>
```

Figura 7 . 5 Iniciar un servicio.

7.2. Programa Cliente de Línea de Comando de MySQL

Este programa es el que le permitirá ejecutar cualquier tipo de comando en el servidor MySQL; para cargar este programa debe ejecutar la siguiente opción de menú:

Inicio/Programas/MySQL/MySQL Server 5.0/MySQL Command Line Client

Se muestra la siguiente ventana:

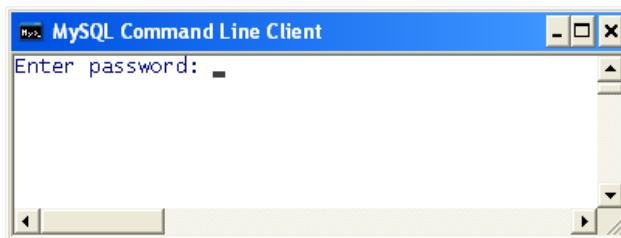


Figura 7 . 6 Consola de Trabajo de MySQL

Por defecto está solicitando la contraseña del usuario root; recuerde que durante la instalación no se le asignó ninguna contraseña, así que por lo tanto, solo debe presionar la tecla [Enter], se obtiene la siguiente ventana:

```
MySQL Command Line Client
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.41-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Figura 7 . 7 Consola de trabajo de MySQL.

Esta consola de comandos es la que debe utilizar para trabajar con el servidor de base de datos MySQL. También existen otras herramientas que puede bajar de la página de MySQL,

herramientas de terceros propietarias como TOAD para MySQL y AQUA Data Studio, y libres como PHPMyAdmin.

7.3. Ejecutando Comandos

1. Debemos tener en cuenta que todas las instrucciones SQL finalizan con punto y coma (;). Para consultar la versión de MySQL ejecute la siguiente instrucción:

```
mysql> select version();
+-----+
| version()           |
+-----+
| 5.0.41-community-nt |
+-----+
1 row in set (0.00 sec)
```

2. Podemos ejecutar dos instrucciones en una misma línea de comando, tal como se ilustra a continuación:

```
mysql> select sysdate(); select version();
+-----+
| sysdate()           |
+-----+
| 2007-11-17 21:56:36 |
+-----+
1 row in set (0.02 sec)

+-----+
| version()           |
+-----+
| 5.0.41-community-nt |
+-----+
1 row in set (0.00 sec)
```

3. Una instrucción también puede ser dividida en varias líneas, el ejemplo se ilustra a continuación:

```
mysql> select
    -> version();
+-----+
| version()           |
+-----+
| 5.0.41-community-nt |
+-----+
1 row in set (0.00 sec)
```

4. Para mostrar las bases de datos existentes, ejecute el siguiente comando:

```
mysql> show databases;
+-----+
| Database          |
+-----+
| information_schema |
| mysql              |
| test               |
+-----+
3 rows in set (0.06 sec)
```

5. Para crear una base de datos, el comando es el siguiente:

```
mysql> create database perudev;
Query OK, 1 row affected (0.03 sec)
```

Cada base de datos tiene su propia carpeta donde grabará sus archivos; esta carpeta se encuentra en C:\Archivos de programa\MySQL\MySQL Server 5.0\data, quiere decir que la carpeta de la base de datos perudev es C:\Archivos de programa\MySQL\MySQL Server 5.0\data\perudev.

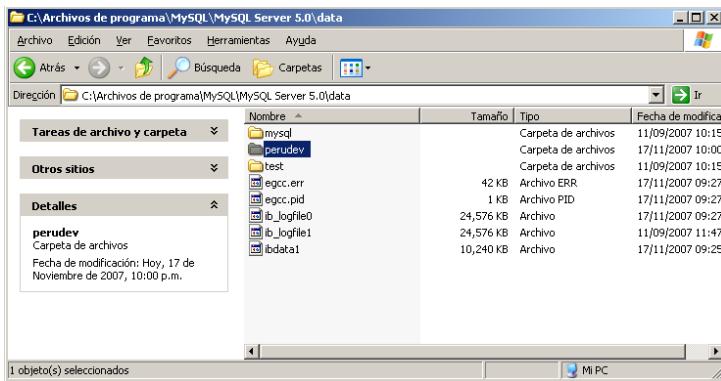


Figura 7 . 8 Carpeta de la base de datos perudev.

6. Antes de trabajar con una base de datos, debemos ubicarnos en ella, para lo cual debe ejecutar la siguiente instrucción:

```
mysql> use perudev;
Database changed
```

7. En ocasiones es importante verificar la base de datos activa, ejecute la siguiente instrucción:

```
mysql> select database();
+-----+
| database() |
+-----+
| perudev    |
+-----+
1 row in set (0.00 sec)
```

8. Para la creación de una tabla ejecutamos la instrucción `create table`, tal como se ilustra en el siguiente ejemplo:

```
mysql> create table articulo(
    -> codigo char(3),
    -> nombre varchar(30),
    -> precio decimal(8,2)
    -> );
Query OK, 0 rows affected (0.16 sec)
```

9. Para consultar las tablas de la base de datos activa debe ejecutar la instrucción `show tables`, como se muestra a continuación:

```
mysql> show tables;
+-----+
| Tables_in_perudev |
+-----+
| articulo          |
+-----+
1 row in set (0.00 sec)
```

10. Para consultar la estructura de una tabla se utiliza la instrucción `describe`. Por ejemplo, a continuación consultamos la estructura de la tabla `articulo`:

```
mysql> describe articulo;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| codigo | char(3) | YES  |     | NULL    |       |
| nombre | varchar(30) | YES  |     | NULL    |       |
| precio | decimal(8,2) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
3 rows in set (0.05 sec)
```

11. La instrucción `insert` se emplea para insertar filas a una tabla; ejecute el ejemplo que se ilustra a continuación:

```
mysql> insert into articulo
      -> values( '001','teclado',15);
Query OK, 1 row affected (0.03 sec)
```

12. La instrucción `select` se usa para consultar el contenido de una tabla; ejecute el siguiente ejemplo:

```
mysql> select * from articulo;
+-----+-----+-----+
| codigo | nombre | precio |
+-----+-----+-----+
| 001   | teclado | 15.00 |
+-----+-----+-----+
1 row in set (0.02 sec)
```

13. Para salir de esta herramienta utilice la instrucción `exit`, tal como se ilustra a continuación:

```
mysql> exit
Bye
```

Hasta aquí hemos visto de manera breve la forma de trabajar con una base de datos desde su misma creación; en los siguientes capítulos estaremos utilizando una base de datos más compleja en un proyecto que desarrollaremos en lo que resta del libro.



Especificaciones del Proyecto Banco

El objetivo de este capítulo es dar las especificaciones del proyecto al que denominaremos **Banco**, que desarrollaremos en los siguientes capítulos del libro.

También definiremos los estándares que utilizaremos en la construcción de la base de datos, y construiremos la base de datos.

Desarrollaremos los siguientes temas:

- 8.1. Requerimiento de Software
- 8.2. Estándares Utilizados
- 8.3. Diseño de la Base de Datos
- 8.4. Creación de la Base de Datos

8.1. Requerimiento de Software

La institución bancaria EurekaBank necesita hacer un sistema para automatizar la administración de las cuentas de ahorro, las condiciones son las siguientes:

1. Se cuenta con varias sucursales distribuidas en todo el país.
2. Los clientes, que solo son personas naturales, pueden tener más de una cuenta.
3. Las cuentas pueden ser en Nuevos Soles o Dólares.
4. Existe un cargo por mantenimiento de cuenta mensual, el cual debe aplicarse el primer día de cada mes según la siguiente tabla:

Moneda	Monto Máximo	Cargo por Mantenimiento
Soles	3500	7.00
Dólares	1200	2.50

El Monto Máximo quiere decir que cuentas con un saldo superior a este importe no se le cobrará el cargo por mantenimiento.

5. Los clientes pueden hacer 15 movimientos mensuales gratis en cada una de sus cuentas, los siguientes movimientos tienen un costo según la siguiente tabla:

Moneda	Costo por Movimiento
Soles	2.00
Dólares	0.60

6. Las cuentas ganan un interés mensual según la siguiente tabla:

Moneda	Interés Mensual
Soles	0.70%
Dólares	0.60%

7. También existen impuestos que se aplican a las transacciones, actualmente es de 0.08% del monto total de la transacción.
8. Todas las operaciones por Internet son gratis.

8.2. Estándares Utilizados

Para crear los objetos de la base de datos se han utilizados estándares que bien podría usted utilizarlos en sus futuros proyectos.

8.2.1. Nombre de las Tablas

Los nombres de las tablas pueden ser de una o más palabras, pero no debe tener espacios, guiones, subrayados, o cualquier otro carácter como conector.

Por ejemplo: Cliente, Factura, DetalleFactura, Movimiento, etc.

8.2.2. Nombre de las Columnas

Los nombres de las columnas deben construirse utilizando la siguiente plantilla:

aaa_bbbbNombre

Donde:

aaa Representa el tipo de datos de la columna, tal como se ilustra a continuación:

chr	Char
vch	Varchar
dec	Decimal
txt	Texto
int	Integer
lon	long
dtt	Date o DateTime

bbbb Representa los cuatro primeros caracteres del nombre de la tabla.

Nombre Representa el nombre que identifica el dato que guardará la columna.

A continuación tenemos algunos ejemplos de como determinar el nombre de columnas para almacenar ciertos datos:

- Si queremos definir en la tabla factura una columna donde almacenaremos el importe de una venta, su definición debería ser de la siguiente manera:

dec_factimporte decimal(12,2)

- Si queremos guardar el nombre del cliente en la tabla `cliente`, la definición de la columna debería ser de la siguiente manera:

```
vch_clienombre varchar(30)
```

- Si queremos guardar el DNI de un trabajador en la tabla `empleado`, la definición de la columna debería ser de la siguiente manera:

```
chr_empldni char(8)
```

- Si queremos guardar el RUC en la tabla `cliente`, la definición de la columna debería ser de la siguiente manera:

```
chr_clieruc char(11)
```

8.2.3. Nombre de las Restricciones

8.2.3.1. Primary Key

Para dar nombre a una restricción de tipo Primary Key se utiliza la siguiente plantilla:

```
pk_NombreTabla
```

Por ejemplo, si queremos definir la Primary Key de la tabla `factura`, el nombre de esta restricción sería:

```
pk_factura
```

8.2.3.2. Foreign Key

Para dar nombre a una restricción de tipo Foreign Key se utiliza la siguiente plantilla:

```
fk_NombreTabla_NombreTablaReferenciada
```

Por ejemplo, si queremos definir la Foreign Key en la tabla `factura` que hace referencia a la tabla `cliente`, el nombre de esta restricción sería:

```
fk_factura_cliente
```

8.2.3.3. Check

Si queremos dar nombre a una restricción de tipo Check tenemos dos casos.

Caso 1

Si se trata de un Check aplicado a una columna, la plantilla es la siguiente:

```
chk_NombreTabla_NombreColumna
```

En este caso se compara el dato que toma la columna con un valor constante.

Por ejemplo, si queremos que la columna `dec_alumnnota` en la tabla `Alumno` solo acepte valores entre 0 y 20; en la definición de la columna debemos definir la restricción correspondiente, tal como se ilustra a continuación:

```
dec_alumnnota decimal(4,2)
constraint chk_alumno_nota
```

```
check( dec_alumnnota between 0.0 and 20.0)
```

Caso 2

Si se trata de un Check aplicado a la tabla, la plantilla es la siguiente:

```
chk_NombreTabla_NombreRestricción
```

En este caso podemos comparar dos columnas.

Por ejemplo, Si queremos que en la tabla pedido la fecha de reparto sea posterior a la fecha de pedido debemos construir una restricción de tipo check aplicado a la tabla, tal como se ilustra a continuación:

```
alter table pedido
  add constraint chk_pedido_fechas
    check( dtt_pedifechareparto > dtt_pedifechapedido );
```

8.2.3.4. Unique

Para dar nombre a una restricción de tipo Unique se utiliza la siguiente plantilla:

```
u_NombreTabla_NombreRestricción
```

Por ejemplo, si queremos asegurarnos que los nombres de los artículos en la tabla articulo sean únicos, debemos crear una restricción de tipo Unique, tal como se ilustra a continuación:

```
alter table articulo
  add constraint u_articulo_nombre
    unique ( vch_artinombre );
```

Podríamos tomar como NombreRestricción el nombre de la columna a la que afecta la restricción.

8.2.4. Nombres de Índices

Para dar nombre a un índice se utiliza la siguiente plantilla:

```
idx_NombreTabla_NombreIndice
```

Por ejemplo, si queremos crear un índice por fecha en la tabla articulo, el nombre debería ser:

```
idx_articulo_fecha
```

Podríamos tomar como NombreIndice el nombre de la columna a la que afecta el índice.

8.3. Diseño de la Base de Datos

Para explicar el diseño de la base de datos utilizaremos sub-esquemas, esto nos ayudará a entender de manera más rápida el modelo de datos. La representación se hará utilizando la notación Information Engineering.

8.3.1. Tablas Generales

Normalmente las tablas generales tienen datos que muy poco cambian en el tiempo, también podrían ser datos que sirven como parámetros de la aplicación.

La Figura 8.1 muestra las tablas de parámetros y sus relaciones.

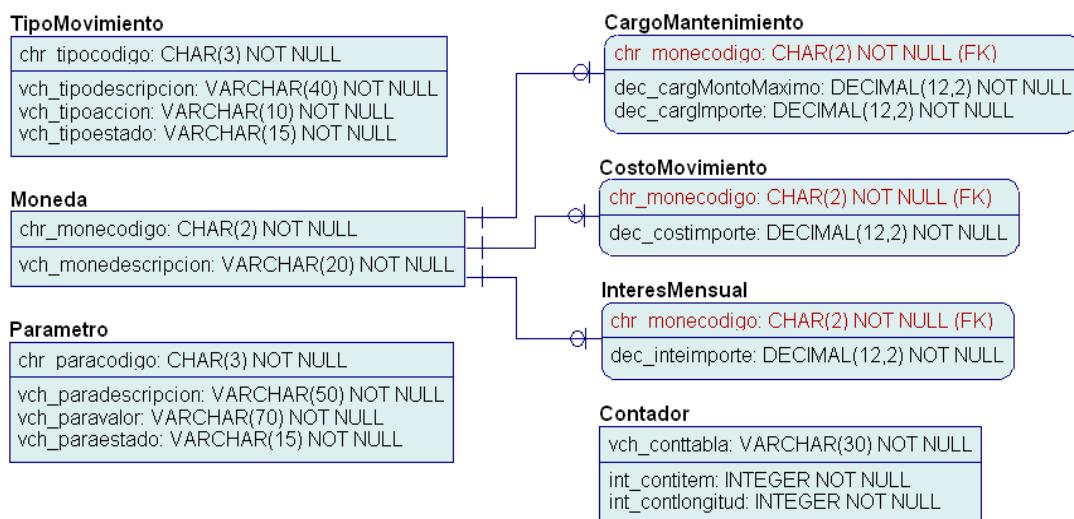


Figura 8 . 1 Sub-Esquema Tablas Generales

8.3.1.1. Tabla: Moneda

Esta tabla guarda los códigos signados a cada moneda.

8.3.1.2. Tabla: CargoMantenimiento

Esta tabla contiene el importe que se debe cargar por mantenimiento según las especificaciones del caso.

8.3.1.3. Tabla: CargoMovimiento

Esta tabla contiene el importe que se debe cargar por movimiento según las especificaciones del caso.

8.3.1.4. Tabla: InteresMensual

Esta tabla contiene el interés (porcentaje) que se debe abonar a cada cuenta en forma mensual según las especificaciones del caso.

8.3.1.5. Tabla: TipoMovimiento

Esta tabla contiene los tipos de movimiento que pueden efectuarse en una cuenta.

8.3.1.6. Tabla: Contador

Esta tabla sirve para controlar los correlativos de las otras tablas.

8.3.1.7. Tabla: Parámetro

Esta tabla sirve para guardar los parámetros del sistema; como por ejemplo, la cantidad de movimientos mensual sin costo que tiene el cliente en cada una de sus cuentas.

8.3.2. Sucursales

En este sub-esquema tenemos las tablas que determinan como se almacenan las sucursales, tal como se puede observar en la Figura 8.2.

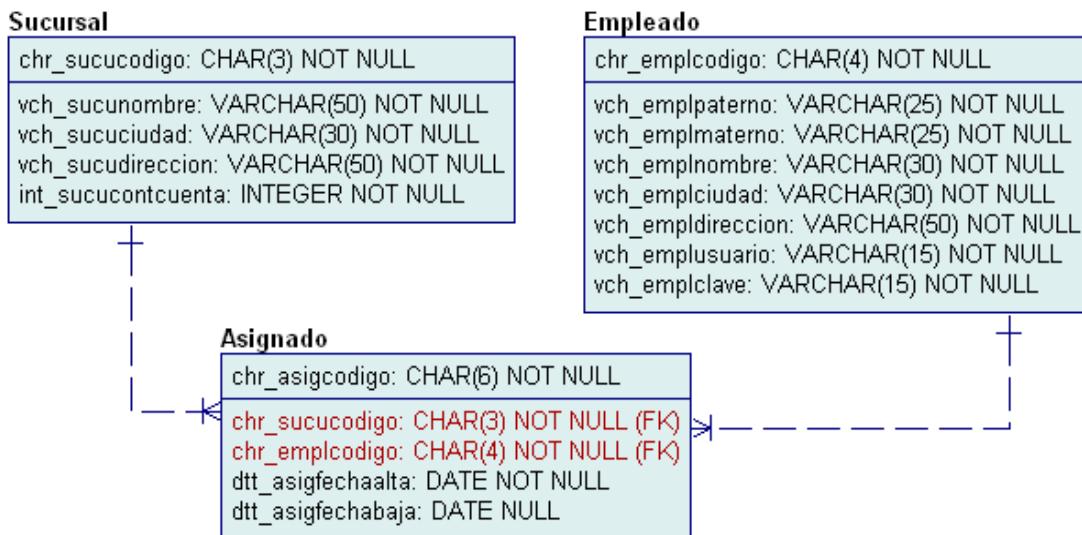


Figura 8 . 2 Sub-Esquema Sucursales

8.3.2.1. Tabla: Sucursal

Esta tabla permite identificar plenamente cada sucursal.

8.3.2.2. Tabla: Empleado

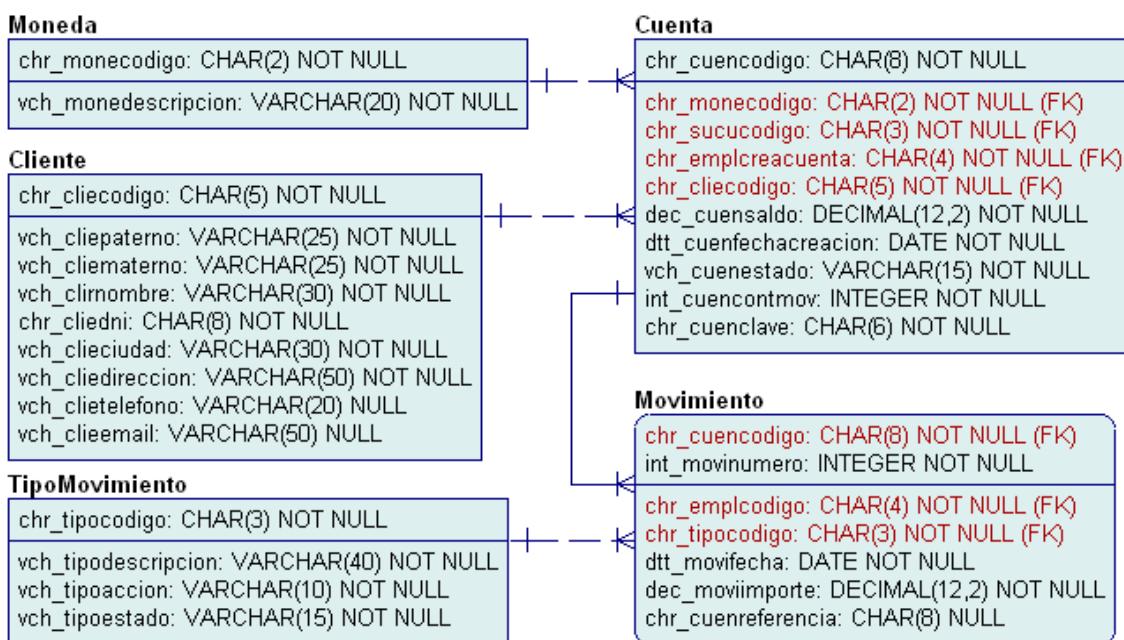
En esta tabla se registran todos los empleados de la empresa. Debe existir un empleado con código **9999** de nombre **Internet** para identificar las operaciones realizadas desde la Web por los mismos clientes.

8.3.2.3. Tabla: Asignado

Esta tabla nos permite identificar que empleados se encuentran asignados a cada sucursal.

8.3.3. Cuentas

En este sub-esquema tenemos tres nuevas tablas que se pueden observar en la Figura 8.3.

**Figura 8 . 3 Sub-Esquema Cuentas**

8.3.3.1. Tabla: Cliente

En esta tabla se registrarán todos los clientes

8.3.3.2. Tabla: Cuenta

En esta tabla se registrarán todas las cuentas. Cada cuenta tiene su propio contador de movimientos, la columna `int_cuencontmov` se utiliza para este propósito.

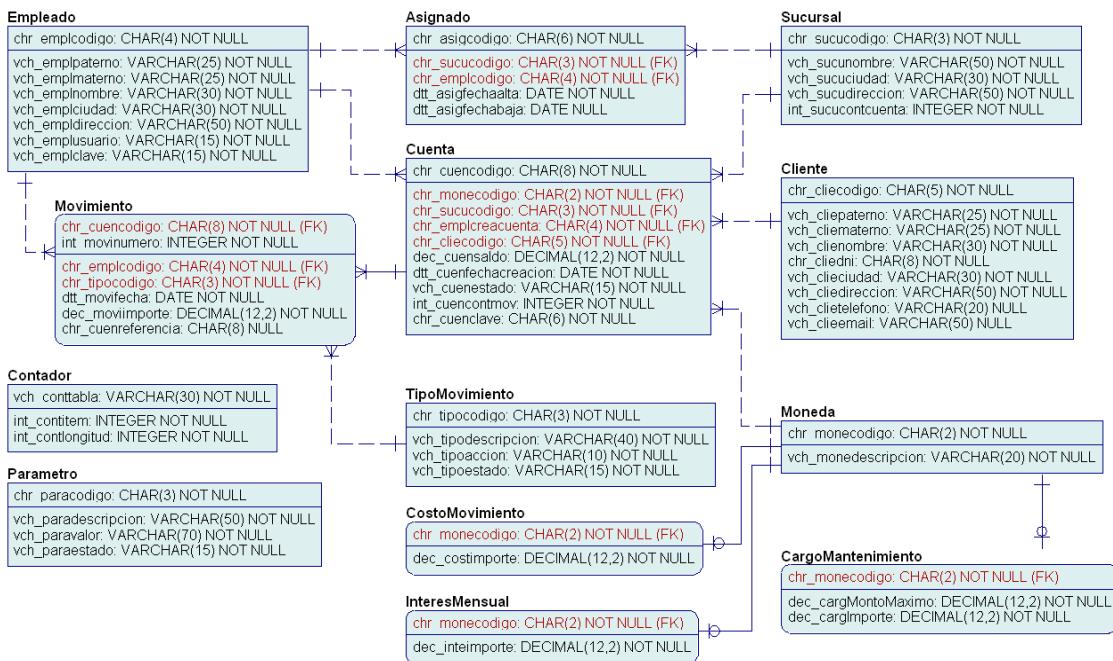
Cada cuenta tiene una clave de acceso para que los clientes puedan hacer consultas y transacciones desde los cajeros e Internet.

8.3.3.3 Tabla: Movimiento

En esta tabla registraremos todos los movimientos de las cuentas.

8.3.4. Esquema Completo

A continuación tenemos el esquema completo de la base de datos de EurekaBank.

**Figura 8 . 4** Esquema completo de la base de datos.

8.4. Creación de la Base de Datos

Este libro trae un CD donde encontrará el archivo **CREA_BD.SQL**, este archivo contiene el script para crear la base de datos.

Para crear la base de datos solo tiene que ejecutar el archivo `crea_bd.sql`. Suponiendo que este archivo se encuentra en la raíz de la unidad **C**; desde la línea de comando de MySQL debe ejecutar la siguiente instrucción:

```
MySQL> \. C:\crea_bd.sql
```

A continuación tenemos un listado del contenido de este archivo:

```
/*
Empresa      : EurekaBank
Software     : Sistema de Cuentas de Ahorro
DBMS        : MySQL Server
Base de Datos : eurekabank
Script       : Crea la Base de Datos
Responsable  : Eric Gustavo Coronel Castillo
Telefono     : (511) 9666-4457
Email        : gcoronel@viabcp.com
*/



-- =====
-- Creación de la Base de Datos
-- =====

CREATE DATABASE IF NOT EXISTS eurekabank;

-- =====
-- Seleccionar la Base de Datos
-- =====

USE eurekabank;

-- =====
-- Eliminar las tablas en caso existan
-- =====
```

```

DROP TABLE IF EXISTS asignado;
DROP TABLE IF EXISTS contador;
DROP TABLE IF EXISTS parametro;
DROP TABLE IF EXISTS movimiento;
DROP TABLE IF EXISTS tipomovimiento;
DROP TABLE IF EXISTS cuenta;
DROP TABLE IF EXISTS cliente;
DROP TABLE IF EXISTS empleado;
DROP TABLE IF EXISTS sucursal;
DROP TABLE IF EXISTS interesmensual;
DROP TABLE IF EXISTS costomovimiento;
DROP TABLE IF EXISTS cargomantenimiento;
DROP TABLE IF EXISTS moneda;

-- =====
-- Creación de los Objetos de la Base de Datos
-- =====

CREATE TABLE Tipomovimiento (
    chr_tipocodigo      CHAR(3) NOT NULL,
    vch_tipodescripcion VARCHAR(40) NOT NULL,
    vch_tipoaccion       VARCHAR(10) NOT NULL,
    vch_tipostado        VARCHAR(15) DEFAULT 'ACTIVO' NOT NULL,
    CONSTRAINT PK_TipoMovimiento
        PRIMARY KEY (chr_tipocodigo),
    CONSTRAINT chk_tipomovimiento_vch_tipoaccion
        CHECK (vch_tipoaccion IN ('INGRESO', 'SALIDA')),
    CONSTRAINT chk_tipomovimiento_vch_tipostado
        CHECK (vch_tipostado IN ('ACTIVO', 'ANULADO', 'CANCELADO'))
) ENGINE = INNODB ;

CREATE TABLE Sucursal (
    chr_sucucodigo      CHAR(3) NOT NULL,
    vch_sucunombre       VARCHAR(50) NOT NULL,
    vch_sucuciudad       VARCHAR(30) NOT NULL,
    vch_sucudireccion    VARCHAR(50) NULL,
    int_sucucontcuenta  INTEGER NOT NULL,
    CONSTRAINT PK_Sucursal
        PRIMARY KEY (chr_sucucodigo)
) ENGINE = INNODB ;

CREATE TABLE Empleado (
    chr_emplcodigo       CHAR(4) NOT NULL,
    vch_emplpaterno      VARCHAR(25) NOT NULL,
    vch_emplmaterno      VARCHAR(25) NOT NULL,
    vch_emplnombre       VARCHAR(30) NOT NULL,
    vch_emplciudad       VARCHAR(30) NOT NULL,
    vch_empldireccion    VARCHAR(50) NULL,
    vch_emplusuario      VARCHAR(15) NOT NULL,
    vch_emplclave        VARCHAR(15) NOT NULL,
    CONSTRAINT PK_Emppleado
        PRIMARY KEY (chr_emplcodigo),
    CONSTRAINT U_Emppleado_vch_emplusuario
        UNIQUE (vch_emplusuario)
) ENGINE = INNODB ;

CREATE TABLE Asignado (
    chr_asigcodigo        CHAR(6) NOT NULL,
    chr_sucucodigo        CHAR(3) NOT NULL,
    chr_emplcodigo         CHAR(4) NOT NULL,
    dtt_asigfechaalta     DATE NOT NULL,
    dtt_asigfechabaja     DATE NULL,
    CONSTRAINT PK_Asignado
        PRIMARY KEY (chr_asigcodigo),
    KEY idx_asignado01 (chr_emplcodigo),
    CONSTRAINT fk_asignado_empleado
        FOREIGN KEY (chr_emplcodigo)
            REFERENCES Empleado (chr_emplcodigo)
            ON DELETE RESTRICT
            ON UPDATE RESTRICT,
    KEY idx_asignado02 (chr_sucucodigo),
    CONSTRAINT fk_asignado_sucursal
        FOREIGN KEY (chr_sucucodigo)
            REFERENCES Sucursal (chr_sucucodigo)
)

```

```
    ON DELETE RESTRICT
    ON UPDATE RESTRICT
) ENGINE = INNODB;

CREATE TABLE Cliente (
    chr_cliecodigo      CHAR(5) NOT NULL,
    vch_cliepaterno     VARCHAR(25) NOT NULL,
    vch_clematerno     VARCHAR(25) NOT NULL,
    vch_clienombre      VARCHAR(30) NOT NULL,
    chr_cliedni         CHAR(8) NOT NULL,
    vch_clieciudad      VARCHAR(30) NOT NULL,
    vch_cliedireccion   VARCHAR(50) NOT NULL,
    vch_cletelefono     VARCHAR(20) NULL,
    vch_clieemail        VARCHAR(50) NULL,
    CONSTRAINT PK_Cliente
        PRIMARY KEY (chr_cliecodigo)
) ENGINE = INNODB ;

CREATE TABLE Moneda (
    chr_monecodigo      CHAR(2) NOT NULL,
    vch_moneddescripcion VARCHAR(20) NOT NULL,
    CONSTRAINT PK_Moneda
        PRIMARY KEY (chr_monecodigo)
) ENGINE = INNODB ;

CREATE TABLE Cuenta (
    chr_cuencodigo      CHAR(8) NOT NULL,
    chr_monecodigo       CHAR(2) NOT NULL,
    chr_sucucodigo      CHAR(3) NOT NULL,
    chr_emplcreacuenta  CHAR(4) NOT NULL,
    chr_cliecodigo       CHAR(5) NOT NULL,
    dec_cuensaldo        DECIMAL(12,2) NOT NULL,
    dtt_cuenfechacreacion DATE NOT NULL,
    vch_cuenestado        VARCHAR(15) DEFAULT 'ACTIVO' NOT NULL,
    int_cuencontmov      INTEGER NOT NULL,
    chr_cuenclave        CHAR(6) NOT NULL,
    CONSTRAINT chk_cuenta_chr_cuenestado
        CHECK (vch_cuenestado IN ('ACTIVO', 'ANULADO', 'CANCELADO')),
    CONSTRAINT PK_Cuenta
        PRIMARY KEY (chr_cuencodigo),
    KEY idx_cuenta01 (chr_cliecodigo),
    CONSTRAINT fk_cuenta_cliente
        FOREIGN KEY (chr_cliecodigo)
            REFERENCES Cliente (chr_cliecodigo)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT,
    KEY idx_cuenta02 (chr_emplcreacuenta),
    CONSTRAINT fk_cuenta_empleado
        FOREIGN KEY (chr_emplcreacuenta)
            REFERENCES Empleado (chr_emplcodigo)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT,
    KEY idx_cuenta03 (chr_sucucodigo),
    CONSTRAINT fk_cuenta_sucursal
        FOREIGN KEY (chr_sucucodigo)
            REFERENCES Sucursal (chr_sucucodigo)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT,
    KEY idx_cuenta04 (chr_monecodigo),
    CONSTRAINT fk_cuenta_moneda
        FOREIGN KEY (chr_monecodigo)
            REFERENCES Moneda (chr_monecodigo)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT
) ENGINE = INNODB ;

CREATE TABLE Movimiento (
    chr_cuencodigo      CHAR(8) NOT NULL,
    int_movinumero       INTEGER NOT NULL,
    dtt_movifecha        DATE NOT NULL,
    chr_emplcodigo       CHAR(4) NOT NULL,
    chr_tipocodigo       CHAR(3) NOT NULL,
    dec_moviimporte      DECIMAL(12,2) NOT NULL,
    chr_cuenreferencia   CHAR(8) NULL,
    CONSTRAINT chk_Movimiento_importe4
        CHECK (dec_moviimporte >= 0.0),
    CONSTRAINT PK_Movimiento

```

```

        PRIMARY KEY (chr_cuencodigo, int_movinumero),
        KEY idx_movimiento01 (chr_tipocodigo),
        CONSTRAINT fk_movimiento_tipomovimiento
          FOREIGN KEY (chr_tipocodigo)
            REFERENCES TipoMovimiento (chr_tipocodigo)
            ON DELETE RESTRICT
            ON UPDATE RESTRICT,
        KEY idx_movimiento02 (chr_emplcodigo),
        CONSTRAINT fk_movimiento_empleado
          FOREIGN KEY (chr_emplcodigo)
            REFERENCES Empleado (chr_emplcodigo)
            ON DELETE RESTRICT
            ON UPDATE RESTRICT,
        KEY idx_movimiento03 (chr_cuencodigo),
        CONSTRAINT fk_movimiento_cuenta
          FOREIGN KEY (chr_cuencodigo)
            REFERENCES Cuenta (chr_cuencodigo)
            ON DELETE RESTRICT
            ON UPDATE RESTRICT
      ) ENGINE = INNODB ;

CREATE TABLE Parametro (
    chr_paracodigo      CHAR(3) NOT NULL,
    vch_paradescripcion VARCHAR(50) NOT NULL,
    vch_paravalor       VARCHAR(70) NOT NULL,
    vch_paraestado      VARCHAR(15) DEFAULT 'ACTIVO' NOT NULL,
    CONSTRAINT chk_parametro_vch_paraestado
      CHECK (vch_paraestado IN ('ACTIVO', 'ANULADO', 'CANCELADO')),
    CONSTRAINT PK_Parametro
      PRIMARY KEY (chr_paracodigo)
) ENGINE = INNODB ;

CREATE TABLE InteresMensual (
    chr_monecodigo      CHAR(2) NOT NULL,
    dec_inteimporte     DECIMAL(12,2) NOT NULL,
    CONSTRAINT PK_InteresMensual
      PRIMARY KEY (chr_monecodigo),
    KEY idx_interesmensual01 (chr_monecodigo),
    CONSTRAINT fk_interesmensual_moneda
      FOREIGN KEY (chr_monecodigo)
        REFERENCES Moneda (chr_monecodigo)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT
) ENGINE = INNODB ;

CREATE TABLE CostoMovimiento (
    chr_monecodigo      CHAR(2) NOT NULL,
    dec_costimporte     DECIMAL(12,2) NOT NULL,
    CONSTRAINT PK_CostoMovimiento
      PRIMARY KEY (chr_monecodigo),
    KEY idx_costomovimiento (chr_monecodigo),
    CONSTRAINT fk_costomovimiento_moneda
      FOREIGN KEY (chr_monecodigo)
        REFERENCES Moneda (chr_monecodigo)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT
) ENGINE = INNODB ;

CREATE TABLE CargoMantenimiento (
    chr_monecodigo      CHAR(2) NOT NULL,
    dec_cargMontoMaximo DECIMAL(12,2) NOT NULL,
    dec_cargImporte     DECIMAL(12,2) NOT NULL,
    CONSTRAINT PK_CargoMantenimiento
      PRIMARY KEY (chr_monecodigo),
    KEY idx_cargomantenimiento01 (chr_monecodigo),
    CONSTRAINT fk_cargomantenimiento_moneda
      FOREIGN KEY (chr_monecodigo)
        REFERENCES Moneda (chr_monecodigo)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT
) ENGINE = INNODB ;

CREATE TABLE Contador (
    vch_conttabla       VARCHAR(30) NOT NULL,
    int_contitem        INTEGER NOT NULL,
    int_contlongitud   INTEGER NOT NULL,
    CONSTRAINT PK_Contador

```

```
    PRIMARY KEY (vch_conttabla)
) ENGINE = INNODB ;
```

También encontrará el archivo **CARGA_DATOS.SQL**, en este archivo se encuentran las instrucciones para cargar datos de prueba, lo debe ejecutar de manera similar al archivo anterior.

A continuación tenemos el listado del archivo **carga_datos.sql**:

```
/*
Empresa      : EurekaBank
Software     : Sistema de Cuentas de Ahorro
DBMS         : MySQL Server
Base de Datos : eurekabank
Script        : Carga Datos
Responsable   : Eric Gustavo Coronel Castillo
Telefono     : (511) 9666-4457
Email         : gcoronel@viabcp.com
*/



-- =====
-- Cargar Datos de Prueba
-- =====

-- Tabla: Moneda

insert into moneda values ( '01', 'Soles' );
insert into moneda values ( '02', 'Dolares' );

-- Tabla: CargoMantenimiento

insert into cargomantenimiento values ( '01', 3500.00, 7.00 );
insert into cargomantenimiento values ( '02', 1200.00, 2.50 );

-- Tabla: CostoMovimiento

insert into CostoMovimiento values ( '01', 2.00 );
insert into CostoMovimiento values ( '02', 0.60 );

-- Tabla: InteresMensual

insert into InteresMensual values ( '01', 0.70 );
insert into InteresMensual values ( '02', 0.60 );

-- Tabla: TipoMovimiento

insert into TipoMovimiento values( '001', 'Apertura de Cuenta', 'INGRESO', 'ACTIVO' );
insert into TipoMovimiento values( '002', 'Cancelar Cuenta', 'SALIDA', 'ACTIVO' );
insert into TipoMovimiento values( '003', 'Deposito', 'INGRESO', 'ACTIVO' );
insert into TipoMovimiento values( '004', 'Retiro', 'SALIDA', 'ACTIVO' );
insert into TipoMovimiento values( '005', 'Interes', 'INGRESO', 'ACTIVO' );
insert into TipoMovimiento values( '006', 'Mantenimiento', 'SALIDA', 'ACTIVO' );
insert into TipoMovimiento values( '007', 'ITF', 'SALIDA', 'ACTIVO' );
insert into TipoMovimiento values( '008', 'Transferencia', 'INGRESO', 'ACTIVO' );
insert into TipoMovimiento values( '009', 'Transferencia', 'SALIDA', 'ACTIVO' );
insert into TipoMovimiento values( '010', 'Cargo por Movimiento', 'SALIDA', 'ACTIVO' );

-- Tabla: Sucursal

insert into sucursal values( '001', 'Sipan', 'Chiclayo', 'Av. Balta 1456', 2 );
insert into sucursal values( '002', 'Chan Chan', 'Trujillo', 'Jr. Independencia 456', 3 );
insert into sucursal values( '003', 'Los Olivos', 'Lima', 'Av. Central 1234', 0 );
insert into sucursal values( '004', 'Pardo', 'Lima', 'Av. Pardo 345 - Miraflores', 0 );
insert into sucursal values( '005', 'Misti', 'Arequipa', 'Bolivar 546', 0 );
insert into sucursal values( '006', 'Machupicchu', 'Cusco', 'Calle El Sol 534', 0 );
insert into sucursal values( '007', 'Grau', 'Piura', 'Av. Grau 1528', 0 );

-- Tabla: Empleado

INSERT INTO empleado VALUES( '9999', 'Internet', 'Internet', 'internet', 'Internet',
'internet', 'internet', 'internet' );
```

```

INSERT INTO empleado VALUES( '0001', 'Romero', 'Castillo', 'Carlos Alberto', 'Trujillo',
'Call 1 Nro. 456', 'cromero', 'chicho' );
INSERT INTO empleado VALUES( '0002', 'Castro', 'Vargas', 'Lidia', 'Lima', 'Federico
Villarreal 456 - SMP', 'lcastro', 'flaca' );
INSERT INTO empleado VALUES( '0003', 'Reyes', 'Ortiz', 'Claudia', 'Lima', 'Av. Aviación
3456 - San Borja', 'aortiz', 'linda' );
INSERT INTO empleado VALUES( '0004', 'Ramos', 'Garibay', 'Angelica', 'Chiclayo', 'Calle
Barcelona 345', 'aramos', 'china' );
INSERT INTO empleado VALUES( '0005', 'Ruiz', 'Zabaleta', 'Claudia', 'Cusco', 'Calle Cruz
Verde 364', 'cvalencia', 'angel' );
INSERT INTO empleado VALUES( '0006', 'Cruz', 'Taraazona', 'Ricardo', 'Arequipa', 'Calle
La Gruta 304', 'rcruz', 'cerebro' );
INSERT INTO empleado VALUES( '0007', 'Diaz', 'Flores', 'Edith', 'Lima', 'Av. Pardo 546',
'ediaz', 'princesa' );
INSERT INTO empleado VALUES( '0008', 'Sarmiento', 'Bellido', 'Claudia Rocio',
'Arequipa', 'Calle Alfonso Ugarte 1567', 'csarmiento', 'chinita' );
INSERT INTO empleado VALUES( '0009', 'Pachas', 'Sifuentes', 'Luis Alberto', 'Trujillo',
'Francisco Pizarro 1263', 'lpachas', 'gato' );
INSERT INTO empleado VALUES( '0010', 'Tello', 'Alarcon', 'Hugo Valentin', 'Cusco', 'Los
Angeles 865', 'htello', 'machupichu' );
INSERT INTO empleado VALUES( '0011', 'Carrasco', 'Vargas', 'Pedro Hugo', 'Chiclayo',
'Av. Balta 1265', 'pcarrasco', 'tinajones' );

```

-- Asignado

```

insert into Asignado values( '000001', '001', '0004', sysdate(), null );
insert into Asignado values( '000002', '002', '0001', sysdate(), null );
insert into Asignado values( '000003', '003', '0002', sysdate(), null );
insert into Asignado values( '000004', '004', '0003', sysdate(), null );
insert into Asignado values( '000005', '005', '0006', sysdate(), null );
insert into Asignado values( '000006', '006', '0005', sysdate(), null );
insert into Asignado values( '000007', '004', '0007', sysdate(), null );
insert into Asignado values( '000008', '005', '0008', sysdate(), null );
insert into Asignado values( '000009', '001', '0011', sysdate(), null );
insert into Asignado values( '000010', '002', '0009', sysdate(), null );
insert into Asignado values( '000011', '006', '0010', sysdate(), null );

```

-- Tabla: Parametro

```

insert into Parametro values( '001', 'ITF - Impuesto a la Transacciones Financieras',
'0.08', 'ACTIVO' );
insert into Parametro values( '002', 'Número de Operaciones Sin Costo', '15', 'ACTIVO'
);

```

-- Tabla: Cliente

```

insert into cliente values( '00001', 'CORONEL', 'CASTILLO', 'ERIC GUSTAVO', '06914897',
'LIMA', 'LOS OLIVOS', '9666-4457', 'gcoronel@viabcp.com' );
insert into cliente values( '00002', 'VALENCIA', 'MORALES', 'PEDRO HUGO', '01576173',
'LIMA', 'MAGDALENA', '924-7834', 'pvalencia@terra.com.pe' );
insert into cliente values( '00003', 'MARCELO', 'VILLALOBOS', 'RICARDO', '10762367',
'LIMA', 'LINCE', '993-62966', 'ricardomarcelo@hotmail.com' );
insert into cliente values( '00004', 'ROMERO', 'CASTILLO', 'CARLOS ALBERTO', '06531983',
'LIMA', 'LOS OLIVOS', '865-84762', 'c.romero@hotmail.com' );
insert into cliente values( '00005', 'ARANDA', 'LUNA', 'ALAN ALBERTO', '10875611',
'LIMA', 'SAN ISIDRO', '834-67125', 'a.aranda@hotmail.com' );
insert into cliente values( '00006', 'AYALA', 'PAZ', 'JORGE LUIS', '10679245', 'LIMA',
'SAN BORJA', '963-34769', 'j.ayala@yahoo.com' );
insert into cliente values( '00007', 'CHAVEZ', 'CANALES', 'EDGAR RAFAEL', '10145693',
'LIMA', 'MIRAFLORES', '999-96673', 'e.chavez@gmail.com' );
insert into cliente values( '00008', 'FLORES', 'CHAFLOQUE', 'ROSA LIZET', '10773456',
'LIMA', 'LA MOLINA', '966-87567', 'r.florez@hotmail.com' );
insert into cliente values( '00009', 'FLORES', 'SHUTE', 'CRISTIAN RAFAEL', '10346723',
'LIMA', 'LOS OLIVOS', '978-43768', 'c.flores@hotmail.com' );
insert into cliente values( '00010', 'GONZALES', 'GARCIA', 'GABRIEL ALEJANDRO',
'10192376', 'LIMA', 'SAN MIGUEL', '945-56782', 'g.gonzales@yahoo.es' );
insert into cliente values( '00011', 'LAY', 'VALLEJOS', 'JUAN CARLOS', '10942287',
'LIMA', 'LINCE', '956-12657', 'j.lay@peru.com' );
insert into cliente values( '00012', 'MONTALVO', 'SOTO', 'DEYSI LIDIA', '10612376',
'LIMA', 'SURCO', '965-67235', 'd.montalvo@hotmail.com' );
insert into cliente values( '00013', 'RICALDE', 'RAMIREZ', 'ROSARIO ESMERALDA',
'10761324', 'LIMA', 'MIRAFLORES', '991-23546', 'r.ricalde@gmail.com' );
insert into cliente values( '00014', 'RODRIGUEZ', 'RAMOS', 'ENRIQUE MANUEL', '10773345',
'LIMA', 'LINCE', '976-82838', 'e.rodriguez@gmail.com' );
insert into cliente values( '00015', 'ROJAS', 'OSCANOA', 'FELIX NINO', '10238943',
'LIMA', 'LIMA', '962-32158', 'f.rojas@yahoo.com' );

```

```
insert into cliente values( '00016', 'TEJADA', 'DEL AGUILA', 'TANIA LORENA', '10446791',
'LIMA', 'PUEBLO LIBRE', '966-23854', 't.tejada@hotmail.com' );
insert into cliente values( '00017', 'VALDEVIESO', 'LEYVA', 'ROXANA', '10452682',
'LIMA', 'SURCO', '956-78951', 'r.valdivieso@terra.com.pe' );
insert into cliente values( '00018', 'VALENTIN', 'COTRINA', 'JUAN DIEGO', '10398247',
'LIMA', 'LA MOLINA', '921-12456', 'j.valentin@terra.com.pe' );
insert into cliente values( '00019', 'YAURICASA', 'BAUTISTA', 'YESABETH', '10934584',
'LIMA', 'MAGDALENA', '977-75777', 'y.yauricasa@terra.com.pe' );
insert into cliente values( '00020', 'ZEGARRA', 'GARCIA', 'FERNANDO MOISES', '10772365',
'LIMA', 'SAN ISIDRO', '936-45876', 'f.zegarra@hotmail.com' );

-- Tabla: Cuenta
insert into cuenta
values('00200001','01','002','0001','00008',7000,'20080105','ACTIVO',15,'123456');
insert into cuenta
values('00200002','01','002','0001','00001',6800,'20080109','ACTIVO',3,'123456');
insert into cuenta
values('00200003','02','002','0001','00007',6000,'20080111','ACTIVO',6,'123456');
insert into cuenta
values('00100001','01','001','0004','00005',6900,'20080106','ACTIVO',7,'123456');
insert into cuenta
values('00100002','02','001','0004','00005',4500,'20080108','ACTIVO',4,'123456');

-- Tabla: Movimiento
insert into movimiento values('00100002',01,'20080108','0004','001',1800,null);
insert into movimiento values('00100002',02,'20080125','0004','004',1000,null);
insert into movimiento values('00100002',03,'20080213','0004','003',2200,null);
insert into movimiento values('00100002',04,'20080308','0004','003',1500,null);

insert into movimiento values('00100001',01,'20080106','0004','001',2800,null);
insert into movimiento values('00100001',02,'20080115','0004','003',3200,null);
insert into movimiento values('00100001',03,'20080120','0004','004',0800,null);
insert into movimiento values('00100001',04,'20080214','0004','003',2000,null);
insert into movimiento values('00100001',05,'20080225','0004','004',0500,null);
insert into movimiento values('00100001',06,'20080303','0004','004',0800,null);
insert into movimiento values('00100001',07,'20080315','0004','003',1000,null);

insert into movimiento values('00200003',01,'20080111','0001','001',2500,null);
insert into movimiento values('00200003',02,'20080117','0001','003',1500,null);
insert into movimiento values('00200003',03,'20080120','0001','004',0500,null);
insert into movimiento values('00200003',04,'20080209','0001','004',0500,null);
insert into movimiento values('00200003',05,'20080225','0001','003',3500,null);
insert into movimiento values('00200003',06,'20080311','0001','004',0500,null);

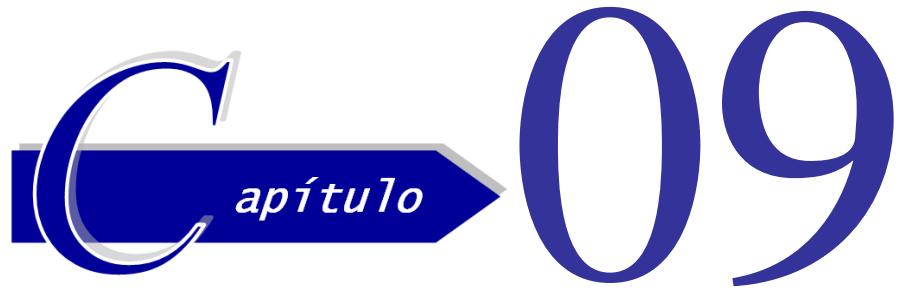
insert into movimiento values('00200002',01,'20080109','0001','001',3800,null);
insert into movimiento values('00200002',02,'20080120','0001','003',4200,null);
insert into movimiento values('00200002',03,'20080306','0001','004',1200,null);

insert into movimiento values('00200001',01,'20080105','0001','001',5000,null);
insert into movimiento values('00200001',02,'20080107','0001','003',4000,null);
insert into movimiento values('00200001',03,'20080109','0001','004',2000,null);
insert into movimiento values('00200001',04,'20080111','0001','003',1000,null);
insert into movimiento values('00200001',05,'20080113','0001','003',2000,null);
insert into movimiento values('00200001',06,'20080115','0001','004',4000,null);
insert into movimiento values('00200001',07,'20080119','0001','003',2000,null);
insert into movimiento values('00200001',08,'20080121','0001','004',3000,null);
insert into movimiento values('00200001',09,'20080123','0001','003',7000,null);
insert into movimiento values('00200001',10,'20080127','0001','004',1000,null);
insert into movimiento values('00200001',11,'20080130','0001','004',3000,null);
insert into movimiento values('00200001',12,'20080204','0001','003',2000,null);
insert into movimiento values('00200001',13,'20080208','0001','004',4000,null);
insert into movimiento values('00200001',14,'20080213','0001','003',2000,null);
insert into movimiento values('00200001',15,'20080219','0001','004',1000,null);

-- Tabla: Contador

insert into Contador Values( 'Moneda', 2, 2 );
insert into Contador Values( 'TipoMovimiento', 10, 3 );
insert into Contador Values( 'Sucursal', 7, 3 );
insert into Contador Values( 'Empleado', 11, 4 );
insert into Contador Values( 'Asignado', 11, 6 );
insert into Contador Values( 'Parametro', 2, 3 );
insert into Contador Values( 'Cliente', 21, 5 );
```

Ahora ya tenemos la base de datos lista para iniciar el diseño y desarrollo de la aplicación.



Consulta de Datos

Una de las tareas que mas realiza un sistema en la base de datos son las consultas, sin duda, es de vital importancia saber construirlas para obtener el resultado correcto.

Este capitulo aborda los fundamentos de como construir consultas simples.

Temas a desarrollar:

- 9.1. Fundamentos
- 9.2. Escribiendo Consultas Simples
- 9.3. Funciones Simples de Filas

9.1. Fundamentos

9.1.1. SQL Fundamentos

El lenguaje SQL esta dividido en cuatro sub-lenguajes:

- Lenguaje de Definición de Datos (DDL)
- Lenguaje de Manipulación de Datos (DML)
- Instrucciones de Control de Transacciones
- Instrucciones de Administración de la Base de Datos

9.1.1.1. Lenguaje de Definición de Datos (DDL)

Usado para crear, modificar, o eliminar objetos de base de datos, como por ejemplo tablas y vistas.

9.1.1.2. Lenguaje de Manipulación de Datos (DML)

Usado para acceder, crear, modificar, o eliminar data en una estructura de una base de datos existente.

9.1.1.3. Instrucciones de Control de Transacciones

Las instrucciones de control de transacciones garantizan la consistencia de los datos, organizando las instrucciones SQL en transacciones lógicas, que se completan o fallan como una sola unidad.

9.1.1.4. Instrucciones de Administración de la Base de Datos

Estas instrucciones permiten realizar tareas como:

- Crear nuevos usuarios
- Eliminar usuarios existentes
- Asignar privilegios sobre objetos
- Cambiar la contraseña de un usuario
- Optimizar una tabla
- etc.

9.1.2. Operadores y Literales

9.1.2.1. Operadores Aritméticos

Operador	Propósito	Ejemplo
+ -	Operadores unarios: Usado para representar datos positivos y negativos. Para datos positivos el operador + es opcional.	-234.56
+	Suma: Usado para sumar dos números o expresiones.	5 + 7
-	Resta: Usado para encontrar la diferencia entre dos números o expresiones.	56.8 - 18
*	Multiplicación: Usado para multiplicar dos números o expresiones.	7 * 15
/	División: Usado para dividir un número o expresión con otro.	8.67 / 3
DIV	División: Usado para realizar divisiones enteras.	15 div 4
%	Módulo: Usado para obtener el residuo de una división.	15 % 4

9.1.2.2. Precedencia de Operadores

Precedencia	Operador	Propósito
1	- +	Operadores unarios
2	* /	Multiplicación, división
3	+ -	Suma, resta,

9.1.2.3. Literales

Son valores que representan un valor fijo, y se ilustran en el siguiente cuadro.

Texto	<ul style="list-style-type: none"> ▪ ‘PeruDev’ ▪ ‘Nos vemos en Peter’s, la tienda del chino’ ▪ ‘El lenguaje es “PHP”, y lo dictan en CEPS-UNI’ ▪ ‘28-JUL-2006’ ▪ “Esto se pone bueno”
Entero	<ul style="list-style-type: none"> ▪ 45 ▪ -345

Coma Flotante	<ul style="list-style-type: none"> ▪ -456.78 ▪ 15E-15
Valores Lógicos	<ul style="list-style-type: none"> ▪ TRUE ▪ FALSE

9.1.2.4. Operadores de Comparación

Los operadores de comparación permiten comparar dos valores y el resultado es verdadero (TRUE) ó falso (FALSE).

La expresión que se construye con estos operadores se llama **Expresión Lógica Simple**.

Operador	Propósito	Ilustración
=	Compara dos valores y retorna TRUE si son iguales, FALSE en caso contrario.	A = B
<	Compara dos valores y retorna TRUE si el primero (A) es menor que el segundo (B), FALSE en caso contrario.	A < B
<=	Compara dos valores y retorna TRUE si el primero (A) es menor o igual que el segundo (B), FALSE en caso contrario.	A <= B
>	Compara dos valores y retorna TRUE si el primero (A) es mayor que el segundo (B), FALSE en caso contrario.	A > B
>=	Compara dos valores y retorna TRUE si el primero (A) es mayor o igual que el segundo (B), FALSE en caso contrario.	A >= B
<>, !=	Compara dos valores y retorna TRUE si ambos son diferentes, FALSE en caso contrario.	A <> B
<=>	A este operador se conoce como <i>Operador de Igualdad con NULL Seguro</i> , se utiliza para comparar valores nulos.	A <=> B

9.1.2.5. Operadores Lógicos

Estos operadores se utilizan para construir **Expresiones Lógicas Compuestas**.

Operador	Propósito	Ilustración
NOT, !	Niega un valor. La expresión toma valor TRUE si A es FALSE.	NOT (A)
AND, &&	La expresión toma valor TRUE si A y B son TRUE, en cualquier otro caso es FALSE	A AND B
OR,	La expresión toma valor TRUE si A es TRUE ó B es TRUE o A y B son TRUE, solo será FALSE cuando ambos sean FALSE.	A OR B
XOR	La expresión toma valor TRUE si A es TRUE ó B es TRUE, será FALSE cuando ambos sean TRUE ó FALSE.	A XOR B

Nota

El valor lógico TRUE se implementa como 1 y el valor lógico FALSE se implementa como 0, tal como se ilustra en el Ejemplo 9.1.

Ejemplo 9 . 1

```
mysql> select true, false;
+-----+-----+
| TRUE | FALSE |
+-----+-----+
|    1 |      0 |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.1.2.6. Reglas de Comparación

MySQL Sigue las siguientes reglas a la hora de comparar valores:

- Si uno o los dos valores a comparar son *NULL*, el resultado es *NULL*, excepto con el operador *<=>*, de comparación con *NULL* segura.
- Si los dos valores de la comparación son cadenas, se comparan como cadenas.
- Si ambos valores son enteros, se comparan como enteros.
- Los valores hexadecimales se tratan como cadenas binarias, si no se comparan como un número.
- Si uno de los valores es del tipo **TIMESTAMP** o **DATETIME** y el otro es una constante, la constante se convierte a **TIMESTAMP** antes de que se lleve a cabo la comparación.
- En el resto de los casos, los valores se comparan como números en coma flotante.

Ejemplo 9 . 2

En el presente script se ilustra el uso de operaciones con *NULL*.

Suma con valor *NULL*:

```
mysql> select 5 + 5 as sumal, 5 + NULL as suma2;
+-----+-----+
| sumal | suma2 |
+-----+-----+
|    10 |   NULL |
+-----+
1 row in set (0.02 sec)
```

Comparación con valor *NULL*:

```
mysql> select 20 = 20 as valor1, 20 = NULL as valor2;
+-----+-----+
| valor1 | valor2 |
+-----+-----+
|      1 |   NULL |
+-----+
1 row in set (0.02 sec)
```

Comparación con valor *NULL*:

```
mysql> select 20 <=> 20 as valor1, 20 <=> NULL as valor2;
+-----+-----+
| valor1 | valor2 |
+-----+-----+
|      1 |      0 |
+-----+
1 row in set (0.02 sec)

mysql>
```

9.2. Escribiendo Consultas Simples

9.2.1. Sintaxis Básica

```
SELECT lista_de_columnas
FROM nombre_tabla
WHERE condición_de_filtro
ORDER BY lista_de_columnas;
```

9.2.2. Usando la Sentencia SELECT

9.2.2.1. Consulta del contenido de una Tabla

El carácter asterisco (*) permite consultar todas las columnas de una tabla, tal como se ilustra en el Ejemplo 9.3.

Ejemplo 9 . 3

```
mysql> select * from moneda;
+-----+-----+
| chr_monecodigo | vch_moned descripcion |
+-----+-----+
| 01 | Soles |
| 02 | Dolares |
+-----+
2 rows in set (0.00 sec)

mysql>
```

9.2.2.2. Seleccionando Columnas

No se recomienda utilizar el carácter asterisco (*), se debe indicar explícitamente las columnas a consultar, tal como se ilustra en el Ejemplo 9.4.

Ejemplo 9 . 4

```
mysql> select chr_emplcodigo, vch_emplpaterno, vch_emplnombre from empleado;
+-----+-----+-----+
| chr_emplcodigo | vch_emplpaterno | vch_emplnombre |
+-----+-----+-----+
| 0001 | Romero | Carlos Alberto |
| 0002 | Castro | Lidia |
| 0003 | Reyes | Claudia |
| 0004 | Ramos | Angelica |
| 0005 | Ruiz | Claudia |
| 0006 | Cruz | Ricardo |
| 0007 | Diaz | Edith |
| 0008 | Sarmiento | Claudia Rocio |
| 0009 | Pachas | Luis Alberto |
| 0010 | Tello | Hugo Valentin |
| 0011 | Carrasco | Pedro Hugo |
| 9999 | Internet | internet |
+-----+-----+-----+
12 rows in set (0.00 sec)

mysql>
```

9.2.2.3. Alias para Nombres de Columnas

El título de una columna en el resultado por defecto es el mismo nombre de la columna, esto podemos cambiarlo haciendo uso de alias de columnas, su sintaxis se ilustra a continuación:

columna as alias

expresión as alias

Ejemplo 9 . 5

Usando un título simple:

```
mysql> select
    ->     chr_emplcodigo as codigo,
    ->     vch_emplpaterno as paterno,
    ->     vch_emplnombre as nombre
    -> from empleado;
+-----+-----+-----+
| codigo | paterno | nombre   |
+-----+-----+-----+
| 0001   | Romero  | Carlos Alberto |
| 0002   | Castro   | Lidia      |
| 0003   | Reyes   | Claudia    |
| 0004   | Ramos   | Angelica   |
| 0005   | Ruiz    | Claudia    |
| 0006   | Cruz    | Ricardo    |
| 0007   | Diaz    | Edith     |
| 0008   | Sarmiento | Claudia Rocio |
| 0009   | Pachas   | Luis Alberto |
| 0010   | Tello    | Hugo Valentin |
| 0011   | Carrasco  | Pedro Hugo   |
| 9999   | Internet | internet   |
+-----+-----+-----+
12 rows in set (0.00 sec)
```

Usando un título compuesto:

```
mysql> select
    ->     chr_cuencodigo as "nro. de cuenta",
    ->     dec_cuensaldo as "saldo de cuenta"
    -> from cuenta;
+-----+-----+
| nro. de cuenta | saldo de cuenta |
+-----+-----+
| 00100001       | 6900.00 |
| 00100002       | 4500.00 |
| 00200001       | 7000.00 |
| 00200002       | 6800.00 |
| 00200003       | 6000.00 |
+-----+-----+
5 rows in set (0.00 sec)
```

Usando expresiones:

```
mysql> select
    ->     chr_emplcodigo as codigo,
    ->     concat(vch_emplnombre,space(1),vch_emplpaterno) as nombre
    -> from empleado;
+-----+-----+
| codigo | nombre           |
+-----+-----+
| 0001   | Carlos Alberto Romero |
| 0002   | Lidia Castro          |
| 0003   | Claudia Reyes          |
| 0004   | Angelica Ramos         |
| 0005   | Claudia Ruiz           |
| 0006   | Ricardo Cruz            |
| 0007   | Edith Diaz             |
| 0008   | Claudia Rocio Sarmiento |
| 0009   | Luis Alberto Pachas    |
| 0010   | Hugo Valentin Tello    |
| 0011   | Pedro Hugo Carrasco    |
| 9999   | internet Internet      |
+-----+-----+
12 rows in set (0.00 sec)
```

```
mysql>
```

9.2.2.4. Asegurando Filas Únicas

Por defecto cuando realizamos una consulta se muestran todas las filas que retorna una consulta, si queremos eliminar las filas duplicadas debemos utilizar la cláusula DISTINCT, tal como se ilustra en el Ejemplo 9.6.

Ejemplo 9 . 6

Consulta mostrando las filas duplicadas:

```
mysql> select chr_sucucodigo from asignado;
+-----+
| chr_sucucodigo |
+-----+
| 001           |
| 001           |
| 002           |
| 002           |
| 003           |
| 003           |
| 004           |
| 004           |
| 005           |
| 005           |
| 006           |
| 006           |
+-----+
11 rows in set (0.00 sec)
```

Consulta eliminando las filas duplicadas:

```
mysql> select distinct chr_sucucodigo from asignado;
+-----+
| chr_sucucodigo |
+-----+
| 001           |
| 002           |
| 003           |
| 004           |
| 005           |
| 006           |
+-----+
6 rows in set (0.05 sec)

mysql>
```

9.2.3. Filtrando Filas

Cuando realiza consultas a la base de datos, debe realizarlas solo a los registros que necesita consultar, y no a toda la tabla, para eso debe construir una condición que permitan realizar los filtros necesarios, esta condición debe ir en la cláusula WHERE.

9.2.3.1. Operador de Igualdad (=)

Ejemplo 9 . 7

En este ejemplo se ilustra el uso del operador de **igualdad** para consultar las cuentas de la sucursal 001.

```
mysql> select chr_cuencodigo, chr_cliecodigo, dec_cuensaldo
-> from cuenta
-> where chr_sucucodigo = '001';
+-----+-----+-----+
| chr_cuencodigo | chr_cliecodigo | dec_cuensaldo |
+-----+-----+-----+
| 00100001      | 00005        |      6900.00 |
| 00100002      | 00005        |      4500.00 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql>
```

9.2.3.2. Operador Diferente (!=, <>)

Ejemplo 9 . 8

En este ejemplo se ilustra el uso del operador **diferente** para consultar las cuentas que no son de la sucursal 001.

```
mysql> select chr_cuencodigo, chr_cliecodigo, dec_cuensaldo
   -> from cuenta
   -> where chr_sucucodigo != '001';
+-----+-----+-----+
| chr_cuencodigo | chr_cliecodigo | dec_cuensaldo |
+-----+-----+-----+
| 00200001      | 00008          |    7000.00 |
| 00200002      | 00001          |    6800.00 |
| 00200003      | 00007          |    6000.00 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql>
```

9.2.3.3. Operador Menor Que (<)

Ejemplo 9 . 9

En el siguiente ejemplo se ilustra el uso del operador **menor que** para consultar las cuentas cuyo saldo es menor que 6500. Si quiere incluir las cuentas que tienen saldo 6500 debe utilizar el operador **menor ó igual que (<=)**.

```
mysql> select chr_cuencodigo, chr_cliecodigo, dec_cuensaldo
   -> from cuenta
   -> where dec_cuensaldo < 6500;
+-----+-----+-----+
| chr_cuencodigo | chr_cliecodigo | dec_cuensaldo |
+-----+-----+-----+
| 00100002      | 00005          |    4500.00 |
| 00200003      | 00007          |    6000.00 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql>
```

9.2.3.4. Operador Mayor Que (>)

Ejemplo 9 . 10

En el siguiente ejemplo se ilustra el uso del operador **mayor que** para consultar las cuentas cuyo saldo es mayor que 6500. Si quiere incluir las cuentas que tienen saldo 65000 debe utilizar el operador **mayor ó igual que (>=)**.

```
mysql> select chr_cuencodigo, chr_cliecodigo, dec_cuensaldo
   -> from cuenta
   -> where dec_cuensaldo > 6500;
+-----+-----+-----+
| chr_cuencodigo | chr_cliecodigo | dec_cuensaldo |
+-----+-----+-----+
| 00100001      | 00005          |    6900.00 |
| 00200001      | 00008          |    7000.00 |
| 00200002      | 00001          |    6800.00 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql>
```

9.2.3.5. Operador: IS [NOT] NULL

Ejemplo 9 . 11

El siguiente ejemplo ilustra la comprobación de valores nulos. La consulta muestra los empleados que han sido dados de baja.

```
mysql> select chr_sucucodigo,chr_emplcodigo,dtt_asigfechaalta,dtt_asigfechabaja
-> from asignado
-> where dtt_asigfechabaja is not null;
+-----+-----+-----+-----+
| chr_sucucodigo | chr_emplcodigo | dtt_asigfechaalta | dtt_asigfechabaja |
+-----+-----+-----+-----+
| 004           | 0003          | 2007-12-12      | 2008-03-25      |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

9.2.3.6. Operador: [NOT] BETWEEN exp_min AND exp_max

Ejemplo 9 . 12

Este operador permite verificar si un valor esta dentro de un rango. La consulta muestra las cuentas cuyo saldo esta entre 6000 y 7000.

```
mysql> select chr_cuencodigo, dec_cuensaldo
-> from cuenta
-> where dec_cuensaldo between 6000 and 7000;
+-----+-----+
| chr_cuencodigo | dec_cuensaldo |
+-----+-----+
| 00100001       | 6900.00   |
| 00200001       | 7000.00   |
| 00200002       | 6800.00   |
| 00200003       | 6000.00   |
+-----+-----+
4 rows in set (0.02 sec)

mysql>
```

9.2.3.7. Operador: [NOT] IN

Ejemplo 9 . 13

El operador IN permite verificar si un valor esta dentro de un conjunto de valores. Este ejemplo permite consultar las cuentas de las sucursales 001 y 002.

```
mysql> select chr_cuencodigo, chr_sucucodigo, dec_cuensaldo
-> from cuenta
-> where chr_sucucodigo in ( '001', '002' );
+-----+-----+-----+
| chr_cuencodigo | chr_sucucodigo | dec_cuensaldo |
+-----+-----+-----+
| 00100001       | 001           | 6900.00   |
| 00100002       | 001           | 4500.00   |
| 00200001       | 002           | 7000.00   |
| 00200002       | 002           | 6800.00   |
| 00200003       | 002           | 6000.00   |
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

9.2.3.8. Operador: NOT

Ejemplo 9 . 14

Este es un operador lógico que niega una condición. Si la condición es TRUE el resultado será FALSE y viceversa. La siguiente consulta muestra las cuentas cuyo saldo es mayor ó igual a 6500.

```
mysql> select chr_cuencodigo, dec_cuensaldo
-> from cuenta
-> where not ( dec_cuensaldo < 6500 );
+-----+-----+
| chr_cuencodigo | dec_cuensaldo |
+-----+-----+
| 00100001      |      6900.00 |
| 00200001      |      7000.00 |
| 00200002      |      6800.00 |
+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

9.2.3.9. Operador: AND

Ejemplo 9 . 15

Este es un operador lógico que combina en una sola expresión dos condiciones simples, tomará valor TRUE solo cuando ambas tomen valor TRUE. La siguiente consulta muestra las cuentas de la sucursal 002 cuyo saldo sea mayor que 6500.

```
mysql> select chr_cuencodigo, chr_sucucodigo, dec_cuensaldo
-> from cuenta
-> where chr_sucucodigo = '002' and dec_cuensaldo > 6500;
+-----+-----+-----+
| chr_cuencodigo | chr_sucucodigo | dec_cuensaldo |
+-----+-----+-----+
| 00200001      | 002           |      7000.00 |
| 00200002      | 002           |      6800.00 |
+-----+-----+-----+
2 rows in set (0.01 sec)

mysql>
```

9.2.3.10. Operador: OR

Ejemplo 9 . 16

Este es un operador lógico que combina en una sola expresión dos condiciones simples, tomará valor TRUE cuando una de las dos condiciones simples tome valor TRUE. Solo tomara valor FALSE cuando ambas condiciones simples tome valor FALSE.

La siguiente consulta muestra los clientes que tienen como uno de sus apellidos FLORES.

```
mysql> select
->     chr_cliecodigo, vch_cliepaterno,
->     vch_cliematerno, vch_clienombre
-> from cliente
-> where vch_cliepaterno = 'FLORES' or vch_cliematerno = 'FLORES';
+-----+-----+-----+-----+
| chr_cliecodigo | vch_cliepaterno | vch_cliematerno | vch_clienombre |
+-----+-----+-----+-----+
| 00008          | FLORES        | CHAFLOQUE     | ROSA LIZET      |
| 00009          | FLORES        | CASTILLO      | CRISTIAN RAFAEL |
| 00014          | RODRIGUEZ     | FLORES        | ENRIQUE MANUEL  |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

9.2.3.11. Operador: XOR

Ejemplo 9 . 17

Este es un operador lógico que combina en una sola expresión dos condiciones simples, tomará valor TRUE cuando **solo una** de las dos condiciones simples tome valor TRUE, caso contrario tomará valor FALSE.

La siguiente consulta muestra los clientes que su apellido paterno es FLORES o que su apellido materno es CASTILLO, pero no ambos.

```
mysql> select
->     chr_cliecodigo, vch_cliepaterno,
->     vch_cliematerno, vch_clienombre
-> from cliente
-> where vch_cliepaterno = 'FLORES' xor vch_cliematerno='CASTILLO';
+-----+-----+-----+-----+
| chr_cliecodigo | vch_cliepaterno | vch_cliematerno | vch_clienombre |
+-----+-----+-----+-----+
| 00001          | CORONEL        | CASTILLO       | ERIC GUSTAVO   |
| 00004          | ROMERO         | CASTILLO       | CARLOS ALBERTO |
| 00008          | FLORES         | CHAFLOQUE      | ROSA LIZET      |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

9.2.3.12. Operador: [NOT] LIKE

Ejemplo 9 . 18

Este operador se utiliza para comparar una cadena con un patrón, este patrón se construye en base a dos caracteres comodines que se explican a continuación:

% (Porcentaje)	Representa cualquier cadena de caracteres.
_ (Subrayado)	Representa un solo carácter.

La siguiente consulta muestra a los empleados cuyo nombre inicia con la letra M.

```
mysql> select vch_emplnombre, vch_emplpaterno
-> from empleado
-> where vch_emplnombre like 'C%';
+-----+-----+
| vch_emplnombre | vch_emplpaterno |
+-----+-----+
| Carlos Alberto | Romero        |
| Claudia        | Reyes          |
| Claudia        | Ruiz           |
| Claudia Rocio | Sarmiento     |
+-----+-----+
4 rows in set (0.00 sec)
```

La siguiente consulta muestra a todos los empleados cuyo nombre finaliza con la letra A.

```
mysql> select vch_emplnombre, vch_emplpaterno
-> from empleado
-> where vch_emplnombre like '%A';
+-----+-----+
| vch_emplnombre | vch_emplpaterno |
+-----+-----+
| Lidia          | Castro         |
| Claudia        | Reyes          |
| Angelica       | Ramos          |
| Claudia        | Ruiz           |
+-----+-----+
4 rows in set (0.00 sec)
```

La siguiente consulta muestra a los empleados que tienen a la letra A como segunda letra en su nombre.

```
mysql> select vch_emplnombre, vch_emplpaterno
   -> from empleado
   -> where vch_emplnombre like '_A%';
+-----+-----+
| vch_emplnombre | vch_emplpaterno |
+-----+-----+
| Carlos Alberto | Romero           |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

9.2.4. Ordenando Filas

Para ordenar un resultado debemos utilizar la cláusula ORDER BY, su sintaxis se ilustra a continuación.

Sintaxis:

```
ORDER BY {col_name|expr|position} [ASC|DESC], ...
```

Como podemos apreciar en la sintaxis se puede utilizar el nombre de una columna, una expresión o en su defecto la posición de la columna, y el ordenamiento puede ser en forma ascendente o descendente.

Ejemplo 9 . 19

Consultar las sucursales ordenadas por ciudad.

```
mysql> select chr_sucucodigo, vch_sucunombre, vch_sucuciudad
   -> from sucursal
   -> order by vch_sucuciudad;
+-----+-----+-----+
| chr_sucucodigo | vch_sucunombre | vch_sucuciudad |
+-----+-----+-----+
| 005            | Misti          | Arequipa      |
| 001            | Sipan          | Chiclayo      |
| 006            | Machupicchu    | Cusco         |
| 003            | Los Olivos     | Lima          |
| 004            | Pardo          | Lima          |
| 007            | Grau           | Piura         |
| 002            | Chan Chan     | Trujillo      |
+-----+-----+-----+
7 rows in set (0.00 sec)
```

Consultar los empleados del departamento 103 ordenadas por nombre:

```
mysql> select vch_cliedireccion, vch_clienombre, vch_cliepaterno
   -> from cliente
   -> where vch_cliedireccion = 'LINCE'
   -> order by vch_clienombre;
+-----+-----+-----+
| vch_cliedireccion | vch_clienombre | vch_cliepaterno |
+-----+-----+-----+
| LINCE            | ENRIQUE MANUEL | RODRIGUEZ      |
| LINCE            | JUAN CARLOS    | LAY            |
| LINCE            | RICARDO        | MARCELO       |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

9.2.5. Uso de LIMIT

LIMIT no es un operador, es más bien parte de la sintaxis de la sentencia SELECT.

Sintaxis

```
LIMIT [desplazamiento,] número_filas
```

La cláusula LIMIT puede ser usada para limitar el número de filas devuelto por la sentencia SELECT. LIMIT toma uno o dos argumentos numéricos, que deben ser constantes enteras. Con dos argumentos, el primero especifica el desplazamiento de la primera fila a devolver, el segundo especifica el máximo número de filas a devolver. El desplazamiento de la fila inicial es 0 (no 1).

Ejemplo 9 . 20

Consultar los últimos 5 movimientos de la cuenta 00200001:

```
mysql> select
->     int_movinumero, dtt_movifecha,
->     chr_tipocodigo, dec_moviimporte
->   from movimiento
->  where chr_cuencodigo = '00200001'
->  order by 1 desc
->  limit 5;
+-----+-----+-----+-----+
| int_movinumero | dtt_movifecha | chr_tipocodigo | dec_moviimporte |
+-----+-----+-----+-----+
|      15 | 2008-02-19    | 004          | 1000.00        |
|      14 | 2008-02-13    | 003          | 2000.00        |
|      13 | 2008-02-08    | 004          | 4000.00        |
|      12 | 2008-02-04    | 003          | 2000.00        |
|      11 | 2008-01-30    | 004          | 3000.00        |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

Consultar el penúltimo movimiento de la cuenta 00200001:

```
mysql> select
->     int_movinumero, dtt_movifecha,
->     chr_tipocodigo, dec_moviimporte
->   from movimiento
->  where chr_cuencodigo = '00200001'
->  order by 1 desc
->  limit 1, 1;
+-----+-----+-----+-----+
| int_movinumero | dtt_movifecha | chr_tipocodigo | dec_moviimporte |
+-----+-----+-----+-----+
|      14 | 2008-02-13    | 003          | 2000.00        |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

9.3. Funciones Simples de Filas

9.3.1. Funciones de Control de Flujo

9.3.1.1. Función CASE

Caso 1

Según el valor que toma una columna o expresión puede tomar diferentes valores.

Sintaxis:

```
CASE {columna|<expresión>}
      WHEN <Valor1> THEN <Valor de Retorno 1>
      WHEN <Valor2> THEN <Valor de Retorno 2>
      .
      .
      .
      [ELSE <Valor de Retorno>]
END
```

Ejemplo 9 . 21

En el siguiente ejemplo se modifica la columna ciudad, si se trata de lima queda igual, en caso contrario cambia por **provincia**.

```
mysql> select chr_sucucodigo,
->       case vch_sucuciudad
->           when 'Lima' then 'Lima'
->           else 'Provincia'
->       end as Ciudad
->     from sucursal;
+-----+-----+
| chr_sucucodigo | Ciudad   |
+-----+-----+
| 001            | Provincia |
| 002            | Provincia |
| 003            | Lima      |
| 004            | Lima      |
| 005            | Provincia |
| 006            | Provincia |
| 007            | Provincia |
+-----+-----+
7 rows in set (0.00 sec)
```

```
mysql>
```

Caso 2

En función a condiciones se puede tomar diferentes valores.

Sintaxis

```
CASE
    WHEN <Condición1> THEN <Valor de Retorno 1>
    WHEN <Condición2> THEN <Valor de Retorno 2>
    WHEN <Condición3> THEN <Valor de Retorno 3>
    .
    .
    .
    [ELSE <Valor de Retorno>]
END
```

Ejemplo 9 . 22

En el siguiente ejemplo se calificará el saldo de los cuentas según el siguiente cuadro:

Saldo	Calificación
[0, 2500>	Bajo
[2500, 5000>	Regular
[5000, 10000>	Bueno
[10000, 15000>	Muy Bueno
[15000, ∞>	Excelente

La consulta es la siguiente:

```
mysql> select chr_cuencodigo, dec_cuensaldo,
->      case
->          when dec_cuensaldo < 2500 then 'Bajo'
->          when dec_cuensaldo < 5000 then 'Regular'
->          when dec_cuensaldo < 10000 then 'Bueno'
->          when dec_cuensaldo < 15000 then 'Muy Bueno'
->          else 'Excelente'
->      end as calificacion
->  from cuenta;
+-----+-----+-----+
| chr_cuencodigo | dec_cuensaldo | calificacion |
+-----+-----+-----+
| 00100001      |    6900.00   | Bueno        |
| 00100002      |    4500.00   | Regular      |
| 00200001      |    7000.00   | Bueno        |
| 00200002      |    6800.00   | Bueno        |
| 00200003      |    6000.00   | Bueno        |
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

9.3.1.2. Función IF()

Sintaxis

IF(expr1, expr2, expr3)

Si expr1 es TRUE retorna expr2, de lo contrario retorna expr3.

Ejemplo 9 . 23

En el siguiente ejemplo se evaluará el empleado, si esta activo o se le ha dado de baja.

```
mysql> select chr_emplcodigo, chr_sucucodigo,
->      if( dtt_asigfechabaja is null, 'Activo', 'Baja' ) as Estado
->  from asignado;
+-----+-----+-----+
| chr_emplcodigo | chr_sucucodigo | Estado |
+-----+-----+-----+
| 0004           | 001            | Activo  |
| 0001           | 002            | Activo  |
| 0002           | 003            | Activo  |
| 0003           | 004            | Baja    |
| 0006           | 005            | Activo  |
| 0005           | 006            | Activo  |
| 0007           | 004            | Activo  |
| 0008           | 005            | Activo  |
| 0011           | 001            | Activo  |
| 0009           | 002            | Activo  |
| 0010           | 006            | Activo  |
```

```
+-----+-----+-----+
11 rows in set (0.01 sec)

mysql>
```

9.3.1.3. Función IFNULL()

Sintaxis

```
IFNULL(expr1,expr2)
```

Si expr1 no es NULL retorna expr1, pero si es NULL retorna expr2.

Ejemplo 9 . 24

En este ejemplo se evalúa la fecha de baja del empleado, si es nulo se reemplaza por una cadena en blanco.

```
mysql> select chr_emplcodigo, chr_sucucodigo,
->      ifnull( dtt_asigfechabaja, '' ) as FechaBaja
->      from asignado;
+-----+-----+-----+
| chr_emplcodigo | chr_sucucodigo | FechaBaja   |
+-----+-----+-----+
| 0004           | 001            |             |
| 0001           | 002            |             |
| 0002           | 003            |             |
| 0003           | 004            | 2008-03-25 |
| 0006           | 005            |             |
| 0005           | 006            |             |
| 0007           | 004            |             |
| 0008           | 005            |             |
| 0011           | 001            |             |
| 0009           | 002            |             |
| 0010           | 006            |             |
+-----+-----+-----+
11 rows in set (0.01 sec)
```

```
mysql>
```

9.3.1.4. Función NULLIF()

Sintaxis

```
IFNULL(expr1,expr2)
```

Si expr1 = expr2 es TRUE, retorna NULL, en caso contrario retorna expr1. Esto es lo mismo que CASE WHEN x = y THEN NULL ELSE x END.

Ejemplo 9 . 25

El siguiente script ilustra el funcionamiento de la función NULLIF.

```
mysql> select nullif( 8, 8 );
+-----+
| nullif( 8, 8 ) |
+-----+
|          NULL |
+-----+
1 row in set (0.02 sec)

mysql> select nullif( 8, 10 );
+-----+
| nullif( 8, 10 ) |
+-----+
|          8    |
+-----+
1 row in set (0.01 sec)
```

mysql>

9.3.2. Funciones de Cadena

Función	Descripción	Ejemplo
ASCII	Retorna el valor ASCII equivalente de un carácter.	ascii('A') = 65
CHR	Retorna el carácter determinado por el valor ASCII equivalente.	chr(65) = A
CONCAT	Concatena dos o más cadenas.	concat('Gustavo','Coronel') = GustavoCoronel
INSERT	Inserta una subcadena dentro de una cadena.	insert('Alianza',3,2,'aba') = alabanza
INSTR	Busca la posición de inicio de una cadena dentro de otra.	instr('Mississippi','i') = 2 instr('Mississippi','alfa') = 0
LCASE	Sinónimo de LOWER.	lcase('UNI') = uni
LEFT	Retorna los "n" primeros carácter de una cadena.	left('perudev',4) = peru
LENGTH	Retorna la longitud de una cadena en caracteres.	length('MySQL es Powerful') = 17
LOCATE	Busca la posición de inicio de una cadena dentro de otra.	locate('i','Mississippi') = 2 locate('s','Mississippi',5) = 6
LOWER	Convierte una cadena a minúsculas.	lower('PACHERREZ') = pacherrez
LPAD	Ajusta a la derecha una cadena, rellenándola a la izquierda con otra cadena	lpad('56.78',8,'#') = ##56.78
LTRIM	Elimina caracteres a la izquierda de una cadena, por defecto espacios en blanco.	ltrim(' Alianza') = Alianza
MID	Permite extraer parte de una cadena. Es sinónimo de SUBSTRING.	mid('naturalmente',8) = mente mid('desmantelado',4,6) = mantel
POSITION	Sinónimo de LOCATE.	position('mantel' IN 'desmantelado') = 4
QUOTE	Entrecorra una cadena para producir un resultado que se pueda utilizar correctamente en una instrucción SQL.	quote("Don't") = 'Don\'t'
REPEAT	Repite una cadena cierta cantidad de veces.	repeat("Peru",3) = PeruPeruPeru
REPLACE	Permite reemplaza parte de una cadena.	replace('la triste noche del triste dia','triste','alegre') = la alegra noche del alegra dia Replace('PagDown','Down','Up') = PagUp
REVERSE	Invierte una cadena.	reverse('pacherrez') = zerrehcap

Función	Descripción	Ejemplo
RIGHT	Retorna los "n" últimos carácter de una cadena.	right('PeruDev',3) = Dev
RPAD	Ajusta a la izquierda una cadena, rellenándola a la derecha con otra cadena.	rpad('56.78',8,'#') = 56.78###
RTRIM	Elimina los espacios en blanco a la derecha de una cadena.	rtrim('EGCC ') = EGCC
SOUNDEX	Retorna la representación fonética de una cadena.	Soundex('HOLA') = H400
SPACE	Retorna una cadena conformada por "n" espacios en blanco.	concat('Hola',space(1),'Peru') = Hola Peru
STRCMP	Compara dos cadenas, devuelve 0 si las cadenas son iguales, -1 si la primera es menor que la segunda, y 1 en otro caso.	strcmp('egcc', 'egcc2') = -1 strcmp('egcc1', 'egcc') = 1 strcmp('egcc', 'egcc') = 0
SUBSTRING	Permite extraer parte de una cadena.	substring('TrujilloChiclayoLima',9,8) = Chiclayo
TRIM	Elimina caracteres de los extremos de una cadena.	trim(both 'a' from 'aaEGCCaa') = EGCC
UCASE	Convierte una cadena a mayúsculas.	ucase('peru') = PERU
UPPER	Convierte una cadena a mayúsculas.	upper('chiclayo') = CHICLAYO

Ejemplo 9 . 26

En el siguiente ejemplo se ilustra el uso de funciones de cadena para juntar los nombres y apellidos de un empleado en una sola columna.

```
mysql> select
->     concat(vch_crienombre,space(1),
->             vch_cliepaterno,space(1),vch_cliematerno) as cliente
->     from cliente;
+-----+
| cliente          |
+-----+
| ERIC GUSTAVO CORONEL CASTILLO |
| PEDRO HUGO VALENCIA MORALES   |
| RICARDO MARCELO VILLALOBOS    |
| CARLOS ALBERTO ROMERO CASTILLO |
| ALAN ALBERTO ARANDA LUNA      |
| JORGE LUIS AYALA PAZ           |
| EDGAR RAFAEL CHAVEZ CANALES   |
| ROSA LIZET FLORES CHAFLOQUE   |
| CRISTIAN RAFAEL FLORES CASTILLO |
| GABRIEL ALEJANDRO GONZALES GARCIA |
| JUAN CARLOS LAY VALLEJOS      |
| DEYSI LIDIA MONTALVO SOTO     |
| ROSARIO ESMERALDA RICALDE RAMIREZ |
| ENRIQUE MANUEL RODRIGUEZ FLORES  |
| FELIX NINO ROJAS OSCANOA      |
| TANIA LORENA TEJADA DEL AGUILA  |
| LIDIA ROXANA VALDEVIESO LEYVA   |
| JUAN DIEGO VALENTIN COTRINA    |
| YESABETH YAURICASA BAUTISTA    |
| FERNANDO MOISES ZEGARRA GARCIA  |
+-----+
20 rows in set (0.00 sec)
```

```
mysql>
```

9.3.3. Funciones Numéricas

Función	Descripción	Ejemplo
ABS	Retorna el valor absoluto de un número.	abs(-5) = 5
ACOS	Retorna el arco coseno.	acos(-1) = 3.14159265
ASIN	Retorna el arco seno.	asin(1) = 1.57079633
ATAN	Retorna el arco tangente.	atan(0) = 0
ATAN2	Retorna el arco tangente; tiene dos valores de entrada.	atan2(0,3.1415) = 0
CEIL CEILING	Retorna el siguiente entero más alto.	ceil(5.1) = 6
COS	Retorna el coseno de un ángulo.	cos(0) = 1
COT	Retorna la cotangente de un ángulo.	cot(pi()/4) = 1
DEGREES	Convierte radianes a grados.	degrees(pi()) = 180
DIV	Operador de división entera.	15 div 7 = 2
EXP	Retorna la base del logaritmo natural elevado a una potencia.	exp(1) = 2.71828183
FLOOR	Retorna el siguiente entero más pequeño.	floor(5.31) = 5
GREATEST	Retorna el mayor de "n" argumentos.	greatest(15,13,18,12) = 18
LEAST	Retorna el menor de "n" argumentos.	least(15,13,18,12) = 12
LN	Retorna el logaritmo natural.	ln(2.7) = 0.99325177
LOG	Retorna el logaritmo.	log(8,64) = 2
LOG10	Retorna el logaritmo de base 10.	log10(100) = 2
LOG2	Retorna el logaritmo de base 2.	log2(8) = 3
MOD	Retorna el residuo de una operación de división.	mod(13,5) = 3
PI	Retorna el valor de π (Pi).	pi() = 3.141593
POW	Retorna un número elevado a una potencia.	pow(2,3) = 8
POWER	Retorna un número elevado a una potencia.	power(2,3) = 8
RADIANS	Convierte grados a radianes-	radians(180) = 3.1415926535898
RAND	Retorna un número aleatorio entre 0 y 1.	rand()
ROUND	Redondea un número.	round(5467,-2) = 5500 round(56.7834,2) = 56.78
SIGN	Retorna el indicador de signo de un número.	Sign(-456) = -1

Función	Descripción	Ejemplo
SIN	Retorna el seno de un ángulo.	<code>Sin(0) = 0</code>
SQRT	Retorna la raíz cuadrada de un número.	<code>sqrt(16) = 4</code>
TAN	Retorna la tangente de un ángulo.	<code>tan(pi()/4) = 1</code>
TRUNC	Trunca un número.	<code>Trunc(456.678,2) = 456.67</code> <code>Trunc(456.678,-1) = 450</code>

9.3.4. Funciones de Fecha

9.3.4.1. Formato de Fecha

Cuando manipulemos datos de tipo fecha debemos utilizar el formato que se ilustra a continuación:

Tipo de Dato	Formato
Date	'YYYY-MM-DD'
DateTime	'YYYY-MM-DD HH:MM:SS'

9.3.4.2. Formatos de Intervalos

En muchas funciones se encontrará en su sintaxis;

INTERVAL expr type

El valor esperado para **type** y el formato de **expr** se ilustran en la siguiente tabla.

Valor de type	Formato de expr
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MICROSECOND	'HOURS.MICROSECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'

Valor de type	Formato de expr
DAY_MICROSECOND	'DAYS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

9.3.4.3. Función: ADDDATE()

Se utiliza para agregar a una fecha un intervalo de tiempo.

Sintaxis

```
ADDDATE( date, INTERVAL expr type )
ADDDATE( date, days )
```

Ejemplo 9 . 27

Agregar días a una fecha:

```
mysql> select
    ->     curdate() as Hoy,
    ->     adddate(curdate(),1) as maniana;
+-----+-----+
| Hoy      | maniana   |
+-----+-----+
| 2008-08-09 | 2008-08-10 |
+-----+-----+
1 row in set (0.00 sec)
```

Agregar años a una fecha:

```
mysql> select
    ->     curdate() as Hoy,
    ->     adddate(curdate(), interval 1 year) as sgte_anio;
+-----+-----+
| Hoy      | sgte_anio  |
+-----+-----+
| 2008-08-09 | 2009-08-09 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.4. Funciones: CURDATE(), CURRENT_DATE, CURRENT_DATE()

Retorna la fecha actual del sistema.

Sintaxis

```
CURDATE()
CURRENT_DATE
CURRENT_DATE()
```

Ejemplo 9 . 28

En este script se muestra la fecha actual del sistema.

```
mysql> select curdate();
+-----+
| curdate() |
+-----+
| 2008-08-09 |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.5. Funciones: CURTIME(), CURRENT_TIME, CURRENT_TIME()

Retorna la hora actual del sistema.

Sintaxis

```
CURTIME()
CURRENT_TIME
CURRENT_TIME()
```

Ejemplo 9 . 29

En este script se muestra la hora actual del sistema.

```
mysql> select curtime();
+-----+
| curtime() |
+-----+
| 07:11:29 |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.6. Funciones: NOW(), CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(), LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP(), SYSDATE()

Estas funciones retornan la fecha y hora del sistema.

Sintaxis

```
NOW()
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP()
LOCALTIME
LOCALTIME()
LOCALTIMESTAMP
LOCALTIMESTAMP()
SYSDATE()
```

Ejemplo 9 . 30

En el siguiente script se muestra la fecha y hora actual del sistema.

```
mysql> select now();
+-----+
| now()           |
+-----+
| 2008-08-09 07:14:45 |
+-----+
```

```

1 row in set (0.00 sec)

mysql> select localtime;
+-----+
| localtime          |
+-----+
| 2008-08-09 07:15:01 |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.7. Función: DATE()

Extrae la fecha de una expresión de tipo datetime.

Sintaxis

`DATE(expr)`

Ejemplo 9 . 31

El siguiente ejemplo muestra la fecha actual del sistema.

```

mysql> select date( sysdate() );
+-----+
| date( sysdate() ) |
+-----+
| 2008-08-09         |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.8. Función: DATEDIFF()

Retorna la diferencia en días entre dos fecha.

Sintaxis

`DATEDIFF(expr, expr2)`

Ejemplo 9 . 32

En el siguiente ejemplo se ilustra el uso de la función datediff.

```

mysql> select datediff('2008-08-09','2007-08-09');
+-----+
| datediff('2008-08-09','2007-08-09') |
+-----+
|           366                         |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.9. Funciones: DATE_ADD(), DATE_SUB()

Estas funciones realizan operaciones aritmética con fecha, también se pueden utilizar los operadores + y -.

Sintaxis

`DATE_ADD(date,INTERVAL expr type)`
`DATE_SUB(date,INTERVAL expr type)`

Ejemplo 9 . 33

Agregar días a una fecha.

```
mysql> select date_add('2008-08-09',interval 5 day);
+-----+
| date_add('2008-08-09',interval 5 day) |
+-----+
| 2008-08-14                                |
+-----+
1 row in set (0.00 sec)

mysql> select '2008-08-09' + interval 5 day;
+-----+
| '2008-08-09' + interval 5 day |
+-----+
| 2008-08-14                                |
+-----+
1 row in set (0.01 sec)

mysql>
```

Restar días a una fecha.

```
mysql> select date_sub('2008-08-09',interval 5 day);
+-----+
| date_sub('2008-08-09',interval 5 day) |
+-----+
| 2008-08-04                                |
+-----+
1 row in set (0.00 sec)

mysql> select '2008-08-09' - interval 5 day;
+-----+
| '2008-08-09' - interval 5 day |
+-----+
| 2008-08-04                                |
+-----+
1 row in set (0.01 sec)

mysql>
```

9.3.4.10. Función: DATE_FORMAT()

Formatea el valor de una fecha de acuerdo con un formato. Dentro del formato debemos utilizar **patrones de formato**.

Sintaxis

```
DATE_FORMAT(fecha,formato)
```

El siguiente es el cuadro de patrones de formato.

Patrón	Descripción
%M	Nombre del mes (January..December)
%W	Nombre de día (Sunday..Saturday)
%D	Día del mes con sufijo en inglés (0th, 1st, 2nd, 3rd, etc.)
%Y	Año, numérico con 4 dígitos
%y	Año, numérico con 2 dígitos
%X	Año para la semana donde el domingo es el primer día de la semana, numérico de 4 dígitos; usado junto con %V

Patrón	Descripción
%x	Año para la semana donde el lunes es el primer día de la semana, numérico de 4 dígitos; usado junto con %v
%a	Nombre de día de semana abreviado (Sun..Sat)
%d	Día del mes, numérico (00..31)
%e	Día del mes, numérico (0..31)
%m	Mes, numérico (00..12)
%c	Mes, numérico (0..12)
%b	Nombre del mes abreviado (Jan..Dec)
%j	Día del año (001..366)
%H	Hora (00..23)
%k	Hora (0..23)
%h	Hora (01..12)
%l	Hora (01..12)
%l	Hora (1..12)
%i	Minutos, numérico (00..59)
%r	Tiempo, 12-horas (hh:mm:ss seguido por AM o PM)
%T	Tiempo, 24-horas (hh:mm:ss)
%S	Segundos (00..59)
%s	Segundos (00..59)
%f	Microsegundos (000000..999999)
%p	AM o PM
%w	Día de la semana (0=Sunday..6=Saturday)
%U	Semana (00..53), donde el domingo es el primer día de la semana
%u	Semana (00..53), donde el lunes es el primer día de la semana
%V	Semana (01..53), donde el domingo es el primer día de la semana; usado con %X
%v	Semana (01..53), donde el lunes es el primer día de la semana; usado con %X
%%	Un '%' literal.

Ejemplo 9 . 34

En el siguiente ejemplo se ilustra el uso de la función date_format.

```
mysql> select date_format( sysdate(), '%Y-%m-%d' );
+-----+
| date_format( sysdate(), '%Y-%m-%d' ) |
+-----+
| 2008-08-09 |
+-----+
1 row in set (0.04 sec)

mysql> select date_format( sysdate(), '%y-%m-%d' );
```

```
+-----+  
| date_format( sysdate(), '%y-%m-%d' ) |  
+-----+  
| 08-08-09 |  
+-----+  
1 row in set (0.01 sec)  
  
mysql>
```

9.3.4.11. Función: DAYOFMONTH(), DAY()

Retorna el día del mes de una fecha.

Sintaxis

```
DAYOFMONTH(date)  
DAY(date)
```

Ejemplo 9 . 35

En este ejemplo se ilustra el uso de la función day().

```
mysql> select now(), day( now() );  
+-----+-----+  
| now() | day( now() ) |  
+-----+-----+  
| 2008-08-09 08:50:15 | 9 |  
+-----+-----+  
1 row in set (0.00 sec)  
  
mysql>
```

9.3.4.12. Función: DAYNAME()

Retorna el nombre del día de la semana.

Sintaxis

```
DAYNAME(date)
```

Ejemplo 9 . 36

Ejemplo que ilustra el uso de dayname().

```
mysql> select curdate(), dayname(curdate());  
+-----+-----+  
| curdate() | dayname(curdate()) |  
+-----+-----+  
| 2008-08-09 | Saturday |  
+-----+-----+  
1 row in set (0.00 sec)  
  
mysql>
```

9.3.4.13. Función: DAYOFWEEK()

Devuelve el día de la semana para una fecha dada (1 = Domingo, 2 = Lunes, . . . 7 = Sábado).

Sintaxis

```
DAYOFWEEK(date)
```

Ejemplo 9 . 37

Ejemplo donde se ilustra el funcionamiento de dayofweek().

```
mysql> select curdate(), dayofweek( curdate() );
+-----+-----+
| curdate() | dayofweek( curdate() ) |
+-----+-----+
| 2008-08-09 | 6 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.14. Función: DAYOFYEAR()

Retorna el día del año.

Sintaxis

```
DAYOFYEAR(date)
```

Ejemplo 9 . 38

Ejemplo donde se ilustra el funcionamiento de dayofyear().

```
mysql> select curdate(), dayofyear( curdate() );
+-----+-----+
| curdate() | dayofyear( curdate() ) |
+-----+-----+
| 2008-08-09 | 222 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.15. Función: EXTRACT()

Extrae y retorna un componente de una expresión tipo fecha.

Sintaxis

```
EXTRACT( type FROM date)
```

Ejemplo 9 . 39

Ejemplo que ilustra el funcionamiento de extract.

```
mysql> select curdate(), extract( year from curdate() );
+-----+-----+
| curdate() | extract( year from curdate() ) |
+-----+-----+
| 2008-08-09 | 2008 |
+-----+-----+
1 row in set (0.01 sec)

mysql> select curdate(), extract( month from curdate() );
+-----+-----+
| curdate() | extract( month from curdate() ) |
+-----+-----+
| 2008-08-09 | 8 |
+-----+-----+
1 row in set (0.01 sec)

mysql> select curdate(), extract( day from curdate() );
+-----+-----+
| curdate() | extract( day from curdate() ) |
+-----+-----+
| 2008-08-09 | 9 |
+-----+-----+
```

```
1 row in set (0.00 sec)

mysql>
```

9.3.4.16. Función: GET_FORMAT()

Devuelve una cadena de formato.

Sintaxis

```
GET_FORMAT( DATE | TIME | TIMESTAMP,
            'EUR' | 'USA' | 'JIS' | 'ISO' | 'INTERNAL')
```

Ejemplo 9 . 40

Script ilustrativo de getformat.

```
mysql> select get_format( date, 'USA' );
+-----+
| get_format( date, 'USA' ) |
+-----+
| %m.%d.%Y
+-----+
1 row in set (0.05 sec)

mysql> select get_format( date, 'ISO' );
+-----+
| get_format( date, 'ISO' ) |
+-----+
| %Y-%m-%d
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.17. Función: LAST_DAY()

Retorna el último día del mes para una fecha dada.

Sintaxis

```
LAST_DAY(date)
```

Ejemplo 9 . 41

Ejemplo ilustrativo de last_day.

```
mysql> select curdate(), last_day( curdate() );
+-----+
| curdate() | last_day( curdate() ) |
+-----+
| 2008-08-09 | 2008-08-31
+-----+
1 row in set (0.01 sec)

mysql>
```

9.3.4.18. Función: MONTH()

Retorna el mes de una fecha

Sintaxis

MONTH(date)

Ejemplo 9 . 42

Ejemplo que ilustra el uso de la función month.

```
mysql> select curdate(), month(curdate());
+-----+-----+
| curdate() | month(curdate()) |
+-----+-----+
| 2008-08-09 |          8 |
+-----+-----+
1 row in set (0.02 sec)

mysql>
```

9.3.4.19. Función: MONTHNAME()

Retorna el nombre de mes de una fecha.

Sintaxis

MONTHNAME(date)

Ejemplo 9 . 43

Ejemplo que ilustra el uso de monthname.

```
mysql> select curdate(), monthname(curdate());
+-----+-----+
| curdate() | monthname(curdate()) |
+-----+-----+
| 2008-08-09 | August           |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.20. Función: WEEK()

Retorna el número de semana para una fecha dada.

Sintaxis

```
WEEK(date [,mode])
```

El parámetro **mode** determina el día en que empieza la semana, su valor por defecto es cero, que significa que el rango es de 0 – 53 y la semana empieza el Domingo.

La siguiente tabla muestra los valores que puede tomar el parámetro **mode** y su significado.

Mode	Primer día de la semana	Rango	Semana 1 es la primera semana ...
0	Domingo	0-53	con un domingo en este año.
1	Lunes	0-53	con mas de 3 días este año.
2	Domingo	1-53	con un domingo en este año.
3	Lunes	1-53	con mas de 3 días este año.
4	Domingo	0-53	con mas de 3 días este año.
5	Lunes	0-53	con un lunes en este año.
6	Domingo	1-53	con mas de 3 días este año.
7	Lunes	1-53	con un lunes en este año.

Ejemplo 9 . 44

Ejemplo ilustrativo de la función week.

```
mysql> select week( '2008-01-01', 0 );
+-----+
| week( '2008-01-01', 0 ) |
+-----+
| 0 |
+-----+
1 row in set (0.00 sec)

mysql> select week( '2008-01-07', 0 );
+-----+
| week( '2008-01-07', 0 ) |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.21. Función: WEEKDAY()

Retorna el día de la semana para una fecha, 0 = Lunes, 1 = Martes, . . . 6 = Domingo.

Sintaxis

```
WEEKDAY(date)
```

Ejemplo 9 . 45

Ejemplo ilustrativo de la función weekday.

```
mysql> select weekday( '2008-01-2' );
```

```
+-----+
| weekday( '2008-01-2' ) |
+-----+
| 2 |
+-----+
1 row in set (0.00 sec)

mysql> select weekday( '2008-01-7' );
+-----+
| weekday( '2008-01-7' ) |
+-----+
| 0 |
+-----+
1 row in set (0.02 sec)

mysql>
```

9.3.4.22. Función: WEEKOFYEAR()

Retorna el número de semana según el calendario de la fecha dada como un número en el rango de 1 a 53.

Sintaxis

```
WEEKOFYEAR(date)
```

Ejemplo 9 . 46

Ejemplo ilustrativo de la función weekofyear.

```
mysql> select weekofyear( '2008-01-05' );
+-----+
| weekofyear( '2008-01-05' ) |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.4.23. Función: YEAR()

Retorna el año de una fecha.

Sintaxis

```
YEAR(date)
```

Ejemplo 9 . 47

Ejemplo ilustrativo de la función year.

```
mysql> select curdate(), year(curdate());
+-----+-----+
| curdate() | year(curdate()) |
+-----+-----+
| 2008-08-09 | 2008 |
+-----+-----+
1 row in set (0.01 sec)

mysql>
```

9.3.5. Funciones de Conversión

9.3.5.1. Tipos de conversión

Los tipos de conversión que se pueden utilizar son:

- BINARY
- CHAR
- DATE
- DATETIME
- SIGNED {INTEGER}
- TIME
- UNSIGNED {INTEGER}

9.3.5.2. Función: Cast()

Convierte una expresión a un tipo de dato específico.

Sintaxis

```
CAST(expression AS type)
```

Ejemplo 9 . 48

Ejemplo ilustrativo de la función cast.

```
mysql> select cast( curdate() as char );
+-----+
| cast( curdate() as char ) |
+-----+
| 2008-08-09                 |
+-----+
1 row in set (0.01 sec)

mysql>
```

9.3.5.3. Función: CONVERT()

Convierte una expresión a un tipo de dato específico.

Sintaxis

```
CONVERT(expression,type)
CONVERT(expr USING transcoding_name)
```

Ejemplo 9 . 49

Ejemplo ilustrativo de la función convert.

```
mysql> select convert( curdate(), char );
+-----+
| convert( curdate(), char ) |
+-----+
| 2008-08-09                 |
+-----+
1 row in set (0.01 sec)

mysql>
```

9.3.6. Funciones de Encriptación

9.3.6.1. Funciones: AES_ENCRYPT(), AES_DECRYPT

Estas funciones permiten encriptar y desencriptar datos usando el algoritmo oficial AES (Advanced Encryption Standard), conocido previamente como Rijndael. Se usa una codificación con una clave de 128 bits de longitud.

Sintaxis

```
AES_ENCRYPT(string,key_string)
AES_DECRYPT(string,key_string)
```

Ejemplo 9 . 50

Este script ilustra el uso de las funciones de encriptación, la cadena a encriptar es **eureka** y la clave utilizada es **password**.

```
mysql> select @clave := aes_encrypt('eureka','password');
+-----+
| @clave := aes_encrypt('eureka','password') |
+-----+
| ♪\øù`¶ù} /L↔_½Ø< |
+-----+
1 row in set (0.00 sec)

mysql> select @clave ;
+-----+
| @clave           |
+-----+
| ♪\øù`¶ù} /L↔_½Ø< |
+-----+
1 row in set (0.00 sec)

mysql> select aes_decrypt(@clave,'password');
+-----+
| aes_decrypt(@clave,'password') |
+-----+
| eureka                     |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.6.2. Funciones: ENCODE(), DECODE()

Estas funciones también permiten encriptar y desencriptar cadenas.

Sintaxis

```
ENCODE(str,pass_str)
DECODE(crypt_str,pass_str)
```

Ejemplo 9 . 51

En el siguiente script la cadena a encriptar es **chiclayo** y la clave de encriptación es **password**.

```
mysql> select @clave := encode('chiclayo','password');
+-----+
| @clave := encode('chiclayo','password') |
+-----+
| →¶íârgDé |
+-----+
1 row in set (0.01 sec)
```

```
mysql> select @clave;
+-----+
| @clave   |
+-----+
| →ÍárgDé |
+-----+
1 row in set (0.00 sec)

mysql> select decode(@clave,'password');
+-----+
| decode(@clave,'password') |
+-----+
| chiclayo                   |
+-----+
1 row in set (0.00 sec)

mysql>
```

9.3.6.3. Funciones: DES_ENCRYPT(), DES_DECRYPT()

Estas funciones también permiten encriptar y desencriptar cadenas.

Sintaxis

```
DES_ENCRYPT(string_to_encrypt [, (key_number | key_string) ])
DES_DECRYPT(string_to_decrypt [, key_string])
```

La encriptación de la clave es usando el algoritmo Triple-DES. Estas funciones sólo están disponibles si MySQL fue configurado con soporte SSL. La clave de encriptado a usar se elige del modo siguiente:

Argumento	Descripción
Sólo un argumento	Se usa la primera clave del fichero de claves des.
key number	Se usa la clave dada (0-9) del fichero de claves des.
key string	Se usa la cadena para encriptar.



10

Consulta Avanzada de Datos

Cuando desarrollados consultas a una base de datos muchas veces queremos información de varias tablas ó algún tipo de resumen, el desarrollo de este tipo de consultas es el tema que desarrollaremos en este capítulo.

Temas a desarrollar

- 10.1. Totalizando Datos y Funciones de Grupo
- 10.2. Consultas Multitablas
- 10.3. Subconsultas

10.1. Totalizando Datos y Funciones de Grupo

10.1.1. Funciones de Grupo

10.1.1.1. Función: AVG()

Obtiene el promedio de una columna o expresión. Se puede aplicar la cláusula DISTINCT.

Sintaxis

AVG([DISTINCT] *expr*)

Ejemplo 10 . 1

El siguiente ejemplo calcula el importe promedio que existe en las cuentas de ahorro.

```
mysql> select avg(dec_cuensaldo) as promedio from cuenta;
+-----+
| promedio |
+-----+
| 6240.00 |
+-----+
1 row in set (0.02 sec)

mysql>
```

10.1.1.2. Función: COUNT()

Cuenta las filas de una consulta.

Sintaxis

```
COUNT(expr)
```

Ejemplo 10 . 2

Cantidad de empleados de la sucursal 002.

```
mysql> select count(*) as empleados
-> from asignado
-> where chr_sucucodigo = '002'
-> and dtt_asigfechabaja is null;
+-----+
| empleados |
+-----+
|          2 |
+-----+
1 row in set (0.00 sec)

mysql>
```

Cantidad de bajas que se han dado en la empresa.

Caso 01

La función `count()` no cuenta los valores nulos.

```
mysql> select count(dtt_asigfechabaja) from asignado;
+-----+
| count(dtt_asigfechabaja) |
+-----+
|                  1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

Caso 02

```
mysql> select count(*) from asignado where dtt_asigfechabaja is not null;
+-----+
| count(*) |
+-----+
|        1 |
+-----+
1 row in set (0.00 sec)

mysql>
```

10.1.1.3. Función: COUNT DISTINCT

Permiten contar valores distintos y diferentes de nulos.

Sintaxis

```
COUNT(DISTINCT expr,[expr...])
```

Ejemplo 10 . 3

Cuantas sucursales tienen por lo menos una cuenta.

```
mysql> select count(distinct chr_sucucodigo) from cuenta;
+-----+
| count(distinct chr_sucucodigo) |
+-----+
| 2 |
+-----+
1 row in set (0.02 sec)
```

mysql>

Cuantas combinaciones (chr_cuencodigo, chr_emplcodigo) diferentes existen en la tabla de movimientos.

```
mysql> select count(distinct chr_cuencodigo, chr_emplcodigo)
-> from movimiento;
+-----+
| count(distinct chr_cuencodigo, chr_emplcodigo) |
+-----+
| 5 |
+-----+
1 row in set (0.03 sec)
```

mysql>

10.1.1.4. Función: GROUP_CONCAT()

Retorna una cadena con los valores de grupo concatenadas.

Sintaxis Simple

GROUP_CONCAT(expr)

Sintaxis Completa

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
[ORDER BY {unsigned_integer | col_name | expr} [ASC | DESC] [,col_name ...]]
[SEPARATOR str_val])
```

Ejemplo 10 . 4

En el siguiente ejemplo se muestra los importes de los depósitos realizados en cada cuenta, para eso se aplica la función group_concat().

```
mysql> select chr_cuencodigo, group_concat(dec_moviimporte) as depositos
-> from movimiento
-> where chr_tipocodigo = '003'
-> group by chr_cuencodigo;
+-----+
| chr_cuencodigo | depositos
+-----+
| 00100001      | 3200.00,2000.00,1000.00
| 00100002      | 2200.00,1500.00
| 00200001      | 2000.00,2000.00,7000.00,2000.00,2000.00,1000.00,4000.00
| 00200002      | 4200.00
| 00200003      | 1500.00,3500.00
+-----+
5 rows in set (0.03 sec)
```

mysql>

10.1.1.5. Función: MAX()

Retorna el valor máximo de una columna ó expresión.

Sintaxis

MAX([DISTINCT] *expr*)

Ejemplo 10 . 5

La siguiente consulta retorna la fecha del último retiro.

```
mysql> select max(dtt_movifecha)
-> from movimiento
-> where chr_tipocodigo = '004';
+-----+
| max(dtt_movifecha) |
+-----+
| 2008-03-11         |
+-----+
1 row in set (0.00 sec)

mysql>
```

10.1.1.6. Función: MIN()

Retorna el mínimo valor de una columna ó expresión.

Sintaxis

MIN([DISTINCT] *expr*)

Ejemplo 10 . 6

La siguiente consulta retorna la fecha de apertura de la primera cuenta.

```
mysql> select min(dtt_movifecha)
-> from movimiento
-> where chr_tipocodigo = '001';
+-----+
| min(dtt_movifecha) |
+-----+
| 2008-01-05         |
+-----+
1 row in set (0.00 sec)

mysql>
```

10.1.1.7. Función: SUM()

Retorna la suma de los valores de una columna.

Sintaxis

SUM([DISTINCT] *expr*)

Ejemplo 10 . 7

La siguiente consulta retorna la suma de todos los depósitos.

```
mysql> select sum(dec_moviimporte) as importe
-> from movimiento
-> where chr_tipocodigo = '003';
+-----+
| importe   |
+-----+
```

```
+-----+
| 39100.00 |
+-----+
1 row in set (0.00 sec)

mysql>
```

10.1.2. GROUP BY

Se utiliza para agrupar data en base a una ó más columnas, para aplicar funciones de grupo.

Sintaxis

```
GROUP BY {col_name | expr | position} [ASC | DESC], . . . [WITH ROLLUP]]
```

Ejemplo 10 . 8

La siguiente consulta muestra la cantidad de empleados por sucursal.

```
mysql> select chr_sucucodigo, count(*) as empleados
-> from asignado
-> where dtt_asigfechabaja is null
-> group by chr_sucucodigo;
+-----+-----+
| chr_sucucodigo | empleados |
+-----+-----+
| 001            |      2 |
| 002            |      2 |
| 003            |      1 |
| 004            |      1 |
| 005            |      2 |
| 006            |      2 |
+-----+-----+
6 rows in set (0.00 sec)
```

```
mysql>
```

Ejemplo 10 . 9

La siguiente consulta muestra el importe de por cada tipo de movimiento, para cada cuenta.

```
mysql> select
->     chr_cuencodigo as sucursal,
->     chr_tipocodigo as tipo_mov,
->     sum(dec_moviimporte) as importe
->   from movimiento
->  group by chr_cuencodigo, chr_tipocodigo;
+-----+-----+-----+
| sucursal | tipo_mov | importe |
+-----+-----+-----+
| 00100001 | 001     | 2800.00 |
| 00100001 | 003     | 6200.00 |
| 00100001 | 004     | 2100.00 |
| 00100002 | 001     | 1800.00 |
| 00100002 | 003     | 3700.00 |
| 00100002 | 004     | 1000.00 |
| 00200001 | 001     | 5000.00 |
| 00200001 | 003     | 20000.00 |
| 00200001 | 004     | 18000.00 |
| 00200002 | 001     | 3800.00 |
| 00200002 | 003     | 4200.00 |
| 00200002 | 004     | 1200.00 |
| 00200003 | 001     | 2500.00 |
| 00200003 | 003     | 5000.00 |
| 00200003 | 004     | 1500.00 |
```

```
+-----+-----+-----+
15 rows in set (0.00 sec)
```

```
mysql>
```

Ejemplo 10 . 10

La siguiente consulta muestra la cantidad de movimientos registrados por cada empleado.

```
mysql> select chr_emplcodigo, count(*) as movimientos
-> from movimiento
-> group by 1;
+-----+-----+
| chr_emplcodigo | movimientos |
+-----+-----+
| 0001           |        24 |
| 0004           |        11 |
+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql>
```

Ejemplo 10 . 11

La siguiente consulta muestra el importe por cada tipo de movimiento.

```
mysql> select
->     chr_tipocodigo as tipo_mov,
->     sum(dec_moviimporte) as importe
-> from movimiento
-> group by chr_tipocodigo;
+-----+
| tipo_mov | importe |
+-----+
| 001      | 15900.00 |
| 003      | 39100.00 |
| 004      | 23800.00 |
+-----+
3 rows in set (0.00 sec)
```

```
mysql>
```

Ejemplo 10 . 12

La siguiente consulta muestra el depósito realizado en cada mes del año 2008.

```
mysql> select
->     month(dtt_movifecha) as mes,
->     sum(dec_moviimporte) as deposito
-> from movimiento
-> where year(dtt_movifecha) = 2008
-> and chr_tipocodigo = '003'
-> group by 1;
+-----+
| mes   | deposito |
+-----+
|    1  | 24900.00 |
|    2  | 11700.00 |
|    3  | 2500.00  |
+-----+
3 rows in set (0.00 sec)
```

```
mysql>
```

10.1.3. HAVING

Permite limitar mediante una condición de grupo el resultado obtenido después de aplicar GROUP BY, tal como se aprecia en la Figura 10.1.

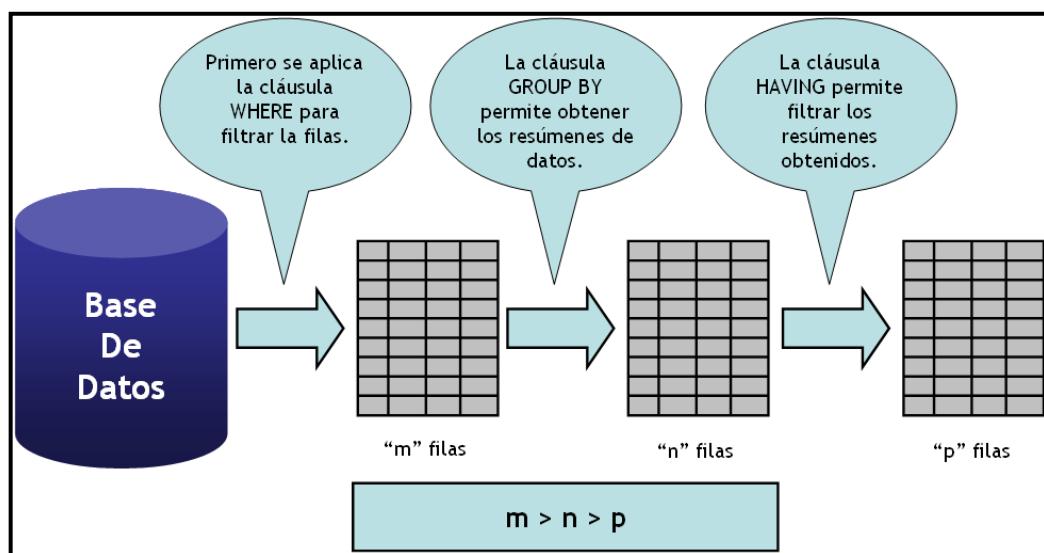


Figura 10 . 1 Esquema de funcionamiento de la sentencia SELECT.

Sintaxis

HAVING Group_condition

Ejemplo 10 . 13

La siguiente consulta retorna los meses del 2008 en los que los depósitos fueron superiores a 20000.

```
mysql> select
->      month(dtt_movifecha) as mes,
->      sum(dec_moviimporte) as importe
->  from movimiento
-> where year(dtt_movifecha) = 2008
-> and chr_tipocodigo = '003'
-> group by 1
-> having sum(dec_moviimporte) > 20000;
+-----+
| mes | importe |
+-----+
|    1 | 24900.00 |
+-----+
1 row in set (0.02 sec)
```

mysql>

Ejemplo 10 . 14

La siguiente consulta permite averiguar que sucursales tienen asignado solamente un empleado.

```
mysql> select
->      chr_sucucodigo as sucursal,
->      count(*) as empleados
->  from asignado
-> where dtt_asigfechabaja is null
-> group by 1
```

```
-> having count(*) = 1;
+-----+-----+
| sucursal | empleados |
+-----+-----+
| 003      |          1 |
| 004      |          1 |
+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

10.2. Consultas Multitablas

10.2.1. ¿Qué es un Join?

Un Join es usado para consultar datos desde más de una tabla. Las filas se combinan (joined) relacionando valores comunes, típicamente valores de primary key y foreign key.

Para hacer referencia a la columna de una tabla se debe usar uno de los siguientes formatos:

NombreTabla.NombreColumna

Alias.NombreColumna

10.2.2. Consultas Simples

Formato

```
SELECT tabla.columna, tabla.columna, . . .
FROM tabla1, tabla2
WHERE (condición_de_combinación)
. . .
```

Ejemplo 10 . 15

La siguiente consulta muestra un listado de los cuentas con sus respectivos saldos y la sucursal a la que pertenece.

```
mysql> select
->     sucursal.vch_sucunombre as sucursal,
->     cuenta.chr_cuencodigo as cuenta,
->     cuenta.dec_cuensaldo as saldo
-> from sucursal, cuenta
-> where (sucursal.chr_sucucodigo = cuenta.chr_sucucodigo);
+-----+-----+-----+
| sucursal | cuenta | saldo   |
+-----+-----+-----+
| Sipan    | 00100001 | 6900.00 |
| Sipan    | 00100002 | 4500.00 |
| Chan Chan | 00200001 | 7000.00 |
| Chan Chan | 00200002 | 6800.00 |
| Chan Chan | 00200003 | 6000.00 |
+-----+-----+-----+
5 rows in set (0.03 sec)
```

```
mysql>
```

10.2.3. Consultas Complejas

Formato

```
SELECT tabla.columna, tabla.columna, . . .
FROM tabla1, tabla2
WHERE (condición_de_combinación)
AND (condición_de_filtro)
. . .
```

Ejemplo 10 . 16

Se necesita una consulta para mostrar las cuentas de la sucursal 002, cual es su saldo y quién es el empleado que la creo.

```
mysql> select
->     cuenta.chr_cuencodigo as cuenta,
->     cuenta.dec_cuensaldo as saldo,
->     concat(empleado.vch_emplnombre,space(1),
->             empleado.vch_emplpaterno) as "Creada por"
->   from empleado, cuenta
->  where (empleado.chr_emplcodigo = cuenta.chr_emplcreacuenta)
->  and (cuenta.chr_sucucodigo = '002');
+-----+-----+
| cuenta | saldo | Creada por |
+-----+-----+
| 00200001 | 7000.00 | Carlos Alberto Romero |
| 00200002 | 6800.00 | Carlos Alberto Romero |
| 00200003 | 6000.00 | Carlos Alberto Romero |
+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

10.2.3.1. Uso de Alias

Los alias simplifican la referencia a las columnas de las tablas que se utilizan en una consulta.

Definición

```
FROM Tabla [AS] Alias1, . . .
```

Referencia a Columnas

```
Alias.NombreColumna
```

Ejemplo 10 . 17

La siguiente consulta muestra las cuentas, con su respectivo saldo y el cliente al que pertenece.

```
mysql> select
->     cu.chr_cuencodigo as cuenta,
->     cu.dec_cuensaldo as saldo,
->     concat(cl.vch_clienombre,space(1),cl.vch_cliepaterno) as cliente
->   from cliente as cl, cuenta as cu
->  where (cl.chr_cliecodigo = cu.chr_cliecodigo);
+-----+-----+
| cuenta | saldo | cliente           |
+-----+-----+
```

```
| 00100001 | 6900.00 | ALAN ALBERTO ARANDA |
| 00100002 | 4500.00 | ALAN ALBERTO ARANDA |
| 00200001 | 7000.00 | ROSA LIZET FLORES |
| 00200002 | 6800.00 | ERIC GUSTAVO CORONEL |
| 00200003 | 6000.00 | EDGAR RAFAEL CHAVEZ |
+-----+-----+-----+
5 rows in set (0.01 sec)
```

mysql>

10.2.4. Usando Sintaxis ANSI

Sintaxis

tabla1 NATURAL [INNER] JOIN tabla2

tabla1 [INNER] JOIN tabla2 USING (columnas)

tabla1 [INNER] JOIN tabla2 ON <condición>

tabla1 STRAIGHT_JOIN tabla2

tabla1 STRAIGHT_JOIN tabla2 ON <condición>

10.2.4.1. NATURAL JOIN

Este tipo de join esta basada en todas las columnas con igual nombre entre ambas tablas.

Ejemplo 10 . 18

En la siguiente consulta se ilustra el uso de **natural join**, note que las tablas **moneda** y **cuenta** tienen la columna **chr_cuencodigo** en común, por lo tanto, esta es la columna a través de la cual se combinan.

```
mysql> select
    ->     c.chr_cuencodigo as cuenta,
    ->     c.dec_cuensaldo as saldo,
    ->     m.vch_moned descripcion as moneda
    -> from moneda as m natural join cuenta as c;
+-----+-----+-----+
| cuenta | saldo | moneda |
+-----+-----+-----+
| 00100001 | 6900.00 | Soles |
| 00200001 | 7000.00 | Soles |
| 00200002 | 6800.00 | Soles |
| 00100002 | 4500.00 | Dolares |
| 00200003 | 6000.00 | Dolares |
+-----+-----+-----+
5 rows in set (0.03 sec)
```

mysql>

10.2.4.2. JOIN . . . USING

Permite indicar las columnas a combinar entre dos tablas.

Ejemplo 10 . 19

La siguiente consulta muestra la forma como se debe utilizar **join – using** para consultar dos tablas, note que la columna que combina ambas tablas es **chr_monecodigo**.

```
mysql> select
->      m.vch_moned descripcion as moneda,
->      c.dec_costo importe
-> from moneda as m
-> join costomovimiento as c using (chr_monecodigo);
+-----+
| moneda | importe |
+-----+
| Soles  |    2.00 |
| Dolares |    0.60 |
+-----+
2 rows in set (0.00 sec)

mysql>
```

Ejemplo 10 . 20

La siguiente consulta muestra la combinación de tres tablas usando **join . . . using**.

```
mysql> select
->      s.vch_sucunombre as sucursal,
->      concat(e.vch_emplnombre,space(1),e.vch_emplpaterno) as empleado
-> from sucursal as s
-> join asignado as a using (chr_sucucodigo)
-> join empleado as e using (chr_emplcodigo);
+-----+
| sucursal | empleado          |
+-----+
| Sipan    | Angelica Ramos     |
| Chan Chan | Carlos Alberto Romero |
| Los Olivos | Lidia Castro        |
| Pardo    | Claudia Reyes       |
| Misti    | Ricardo Cruz        |
| Machupicchu | Claudia Ruiz        |
| Pardo    | Edith Diaz          |
| Misti    | Claudia Rocio Sarmiento |
| Sipan    | Pedro Hugo Carrasco   |
| Chan Chan | Luis Alberto Pachas   |
| Machupicchu | Hugo Valentin Tello   |
+-----+
11 rows in set (0.00 sec)

mysql>
```

10.2.4.3. JOIN ... ON

La condición que permite combinar ambas tablas se debe especificar en la cláusula ON.

Ejemplo 10 . 21

El siguiente script ilustra el uso de **join . . . on** para consultar datos de dos tablas.

```
mysql> select
->     c.chr_cuencodigo,
->     m.vch_monedescripcion,
->     c.dec_cuensaldo
-> from cuenta as c
-> join moneda as m on c.chr_monecodigo = m.chr_monecodigo;
+-----+-----+-----+
| chr_cuencodigo | vch_monedescripcion | dec_cuensaldo |
+-----+-----+-----+
| 00100001      | Soles                 | 6900.00   |
| 00100002      | Dolares               | 4500.00   |
| 00200001      | Soles                 | 7000.00   |
| 00200002      | Soles                 | 6800.00   |
| 00200003      | Dolares               | 6000.00   |
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

Ejemplo 10 . 22

En la siguiente consulta se puede apreciar la consulta de tres tablas usando **join . . . on**.

```
mysql> select
->     c.chr_cuencodigo,
->     c.dec_cuensaldo,
->     t.vch_tipodescripcion,
->     m.dec_moviimporte
-> from cuenta as c
-> join movimiento as m on c.chr_cuencodigo = m.chr_cuencodigo
-> join tipomovimiento as t on m.chr_tipocodigo = t.chr_tipocodigo
-> where c.chr_sucucodigo = '001';
+-----+-----+-----+
| chr_cuencodigo | dec_cuensaldo | vch_tipodescripcion | dec_moviimporte |
+-----+-----+-----+
| 00100001      | 6900.00    | Apertura de Cuenta | 2800.00   |
| 00100001      | 6900.00    | Deposito           | 3200.00   |
| 00100001      | 6900.00    | Retiro              | 800.00    |
| 00100001      | 6900.00    | Deposito           | 2000.00   |
| 00100001      | 6900.00    | Retiro              | 500.00    |
| 00100001      | 6900.00    | Retiro              | 800.00    |
| 00100001      | 6900.00    | Deposito           | 1000.00   |
| 00100002      | 4500.00    | Apertura de Cuenta | 1800.00   |
| 00100002      | 4500.00    | Retiro              | 1000.00   |
| 00100002      | 4500.00    | Deposito           | 2200.00   |
| 00100002      | 4500.00    | Deposito           | 1500.00   |
+-----+-----+-----+
11 rows in set (0.00 sec)

mysql>
```

10.2.4.4. STRAIGHT_JOIN

MySQL no se preocupa del orden en que aparecen las tablas en un JOIN. EL optimizador probará todas las combinaciones y escogerá la que crea más eficiente. Esto puede provocar que algunas veces la elección no sea la más óptima. Para comprobarlo siempre podemos usar EXPLAIN. Si el resultado nos confirma que existe un orden mejor, se puede usar STRAIGHT_JOIN.

Ejemplo 10 . 23

En la siguiente consulta se ilustra el uso de STRAIGHT_JOIN para hacer consultas a mas de dos tablas.

```
mysql> select
->      s.vch_sucunombre as sucursal,
->      sum(m.dec_moviimporte) as deposito
->  from sucursal as s
->  straight_join cuenta as c on s.chr_sucucodigo = c.chr_suencodigo
->  straight_join movimiento as m on c.chr_cuencodigo = m.chr_cuencodigo
->  where m.chr_tipocodigo = '003'
->  group by s.vch_sucunombre;
+-----+-----+
| sucursal | deposito |
+-----+-----+
| Chan Chan | 29200.00 |
| Sipan     | 9900.00 |
+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

10.2.5. Producto Cartesiano

Si dos tablas en una consulta no tienen ninguna condición de combinación, entonces se retorna su producto cartesiano. MySQL combina cada fila de una tabla con cada fila de la otra tabla. Un producto cartesiano genera muchas filas y es siempre raramente útil. Por ejemplo, el producto cartesiano de dos tablas, cada una con 100 filas, tiene 10.000 filas.

Sintaxis

```
FROM Tabla1, Tabla2, ...
```

Sintaxis ANSI

```
FROM Tabla1 CROSS JOIN Tabla2
```

Ejemplo 10 . 24

Las siguientes consultas muestran como obtener el producto cartesiano de dos tablas.

```
mysql> select s.chr_sucucodigo, c.chr_suencodigo
->  from sucursal as s, cuenta as c
->  order by 1;

mysql> select s.chr_sucucodigo, c.chr_suencodigo
->  from sucursal as s cross join cuenta as c
->  order by 1;
```

El resultado de ambas consultas es el mismo.

```
+-----+-----+
| chr_sucucodigo | chr_suencodigo |
```

```
+-----+-----+
| 001   | 00100001   |
| 001   | 00100002   |
| 001   | 00200001   |
| 001   | 00200003   |
| 001   | 00200002   |
| 002   | 00200002   |
| 002   | 00100001   |
.
.
.
| 007   | 00200002   |
| 007   | 00100001   |
| 007   | 00100002   |
| 007   | 00200001   |
+-----+
35 rows in set (0.00 sec)
```

10.2.6. Joins Externos

Una combinación externa amplía el resultado de una combinación simple. Una combinación externa devuelve todas las filas que satisfagan la condición de combinación y también vuelve todas o parte de las filas de una tabla para la cual ninguna filas de la otra satisfagan la condición de combinación.

10.2.6.1. LEFT JOIN

Sintaxis

```
Tabla1 NATURAL LEFT [OUTER] JOIN Tabla2
```

```
Tabla1 LEFT [OUTER] JOIN Tabla2 ON Condición
```

Este tipo de combinación dará como resultado las filas que se combinan, pero incluirá las filas de la tabla que se encuentra a la izquierda del operador que no se combinan, tal como se puede apreciar en el Ejemplo 10.25.

Ejemplo 10 . 25

La siguiente consulta muestra la combinación de la tabla sucursal con la tabla cuenta, como se puede apreciar en el resultado aparecen todas las sucursales, incluidas los que no tienen cuentas.

```
mysql> select s.chr_sucucodigo, c.chr_cuencodigo, c.dec_cuensaldo
-> from sucursal as s
-> left join cuenta as c on s.chr_sucucodigo = c.chr_sucucodigo;
+-----+-----+-----+
| chr_sucucodigo | chr_cuencodigo | dec_cuensaldo |
+-----+-----+-----+
| 001           | 00100001      |    6900.00 |
| 001           | 00100002      |    4500.00 |
| 002           | 00200001      |    7000.00 |
| 002           | 00200002      |    6800.00 |
| 002           | 00200003      |    6000.00 |
| 003           | NULL          |      NULL  |
| 004           | NULL          |      NULL  |
| 005           | NULL          |      NULL  |
| 006           | NULL          |      NULL  |
| 007           | NULL          |      NULL  |
+-----+-----+-----+
10 rows in set (0.00 sec)

mysql>
```

10.2.6.2. RIGHT JOIN

Sintaxis

Tabla1 NATURAL RIGHT [OUTER] JOIN Tabla2

Tabla1 RIGHT [OUTER] JOIN Tabla2 ON Condición

Este tipo de combinación dará como resultado las filas que se combinan, pero incluirá las filas de la tabla que se encuentra a la derecha del operador que no se combinan, tal como se puede apreciar en el Ejemplo 10.26.

Ejemplo 10 . 26

La siguiente consulta muestra la combinación de la tabla cuenta con cliente, puede usted notar que aparecen todos los clientes, incluido los que no tienen cuenta alguna.

```
mysql> select cu.chr_cuencodigo, cu.dec_cuensaldo, cl.vch_clienombre
-> from cuenta as cu
-> right join cliente as cl on cu.chr_cliecodigo = cl.chr_cliecodigo
-> order by 1 desc, 3;
+-----+-----+-----+
| chr_cuencodigo | dec_cuensaldo | vch_clienombre |
+-----+-----+-----+
| 00200003 | 6000.00 | EDGAR RAFAEL |
| 00200002 | 6800.00 | ERIC GUSTAVO |
| 00200001 | 7000.00 | ROSA LIZET |
| 00100002 | 4500.00 | ALAN ALBERTO |
| 00100001 | 6900.00 | ALAN ALBERTO |
| NULL | NULL | CARLOS ALBERTO |
| NULL | NULL | CRISTIAN RAFAEL |
| NULL | NULL | DEYSI LIDIA |
| NULL | NULL | ENRIQUE MANUEL |
| NULL | NULL | FELIX NINO |
| NULL | NULL | FERNANDO MOISES |
| NULL | NULL | GABRIEL ALEJANDRO |
| NULL | NULL | JORGE LUIS |
| NULL | NULL | JUAN CARLOS |
| NULL | NULL | JUAN DIEGO |
| NULL | NULL | LIDIA ROXANA |
| NULL | NULL | PEDRO HUGO |
| NULL | NULL | RICARDO |
| NULL | NULL | ROSARIO ESMERALDA |
| NULL | NULL | TANIA LORENA |
| NULL | NULL | YESABETH |
+-----+-----+-----+
21 rows in set (0.00 sec)
```

mysql>

10.2.7. Consultas Autoreferenciadas

En este tipo de consultas se hace referencia a la misma tabla dos veces en la misma consulta, esto se consigue asignando alias diferentes.

Este tipo de consultas se construyen sobre tablas que también están autoreferenciadas.

Ejemplo 10 . 27

Para ilustrar este caso crearemos la tabla trabajador:

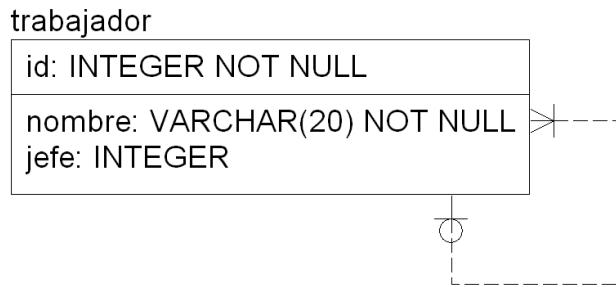


Figura 10 . 2 Representación de la tabla trabajador autoreferenciada.

La columna **jefe** es una clave foránea de la tabla **trabajador** que hace referencia a la misma tabla **trabajador**.

```
mysql> create table trabajador(
->     id int not null auto_increment,
->     nombre varchar(20) not null,
->     jefe int null,
->     primary key(id)
-> );
Query OK, 0 rows affected (0.08 sec)
```

```
mysql>
```

Cargaremos algunos datos:

```
mysql> insert into trabajador(nombre,jefe) values('Gustavo',null);
mysql> insert into trabajador(nombre,jefe) values('Ricardo',1);
mysql> insert into trabajador(nombre,jefe) values('Sergio',1);
mysql> insert into trabajador(nombre,jefe) values('Hugo',1);
mysql> insert into trabajador(nombre,jefe) values('Claudia',2);
mysql> insert into trabajador(nombre,jefe) values('Julio',2);
mysql> insert into trabajador(nombre,jefe) values('Cesar',2);
mysql> insert into trabajador(nombre,jefe) values('Adriana',3);
mysql> insert into trabajador(nombre,jefe) values('Miguel',3);
mysql> insert into trabajador(nombre,jefe) values('Rosario',3);
mysql> insert into trabajador(nombre,jefe) values('Arturo',4);
mysql> insert into trabajador(nombre,jefe) values('Angel',4);
mysql> insert into trabajador(nombre,jefe) values('Andrea',4);
```

Ahora se necesita consultar los trabajadores con sus respectivos nombre de sus jefes.

```
mysql> select t.id, t.nombre, j.nombre as jefe
-> from trabajador as t
-> join trabajador as j on t.jefe = j.id;
+----+-----+-----+
| id | nombre | jefe   |
+----+-----+-----+
|  2 | Ricardo | Gustavo |
|  3 | Sergio  | Gustavo |
|  4 | Hugo    | Gustavo |
|  5 | Claudia | Ricardo |
|  6 | Julio   | Ricardo |
|  7 | Cesar   | Ricardo |
|  8 | Adriana | Sergio  |
|  9 | Miguel  | Sergio  |
| 10 | Rosario | Sergio  |
| 11 | Arturo  | Hugo   |
| 12 | Angel   | Hugo   |
| 13 | Andrea  | Hugo   |
+----+-----+-----+
12 rows in set (0.00 sec)
```

```
mysql>
```

10.2.8. Unión de Resultados

Sintaxis

```
SELECT . . .
```

```
UNION [ALL | DISTINCT]
```

```
SELECT . . .
```

```
UNION [ALL | DISTINCT]
```

```
SELECT . . .
```

```
...
```

Este operador permite unir en un único resultado el resultado de dos ó más sentencias SELECT.

Los resultados a unir deben tener igual número de columnas y además deben ser de tipos compatibles.

Figura 10 . 3

En la siguiente consulta se puede apreciar el funcionamiento del operador UNION para mostrar los tipos de ocupación (cargo) y departamentos de la empresa en un único resultado.

```
mysql> select 'Cargo' as Tipo, nombre
-> from cargo
-> union
-> select 'departamento', nombre
-> from departamento;
+-----+-----+
| Tipo      | nombre        |
+-----+-----+
| Cargo     | Gerente General |
| Cargo     | Gerente       |
| Cargo     | Empleado      |
| Cargo     | Analista      |
| Cargo     | Vendedor      |
| Cargo     | Oficinista    |
| Cargo     | Programador   |
| Cargo     | Investigador |
| Cargo     | Digitador     |
| departamento | Gerencia      |
| departamento | Contabilidad |
| departamento | Investigacion |
| departamento | Ventas       |
| departamento | Operaciones  |
| departamento | Sistemas     |
| departamento | Recursos Humanos |
| departamento | Finanzas     |
+-----+-----+
17 rows in set (0.05 sec)
```

```
mysql>
```

10.3. Subconsultas

Una subconsulta es una consulta dentro de otra consulta. Una subconsulta responde a consultas que tienen múltiples partes; la subconsulta responde una parte de toda la consulta.

Cuando se anidan muchas subconsultas, las consultas mas internas son las que se evalúan primero. Las subconsultas se pueden utilizar en todas las instrucciones DML.

10.3.1. Subconsultas Como Tabla Derivada

En este caso las subconsulta se ubican en la cláusula FROM de la consulta padre.

Formato

```
SELECT . . .
FROM ( Subconsulta ) As Alias . . .
WHERE . . .
. . .
```

Ejemplo 10 . 28

La siguiente consulta muestra la cantidad de cuentas que hay por sucursal.

```
mysql> select chr_sucucodigo, count(*) as cuentas
      -> from cuenta
      -> group by chr_sucucodigo;
+-----+-----+
| chr_sucucodigo | cuentas |
+-----+-----+
| 001           |      2 |
| 002           |      3 |
+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql>
```

Ahora queremos agregarle una columna mas con el nombre de la sucursal, para lo cual utilizaremos la consulta anterior como tabla derivada y la combinaremos con la tabla de sucursales, tal como se aprecia a continuación:

```
mysql> select t.chr_sucucodigo, s.vch_sucunombre, t.cuentas
      -> from ( select chr_sucucodigo, count(*) as cuentas
      ->           from cuenta
      ->           group by chr_sucucodigo ) as t
      -> natural join sucursal as s;
+-----+-----+-----+
| chr_sucucodigo | vch_sucunombre | cuentas |
+-----+-----+-----+
| 001           | Sipan          |      2 |
| 002           | Chan Chan       |      3 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql>
```

10.3.2. Subconsulta Como Expresión

En este caso la subconsulta retorna un único valor y se debe tratar como tal.

Ejemplo 10 . 29

La siguiente consulta muestra las cuentas que tienen un saldo superior al saldo promedio.

```
mysql> select chr_cuencodigo, dec_cuensaldo
-> from cuenta
-> where dec_cuensaldo > (select avg(dec_cuensaldo) from cuenta);
+-----+-----+
| chr_cuencodigo | dec_cuensaldo |
+-----+-----+
| 00100001      |      6900.00 |
| 00200001      |      7000.00 |
| 00200002      |      6800.00 |
+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

Ejemplo 10 . 30

La siguiente consulta muestra el saldo de cada cuenta, el saldo promedio, y la diferencia entre estos dos.

```
mysql> select
->     chr_cuencodigo,
->     dec_cuensaldo,
->     (select avg(dec_cuensaldo) from cuenta) as promedio,
->     dec_cuensaldo - (select avg(dec_cuensaldo) from cuenta) as
diferencia
-> from cuenta;
+-----+-----+-----+-----+
| chr_cuencodigo | dec_cuensaldo | promedio | diferencia |
+-----+-----+-----+-----+
| 00100001      |      6900.00 | 6240.000000 |    660.000000 |
| 00100002      |      4500.00 | 6240.000000 |   -1740.000000 |
| 00200001      |      7000.00 | 6240.000000 |    760.000000 |
| 00200002      |      6800.00 | 6240.000000 |    560.000000 |
| 00200003      |      6000.00 | 6240.000000 |   -240.000000 |
+-----+-----+-----+-----+
5 rows in set (0.01 sec)
```

```
mysql>
```

10.3.3. Subconsulta para Correlacionar Datos

En este caso la subconsulta utiliza columnas de la consulta externa ó padre, de tal manera que la subconsulta es evaluada por cada fila de la consulta externa.

Este tipo de subconsulta consume muchos recursos, sobre todo cuando se tiene que analizar gran volumen de información.

Ejemplo 10 . 31

La siguiente consulta realiza un listado de los sucursales y la cantidad de cuentas en cada una de ellas, puede usted notar que en la subconsulta se utiliza una columna de la tabla sucursal (s.chr_sucucodigo).

```
mysql> select
->     s.chr_sucucodigo, s.vch_sucunombre,
```

```
->      ( select count(*) from cuenta as c
->          where c.chr_sucucodigo = s.chr_sucucodigo ) as cuentas
-> from sucursal as s;
+-----+-----+-----+
| chr_sucucodigo | vch_sucunombre | cuentas |
+-----+-----+-----+
| 001            | Sipan           |      2 |
| 002            | Chan Chan       |      3 |
| 003            | Los Olivos      |      0 |
| 004            | Pardo           |      0 |
| 005            | Misti           |      0 |
| 006            | Machupicchu     |      0 |
| 007            | Grau            |      0 |
+-----+-----+-----+
7 rows in set (0.03 sec)
```

mysql>

10.3.4. Operador: [NOT] EXISTS

Sintaxis

[NOT] EXISTS (subconsulta)

Este operador se utiliza para verificar si una subconsulta tiene ó no resultados, específicamente si tiene filas ó no, no interesa que columnas

Ejemplo 10 . 32

La siguiente consulta retorna los empleados que por lo menos han registrado un movimiento.

```
mysql> select e.chr_emplcodigo, e.vch_emplnombre, e.vch_emplpaterno
-> from empleado as e
-> where exists (
->     select 1 from movimiento as m
->     where e.chr_emplcodigo = m.chr_emplcodigo );
+-----+-----+-----+
| chr_emplcodigo | vch_emplnombre | vch_emplpaterno |
+-----+-----+-----+
| 0001            | Carlos Alberto | Romero           |
| 0004            | Angelica       | Ramos            |
+-----+-----+-----+
2 rows in set (0.02 sec)
```

mysql>

10.3.5. Operador: [NOT] IN

Sintaxis

Columna [NOT] IN (Subconsulta)

En este caso la subconsulta debe retornan una sola columna, pero pueden ser muchas filas.

Se utiliza para verificar si el valor de una columna esta en el resultado de la subconsulta.

Ejemplo 10 . 33

La siguiente consulta muestra los empleados que han registrado por lo menos un movimiento, hay que tener en cuenta que en este caso se está utilizando el operador IN.

```
mysql> select chr_emplcodigo, vch_emplnombre, vch_emplpaterno
```

```

-> from empleado
-> where chr_emplcodigo in
->      ( select distinct chr_emplcodigo from movimiento );
+-----+-----+
| chr_emplcodigo | vch_emplnombre | vch_emplpaterno |
+-----+-----+
| 0001           | Carlos Alberto | Romero          |
| 0004           | Angelica       | Ramos            |
+-----+-----+
2 rows in set (0.00 sec)

mysql>

```

10.3.6. Consultas de Referencias Cruzadas

Una consulta de referencias cruzadas es aquella que nos permite resumir datos en filas y en columnas, estilo tabla.

Ejemplo 10 . 34

Para el siguiente ejemplo utilizaremos la tabla `venta`. A continuación tenemos el script que crea la tabla `venta` y carga datos de prueba.

```

-- Tabla Ventas

create table ventas(
    id int not null auto_increment,
    tienda char(2) not null,
    fecha date not null,
    importe numeric(10,2) not null,
    primary key( id )
);

-- Enero

insert into ventas(tienda,fecha,importe) values('01','2008-01-05',400.00);
insert into ventas(tienda,fecha,importe) values('01','2008-01-11',300.00);
insert into ventas(tienda,fecha,importe) values('01','2008-01-14',800.00);
insert into ventas(tienda,fecha,importe) values('01','2008-01-20',100.00);
insert into ventas(tienda,fecha,importe) values('01','2008-01-25',700.00);

insert into ventas(tienda,fecha,importe) values('02','2008-01-05',200.00);
insert into ventas(tienda,fecha,importe) values('02','2008-01-11',800.00);
insert into ventas(tienda,fecha,importe) values('02','2008-01-14',100.00);
insert into ventas(tienda,fecha,importe) values('02','2008-01-20',300.00);
insert into ventas(tienda,fecha,importe) values('02','2008-01-25',600.00);

insert into ventas(tienda,fecha,importe) values('03','2008-01-05',900.00);
insert into ventas(tienda,fecha,importe) values('03','2008-01-11',200.00);
insert into ventas(tienda,fecha,importe) values('03','2008-01-14',600.00);
insert into ventas(tienda,fecha,importe) values('03','2008-01-20',400.00);
insert into ventas(tienda,fecha,importe) values('03','2008-01-25',200.00);

-- Febrero

insert into ventas(tienda,fecha,importe) values('01','2008-02-05',300.00);
insert into ventas(tienda,fecha,importe) values('01','2008-02-11',500.00);
insert into ventas(tienda,fecha,importe) values('01','2008-02-14',100.00);
insert into ventas(tienda,fecha,importe) values('01','2008-02-20',800.00);
insert into ventas(tienda,fecha,importe) values('01','2008-02-25',600.00);

insert into ventas(tienda,fecha,importe) values('02','2008-02-05',900.00);
insert into ventas(tienda,fecha,importe) values('02','2008-02-11',300.00);
insert into ventas(tienda,fecha,importe) values('02','2008-02-14',400.00);
insert into ventas(tienda,fecha,importe) values('02','2008-02-20',100.00);
insert into ventas(tienda,fecha,importe) values('02','2008-02-25',400.00);

```

```
insert into ventas(tienda,fecha,importe) values('03','2008-02-05',330.00);
insert into ventas(tienda,fecha,importe) values('03','2008-02-11',150.00);
insert into ventas(tienda,fecha,importe) values('03','2008-02-14',450.00);
insert into ventas(tienda,fecha,importe) values('03','2008-02-20',720.00);
insert into ventas(tienda,fecha,importe) values('03','2008-02-25',440.00);

-- Marzo

insert into ventas(tienda,fecha,importe) values('01','2008-03-05',530.00);
insert into ventas(tienda,fecha,importe) values('01','2008-03-11',650.00);
insert into ventas(tienda,fecha,importe) values('01','2008-03-14',910.00);
insert into ventas(tienda,fecha,importe) values('01','2008-03-20',180.00);
insert into ventas(tienda,fecha,importe) values('01','2008-03-25',260.00);

insert into ventas(tienda,fecha,importe) values('02','2008-03-05',910.00);
insert into ventas(tienda,fecha,importe) values('02','2008-03-11',330.00);
insert into ventas(tienda,fecha,importe) values('02','2008-03-14',420.00);
insert into ventas(tienda,fecha,importe) values('02','2008-03-20',160.00);
insert into ventas(tienda,fecha,importe) values('02','2008-03-25',480.00);

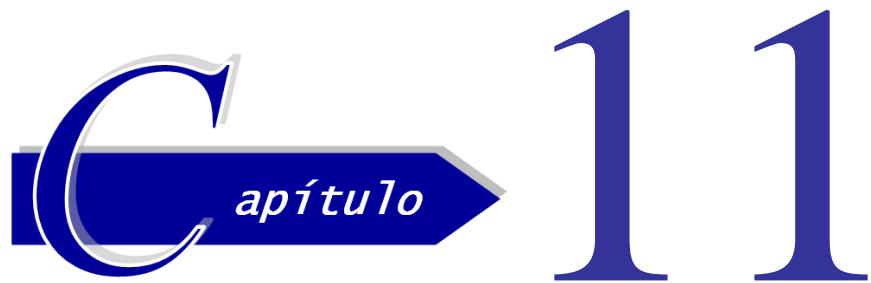
insert into ventas(tienda,fecha,importe) values('03','2008-03-05',230.00);
insert into ventas(tienda,fecha,importe) values('03','2008-03-11',350.00);
insert into ventas(tienda,fecha,importe) values('03','2008-03-14',650.00);
insert into ventas(tienda,fecha,importe) values('03','2008-03-20',120.00);
insert into ventas(tienda,fecha,importe) values('03','2008-03-25',840.00);
```

Se necesita una consulta que muestre las ventas totales por mes en cada una de las tiendas.
Se debe considerar el primer trimestre del 2008.

La consulta para este requerimiento es:

```
mysql> select
->     tienda,
->     sum( case month(fecha) when 1 then importe else 0 end ) as Enero,
->     sum( case month(fecha) when 2 then importe else 0 end ) as Febrero,
->     sum( case month(fecha) when 3 then importe else 0 end ) as Marzo,
->     sum( importe ) as total
->   from ventas
->   where year(fecha) = 2008
->   and month(fecha) in (1,2,3)
->   group by tienda;
+-----+-----+-----+-----+
| tienda | Enero | Febrero | Marzo | total |
+-----+-----+-----+-----+
| 01     | 2300.00 | 2300.00 | 2530.00 | 7130.00 |
| 02     | 2000.00 | 2100.00 | 2300.00 | 6400.00 |
| 03     | 2300.00 | 2090.00 | 2190.00 | 6580.00 |
+-----+-----+-----+-----+
3 rows in set (0.02 sec)

mysql>
```



Trabajando con Datos

Este es un capítulo muy importante, aquí veremos como insertar datos, actualizarlos y eliminarlos si es necesario, y un tema muy importe, el manejo de transacciones.

Temas a desarrollar

- 11.1. Insertando Filas
- 11.2. Modificando Datos
- 11.3. Eliminando Filas
- 11.4. Transacciones

11.1. Insertando Filas

La instrucción para insertar filas en una tabla es INSERT, esta instrucción tiene varias alternativas que desarrollaremos a continuación.

11.1.1. Caso 1

Sintaxis

```
INSERT [INTO] nombre_tabla [ ( lista_de_columnas ) ]
VALUES ( lista_de_valores ), . . .
[ ON DUPLICATE KEY UPDATE nombre_columna = expr, ... ] ;
```

Si se especifica la cláusula ON DUPLICATE KEY UPDATE (nueva desde MySQL 4.1.0), y se inserta una fila que puede provocar un valor duplicado en una clave PRIMARY KEY o UNIQUE, se realiza un UPDATE (actualización) de la fila antigua.

11.1.1.1. Inserciones una Sola Fila

Ejemplo 11 . 1

Si omitimos las columnas se debe insertar datos para todas las columnas, y en el orden que fueron creadas.

En este ejemplo estamos insertando una nueva sucursal y un nuevo empleado.

```
mysql> insert into sucursal values
    -> ('123','Pacherrez','Chiclayo','Av. Bolognesi 234',0);
Query OK, 1 row affected (0.02 sec)

mysql> insert into empleado values
    -> ('4321','Salazar','Castro','Sara','Chiclayo',
    -> 'Calle Casinelli 654','ssalazar','diva');
Query OK, 1 row affected (0.13 sec)

mysql>
```

11.1.1.2. Insertando Valores Nulos

Ejemplo 11 . 2

Si se omiten algunas columnas, estas deben aceptar valores nulos o deben tener definido un valor por defecto.

En este ejemplo estamos insertando un nuevo empleado a una sucursal, las columnas que no figuran en la lista de columnas tomarán valor NULL.

```
mysql> insert into
asignado(chr_asigcodigo,chr_sucucodigo,chr_emplcodigo,
    -> dtt_asigfechaalta) values('345678','123','4321','2008-07-15');
Query OK, 1 row affected (0.08 sec)

mysql> select
    ->     chr_asigcodigo as codigo,
    ->     chr_sucucodigo as sucursal,
    ->     chr_emplcodigo as empleado,
    ->     dtt_asigfechaalta as alta,
    ->     dtt_asigfechabaja as baja
    -> from asignado
    -> where chr_asigcodigo = '345678';
+-----+-----+-----+-----+
| codigo | sucursal | empleado | alta      | baja   |
+-----+-----+-----+-----+
| 345678 | 123      | 4321     | 2008-07-15 | NULL   |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

11.1.1.3. Insertando Varias Filas

La instrucción **insert** soporta la especificación de varias filas en una sola instrucción.

Ejemplo 11 . 3

En este ejemplo se ilustra como insertar varias filas con una sola instrucción **insert**, el ejemplo inserta 5 nuevas sucursales.

```
mysql> insert into sucursal values
    -> ('880','Sucursal 880','Lima','Lima',0),
    -> ('881','Sucursal 881','Lima','Lima',0),
    -> ('882','Sucursal 882','Lima','Lima',0),
    -> ('883','Sucursal 883','Lima','Lima',0),
    -> ('884','Sucursal 884','Lima','Lima',0);
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```

mysql> select chr_sucucodigo, vch_sucunombre
-> from sucursal where chr_sucucodigo >= '880';
+-----+-----+
| chr_sucucodigo | vch_sucunombre |
+-----+-----+
| 880           | Sucursal 880   |
| 881           | Sucursal 881   |
| 882           | Sucursal 882   |
| 883           | Sucursal 883   |
| 884           | Sucursal 884   |
+-----+-----+
5 rows in set (0.01 sec)

mysql>

```

11.1.2. Caso 2

Sintaxis

```

INSERT [INTO] nombre_tabla
SET nombre_columna = {expr | DEFAULT}, . . .
[ ON DUPLICATE KEY UPDATE nombre_columna = expr, . . . ]

```

Ejemplo 11 . 4

El siguiente ejemplo inserta un nuevo empleado, note la manera como se utiliza la instrucción insert.

```

mysql> insert into empleado
-> set chr_emplcodigo = '8765',
->      vch_emplpaterno = 'Sifuentes',
->      vch_emplmaterno = 'Caballero',
->      vch_emplnombre = 'Jorge',
->      vch_emplciudad = 'Lima',
->      vch_empldireccion = 'Lima',
->      vch_emplusuario = 'jorge',
->      vch_emplclave = '123456';
Query OK, 1 row affected (0.03 sec)

mysql> select chr_emplcodigo, vch_emplpaterno, vch_emplnombre
-> from empleado
-> where chr_emplcodigo = '8765';
+-----+-----+-----+
| chr_emplcodigo | vch_emplpaterno | vch_emplnombre |
+-----+-----+-----+
| 8765          | Sifuentes       | Jorge          |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

11.1.3. Caso 3

Sintaxis

```
INSERT [INTO] nombre_tabla [ ( lista_columnas, . . . ) ]  
SELECT . . .  
[ ON DUPLICATE KEY UPDATE nombre_columna = expr, . . . ]
```

Esta sintaxis permite insertar los valores de una tabla en otra tabla.

Ejemplo 11 . 5

En este ejemplo vamos a crear una nueva tabla de nombre **emp1** donde grabaremos los datos de los empleados.

Note usted que primero creamos la tabla y luego ejecutamos la instrucción **insert**.

```
mysql> create table emp1  
-> codigo char(4) primary key,  
-> nombre varchar(50)  
-> );  
Query OK, 0 rows affected (0.08 sec)  
  
mysql> insert into emp1  
-> select chr_emplcodigo,  
-> concat(vch_emplpaterno, ' ', vch_emplmaterno, ' ', vch_emplnombre)  
-> from empleado;  
Query OK, 12 rows affected (0.02 sec)  
Records: 12 Duplicates: 0 Warnings: 0  
  
mysql> select * from emp1;  
+-----+-----+  
| codigo | nombre |  
+-----+-----+  
| 0001 | Romero Castillo, Carlos Alberto |  
| 0002 | Castro Vargas, Lidia |  
| 0003 | Reyes Ortiz, Claudia |  
| 0004 | Ramos Garibay, Angelica |  
| 0005 | Ruiz Zabaleta, Claudia |  
| 0006 | Cruz Tarazona, Ricardo |  
| 0007 | Diaz Flores, Edith |  
| 0008 | Sarmiento Bellido, Claudia Rocio |  
| 0009 | Pachas Sifuentes, Luis Alberto |  
| 0010 | Tello Alarcon, Hugo Valentin |  
| 0011 | Carrasco Vargas, Pedro Hugo |  
| 9999 | Internet Internet, internet |  
+-----+-----+  
12 rows in set (0.02 sec)
```

```
mysql>
```

11.1.4. Cláusula: ON DUPLICATE KEY UPDATE

Esta cláusula en la instrucción **insert** permite que en caso que ya exista el registro que estoy insertando, pueda modificar una ó más columnas, de esta manera ya no se produce ningún error.

Ejemplo 11 . 6

En este ejemplo intentamos insertar un empleado con código **8765**, pero como ya existe, solo modificamos los campos **vch_emplciudad**, **vch_empldirección**.

```
mysql> select vch_emplpaterno, vch_emplnombre, vch_emplciudad,
vch_empldirección
-> from empleado
-> where chr_emplcodigo = '8765';
+-----+-----+-----+
--+
| vch_emplpaterno | vch_emplnombre | vch_emplciudad |
vch_empldirección |
+-----+-----+-----+
--+
| Sifuentes       | Jorge          | Lima           | Lima
|
+-----+-----+-----+
--+
1 row in set (0.00 sec)

mysql> INSERT INTO empleado(chr_emplcodigo,vch_emplpaterno,
-> vch_emplmaterno,vch_emplnombre,vch_emplciudad,
-> vch_empldirección,vch_emplusuario,vch_emplclave)
-> VALUES('8765','Sifuentes','Caballero','Jorge',
-> 'Trujillo','Trujillo','jorge','123456')
-> ON DUPLICATE KEY UPDATE
-> vch_emplciudad = 'Trujillo',
-> vch_empldirección = 'Trujillo';
Query OK, 2 rows affected (0.03 sec)

mysql> select vch_emplpaterno, vch_emplnombre, vch_emplciudad,
vch_empldirección
-> from empleado
-> where chr_emplcodigo = '8765';
+-----+-----+-----+
--+
| vch_emplpaterno | vch_emplnombre | vch_emplciudad |
vch_empldirección |
+-----+-----+-----+
--+
| Sifuentes       | Jorge          | Trujillo      | Trujillo
|
+-----+-----+-----+
--+
1 row in set (0.00 sec)

mysql>
```

11.2. Modificando Datos

Podemos modificar valores de las filas de una tabla usando la sentencia **UPDATE**. En su forma más simple, los cambios se aplican a todas las filas, y a las columnas que especifiquemos.

Sinatxis

```
UPDATE nombre_tabla  
SET nombre_columna1 = expr1 [, nombre_columna2 = expr2 . . . ]  
[ WHERE condición ];
```

11.2.1. Actualización Simple

Este tipo de actualización modifica todas las filas de una tabla, en la mayoría de los casos no es lo mas recomendable.

Ejemplo 11 . 7

En el presente ejemplo estamos incrementando en 10% el costo por movimiento.

```
mysql> select * from costomovimiento;  
+-----+-----+  
| chr_monecodigo | dec_costimporte |  
+-----+-----+  
| 01 | 2.00 |  
| 02 | 0.60 |  
+-----+-----+  
2 rows in set (0.00 sec)  
  
mysql> update costomovimiento  
      -> set dec_costimporte = dec_costimporte * 1.10;  
Query OK, 2 rows affected (0.05 sec)  
Rows matched: 2    Changed: 2    Warnings: 0  
  
mysql> select * from costomovimiento;  
+-----+-----+  
| chr_monecodigo | dec_costimporte |  
+-----+-----+  
| 01 | 2.20 |  
| 02 | 0.66 |  
+-----+-----+  
2 rows in set (0.00 sec)  
  
mysql>
```

11.2.2. Seleccionando las Filas a Actualizar

En la mayoría de los casos la actualización de datos se realiza sobre filas específicas, esto se consigue utilizando la cláusula **WHERE**.

Ejemplo 11 . 8

Con el fin de promocionar el ahorro en moneda nacional, el banco en una promoción inusual va a regalar 10% del saldo de la cuenta con mayor saldo a todos sus clientes, esta promoción solo se aplica a las cuentas en Nuevos Soles, y el importe será incrementado automáticamente en sus saldos.

Como primer paso consultemos los sueldos de las cuentas en Nuevos Soles:

```
mysql> select chr_cuencodigo, chr_monecodigo, dec_cuensaldo
      -> from cuenta
      -> where chr_monecodigo = '01';
+-----+-----+-----+
| chr_cuencodigo | chr_monecodigo | dec_cuensaldo |
+-----+-----+-----+
| 00100001      | 01           |    6900.00   |
| 00200001      | 01           |    7000.00   |
| 00200002      | 01           |    6800.00   |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

mysql>

Luego, debemos determinar el saldo mayor, y lo almacenaremos en una variable, en este caso la variable es `@saldo_mayor`:

```
mysql> select @saldo_mayor := max(dec_cuensaldo)
      -> from cuenta
      -> where chr_monecodigo = '01';
+-----+
| @saldo_mayor := max(dec_cuensaldo) |
+-----+
|          7000.00 |
+-----+
1 row in set (0.00 sec)
```

mysql>

Ahora debemos incrementar el saldo de todos los cuentas en Nuevo Soles con el 10% del valor que se almacena en la variable `@saldo_mayor`:

```
mysql> update cuenta
      -> set dec_cuensaldo = dec_cuensaldo + @saldo_mayor * 0.10
      -> where chr_monecodigo = '01';
Query OK, 3 rows affected (0.03 sec)
Rows matched: 3    Changed: 3    Warnings: 0
```

mysql>

Finalmente consultemos si se realizaron los cambios:

```
mysql> select chr_cuencodigo, chr_monecodigo, dec_cuensaldo
      -> from cuenta
      -> where chr_monecodigo = '01';
+-----+-----+-----+
| chr_cuencodigo | chr_monecodigo | dec_cuensaldo |
+-----+-----+-----+
| 00100001      | 01           |    7600.00   |
| 00200001      | 01           |    7700.00   |
| 00200002      | 01           |    7500.00   |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

mysql>

11.2.3. Actualizando Columnas con Subconsultas

El uso de subconsultas nos permite realizar actualizaciones de columnas con valores que se calcularan dinámicamente, incluso se puede utilizar correlación de datos.

Ejemplo 11 . 9

En otra promoción inusual, la cuenta **0010002** ha sido beneficiada con el 15% de importe del depósito de mayor valor.

Primero consultemos el saldo de la cuenta 00100002:

```
mysql> select chr_cuencodigo, dec_cuensaldo
    -> from cuenta
    -> where chr_cuencodigo = '00100002';
+-----+-----+
| chr_cuencodigo | dec_cuensaldo |
+-----+-----+
| 00100002      |      4500.00 |
+-----+-----+
1 row in set (0.00 sec)
```

```
mysql>
```

Consultemos su depósito de mayor importe:

```
mysql> select max(dec_moviimporte)
    -> from movimiento
    -> where chr_cuencodigo = '00100002'
    -> and chr_tipocodigo = '003';
+-----+
| max(dec_moviimporte) |
+-----+
|          2200.00 |
+-----+
1 row in set (0.00 sec)
```

```
mysql>
```

Ejecutemos la actualización:

```
mysql> update cuenta
    -> set dec_cuensaldo = dec_cuensaldo +
    ->       (select max(dec_moviimporte)
    ->         from movimiento
    ->        where chr_cuencodigo = '00100002'
    ->        and chr_tipocodigo = '003') * 0.15
    ->       where chr_cuencodigo = '00100002';
Query OK, 1 row affected (0.03 sec)
Rows matched: 1    Changed: 1    Warnings: 0
```

```
mysql>
```

Finalmente consultemos el resultado:

```
mysql> select chr_cuencodigo, dec_cuensaldo
    -> from cuenta
    -> where chr_cuencodigo = '00100002';
+-----+-----+
| chr_cuencodigo | dec_cuensaldo |
+-----+-----+
```

```
| 00100002      |      4830.00 |
+-----+-----+
1 row in set (0.00 sec)
```

mysql>

Ejemplo 11 . 10

Supongamos que necesitamos una tabla donde se guarde un resumen de los ingresos y salidas por cada cuenta, tal como se ilustra a continuación:

Cuenta	Ingresos	Salidas
00100001		
00100002		
00200001		
00200002		
00200003		

Como primer paso, crearemos la tabla, le asignaremos el nombre resumen:

```
mysql> create table resumen(
->     cuenta char(8) not null primary key,
->     ingresos decimal(12,2) not null default 0.0,
->     salidas decimal(12,2) not null default 0.0
-> ) engine = innodb;
Query OK, 0 rows affected (0.08 sec)
```

mysql>

Luego cargaremos los números de cuenta:

```
mysql> insert into resumen( cuenta )
-> select chr_cuencodigo from cuenta;
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

mysql>

Ahora actualizamos las columnas **ingresos** y **salidas** utilizando subconsultas correlacionadas:

```
mysql> update resumen
-> set ingresos = ( select sum(m.dec_moviimporte)
->                     from movimiento as m
->                     inner join tipomovimiento as t
->                     on m.chr_tipocodigo = t.chr_tipocodigo
->                     where t.vch_tipoaccion = 'INGRESO'
->                     and m.chr_cuencodigo = resumen.cuenta ),
->     salidas = ( select sum(m.dec_moviimporte)
->                     from movimiento as m
->                     inner join tipomovimiento as t
->                     on m.chr_tipocodigo = t.chr_tipocodigo
->                     where t.vch_tipoaccion = 'SALIDA'
->                     and m.chr_cuencodigo = resumen.cuenta );
Query OK, 5 rows affected (0.01 sec)
Rows matched: 5  Changed: 5  Warnings: 0
```

mysql>

Finalmente consultamos el contenido de la tabla resumen:

```
mysql> select * from resumen;
+-----+-----+-----+
| cuenta | ingresos | salidas |
+-----+-----+-----+
| 00100001 | 9000.00 | 2100.00 |
| 00100002 | 5500.00 | 1000.00 |
| 00200001 | 25000.00 | 18000.00 |
| 00200002 | 8000.00 | 1200.00 |
| 00200003 | 7500.00 | 1500.00 |
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

11.2.4. Error de Integridad Referencial

El error de integridad referencial se da cuando tratamos de modificar la clave primaria de un registro que tiene registros relacionados en otra tabla.

Ejemplo 11 . 11

En el siguiente ejemplo se intenta cambiar el número de cuenta **00100001**, se puede notar que no es posible por que ya tiene registros relacionados en la tabla movimiento.

```
mysql> update cuenta
      -> set chr_cuencodigo = '00100100'
      -> where chr_cuencodigo = '00100001';
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign
key
constraint fails (`eurekabank/movimiento`, CONSTRAINT
`fk_movimiento_cuenta` FOREIGN KEY (`chr_cuencodigo`) REFERENCES
`cuenta` (`chr_cuencodigo`))

mysql>
```

Nota

Esta operación se pudo haber realizado si la definición de la restricción **FOREIGN KEY** en la tabla movimiento para la acción **UPDATE** se hubiera definido como **CASCADE**.

11.3. Eliminando Filas

11.3.1. Introducción

La eliminación de filas es un proceso poco usual en sistemas comerciales, por el contrario existe un cambio de estado de la fila.

Por ejemplo si nos desanimamos de la compra de un producto por alguna razón, el procedimiento será devolver el producto y a cambio nos darán una nota de crédito, esta nota de crédito nos servirá como dinero en efectivo para realizar otra compra, la factura anterior cambiará de estado, pero no será eliminada.

Sintaxis

```
DELETE FROM nombre_tabla
[ WHERE condición ] ;
```

Para ilustrar los ejemplos de este tema crearemos una tabla auxiliar a partir de la tabla **empleado**, a esta tabla le asignaremos el nombre **emp_aux**, la instrucción es la siguiente:

```
mysql> create table emp_aux
    -> engine = innodb
    -> select * from empleado;
Query OK, 12 rows affected (0.14 sec)
Records: 12  Duplicates: 0  Warnings: 0

mysql>
```

Para verificar el estado de la tabla **emp_aux** utilice la instrucción:

```
SHOW TABLE STATUS LIKE 'emp_aux' \G;
```

Podrá usted notar que es de tipo INNODB, lo que quiere decir que soporta transacciones.

11.3.2. Eliminar Todas las Filas de una Tabla

Cuando no utilizamos la cláusula WHERE se eliminan todas las filas de la tabla referenciada.

Ejemplo 11 . 12

En este ejemplo se va eliminar todos los registros de la tabla **emp_aux**, pero dentro de una transacción, para luego recuperar todos los registros eliminados.

Inicio de transacción:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

Eliminar todos los registros:

```
mysql> delete from emp_aux;
Query OK, 12 rows affected (0.00 sec)
```

Verificar que se han eliminado todos los registros:

```
mysql> select count(*) from emp_aux;
+-----+
| count(*) |
+-----+
|      0   |
+-----+
1 row in set (0.00 sec)
```

Cancelar transacción:

```
mysql> rollback;
Query OK, 0 rows affected (0.02 sec)
```

Verificar que se recuperaron los registros:

```
mysql> select count(*) from emp_aux;
+-----+
| count(*) |
+-----+
|      12  |
+-----+
```

```
1 row in set (0.00 sec)

mysql>
```

11.3.3. Seleccionando las Filas a Eliminar

En caso de ser necesario eliminar una ó mas filas, debe utilizar la cláusula WHERE para seleccionar las filas a eliminar, caso contrario estaría afectando la consistencia de la base de datos.

11.3.3.1. Eliminando una Sola Fila

Debe crear la condición que identifica únicamente la fila a eliminar, normalmente se utiliza la clave primaria.

Ejemplo 11 . 13

En este ejemplo eliminaremos al empleado de código **0010**.

Verificar que el empleado de código **0010** existe:

```
mysql> select chr_emplcodigo, vch_emplpaterno, vch_emplnombre
-> from emp_aux
-> where chr_emplcodigo = '0010';
+-----+-----+
| chr_emplcodigo | vch_emplpaterno | vch_emplnombre |
+-----+-----+-----+
| 0010           | Tello          | Hugo Valentin |
+-----+-----+
1 row in set (0.00 sec)
```

Eliminando el empleado de código **0010**:

```
mysql> delete from emp_aux
-> where chr_emplcodigo = '0010';
Query OK, 1 row affected (0.02 sec)
```

Verificando que el empleado ya no existe en la tabla **emp_aux**:

```
mysql> select chr_emplcodigo, vch_emplpaterno, vch_emplnombre
-> from emp_aux
-> where chr_emplcodigo = '0010';
Empty set (0.00 sec)
```

```
mysql>
```

11.3.3.2. Eliminando un Grupo de Filas

Debe crear la condición que o criterio que identifica a todo el grupo de filas que quiere eliminar.

Ejemplo 11 . 14

En este ejemplo se eliminaran a todos los empleados de la ciudad de Lima.

Verificar cuantos empleados hay en la ciudad de Lina:

```
mysql> select count(*) from emp_aux
-> where vch_emplciudad = 'Lima';
+-----+
| count(*) |
+-----+
```

```
+-----+
|      3 |
+-----+
1 row in set (0.00 sec)
```

Eliminar los empleados:

```
mysql> delete from emp_aux
-> where vch_emplciudad = 'Lima';
Query OK, 3 rows affected (0.03 sec)
```

Verificar que las filas fueron eliminadas:

```
mysql> select count(*) from emp_aux
-> where vch_emplciudad = 'Lima';
+-----+
| count(*) |
+-----+
|      0 |
+-----+
1 row in set (0.00 sec)
```

```
mysql>
```

11.3.4. Uso de Subconsultas

También es posible utilizar subconsultas con la instrucción DELETE.

Ejemplo 11 . 15

El siguiente ejemplo elimina todos los empleados de la tabla **emp_aux** que pertenecen a la sucursal **Sipan** en la ciudad de **Chiclayo**.

Averiguamos el código de la sucursal **Sipan**:

```
mysql> select chr_sucucodigo, vch_sucunombre, vch_sucuciudad
-> from sucursal;
+-----+-----+-----+
| chr_sucucodigo | vch_sucunombre | vch_sucuciudad |
+-----+-----+-----+
| 001            | Sipan          | Chiclayo       |
| 002            | Chan Chan      | Trujillo       |
| 003            | Los Olivos     | Lima           |
| 004            | Pardo          | Lima           |
| 005            | Misti          | Arequipa       |
| 006            | Machupicchu    | Cusco          |
| 007            | Grau           | Piura          |
+-----+-----+-----+
7 rows in set (0.00 sec)
```

Eliminamos los empleados:

```
mysql> delete from emp_aux
-> where chr_emplcodigo in
->      ( select chr_emplcodigo from asignado
->        where chr_sucucodigo = '001' );
Query OK, 2 rows affected (0.03 sec)
```

```
mysql>
```

11.3.5. Error de Integridad Referencial

Este tipo de error se produce cuando intentamos eliminar una fila que tiene registros relacionados en otra tabla.

Ejemplo 11 . 16

En el siguiente ejemplo se intentará eliminar al cliente de código **00001**, se produce un error por que tiene registros relacionados en la tabla **cuenta**.

```
mysql> delete from cliente
      -> where chr_cliecodigo = '00001';
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign
key
constraint fails (`eurekabank/cuenta`, CONSTRAINT `fk_cuenta_cliente`
FOREIGN KEY (`chr_cliecodigo`) REFERENCES `cliente`
(`chr_cliecodigo`))

mysql>
```

11.3.6. Truncando una Tabla

La instrucción TRUNCATE permite eliminar todos los registros de una tabla, lógicamente es equivalente a la instrucción DELETE sin la cláusula WHERE, pero más rápida.

Al usar TRUNCATE debemos tener en cuenta que esta instrucción no es transaccional, esto quiere decir, que aún ejecutándola dentro de una transacción las filas de la tabla serán eliminadas definitivamente, la instrucción ROLLBACK no las recuperará.

Ejemplo 11 . 17

En este ejemplo se está eliminando todas las filas de la tabla **emp_aux**.

```
mysql> truncate table emp_aux;
Query OK, 8 rows affected (0.01 sec)

mysql>
```

11.4. Transacciones

11.4.1. Introducción

Una transacción es un grupo de acciones que hacen transformaciones consistentes en las tablas, preservando la consistencia de la base de datos.

Una base de datos está en un estado consistente si obedece todas las restricciones de integridad definidas sobre ella. Los cambios de estado ocurren debido a actualizaciones, inserciones, y eliminación de datos.

Se quiere asegurar que la base de datos nunca entre en un estado de inconsistencia. Sin embargo, durante la ejecución de una transacción, la base de datos puede estar temporalmente en un estado inconsistente. El punto importante aquí es asegurar que la base de datos regresa a un estado consistente al finalizar la ejecución de una transacción, tal como se puede apreciar en el siguiente gráfico.

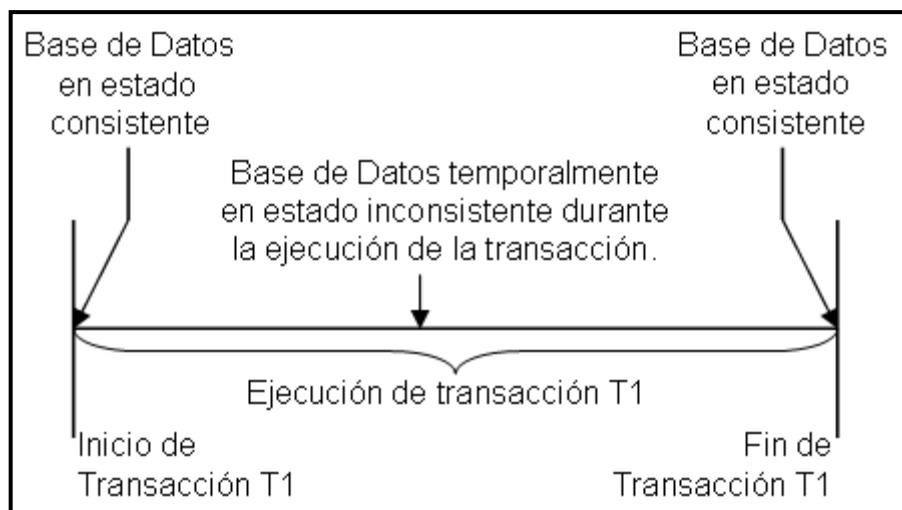


Figura 11 . 1 Esta es la base de datos durante una transacción.

Lo que se persigue con el manejo de transacciones es por un lado tener una transparencia adecuada de las acciones concurrentes a una base de datos y por otro lado tener una transparencia adecuada en el manejo de las fallas que se pueden presentar.

11.4.2. Propiedades de una Transacción

Una transacción debe tener las propiedades ACID, que son las iniciales en inglés de las siguientes características: Atomicity, Consistency, Isolation, Durability.

11.4.2.1. Atomicity (Atomicidad)

Una transacción constituye una unidad atómica de ejecución, por lo tanto, garantiza que se ejecuten todas sus instrucciones ó no se ejecute ninguna de ellas.

11.4.2.2. Consistency (Coherencia)

Una transacción mantiene la coherencia de los datos, transformando un estado coherente de datos en otro estado coherente de datos. Los datos enlazados por una transacción deben conservarse semánticamente.

11.4.2.3. Isolation (Aislamiento)

Una transacción es una unidad de aislamiento y cada una se produce aislada e independientemente de las transacciones concurrentes. Una transacción nunca debe ver las fases intermedias de otra transacción.

11.4.2.4. Durability (Durabilidad)

Una transacción es una unidad de recuperación. Si una transacción tiene éxito, sus actualizaciones persisten, aun cuando falle el equipo o se apague. Si una transacción no tiene éxito, el sistema permanece en el estado anterior antes de la transacción.

11.4.3. Operación de Transacciones

El siguiente gráfico ilustra el funcionamiento de una transacción, cuando es confirmada y cuando es cancelada.

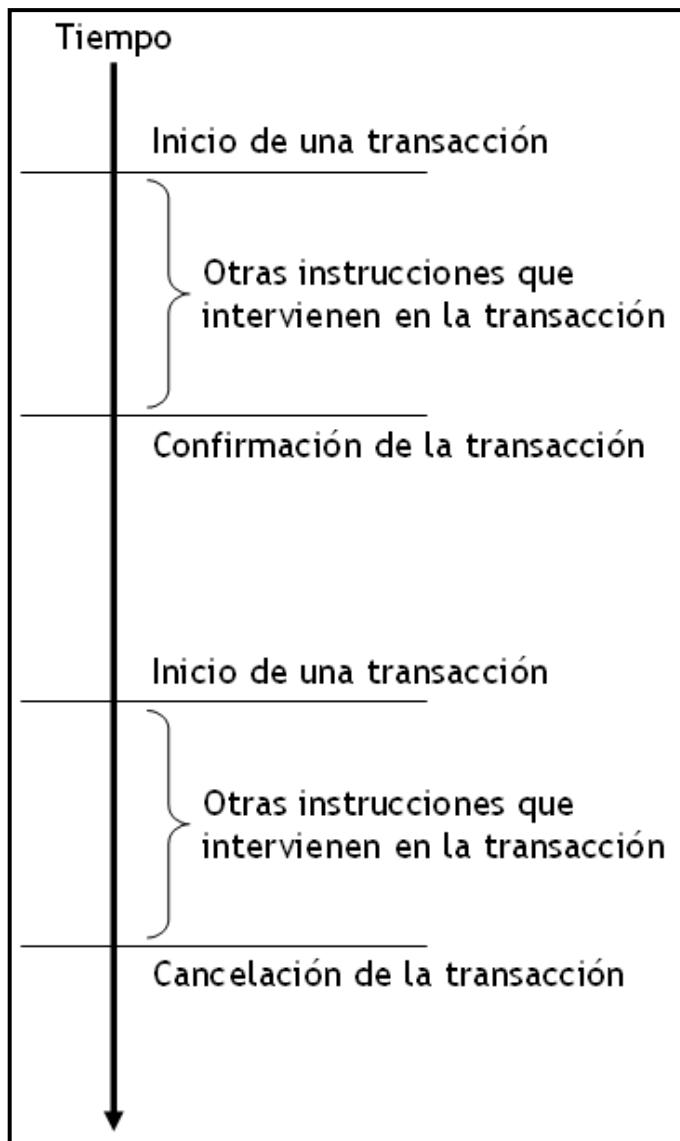


Figura 11 . 2 Esquema de funcionamiento de una transacción en el tiempo.

11.4.3.1. Inicio de una transacción

Sintaxis

```
START TRANSACTION;
```

```
BEGIN;
```

```
BEGIN WORK;
```

Por defecto, MySQL se ejecuta en modo AUTOCOMMIT, esto quiere decir que cada instrucción es confirmada inmediatamente.

Cualquiera de estas instrucciones inicia una transacción de manera explícita, las instrucciones que se ejecuten a continuación formaran parte de esta transacción.

11.4.3.2. Confirmación de una transacción

Sintaxis

```
COMMIT;
```

Esta instrucción confirma las instrucciones que se ejecutaron después de iniciar una transacción con START TRANSACTION. Después que una transacción es confirmada, ya no puede ser revertida.

Ejemplo 11 . 18

En este ejemplo ilustraremos el uso de una transacción, note que este ejemplo es ilustrativo, en aplicaciones reales las transacciones pueden estar conformadas por una, dos, tres, ó más instrucciones.

Inicio de una transacción:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

Instrucciones de la transacción:

```
mysql> insert into sucursal
-> values('888','Minka','Lima','Av. Argentina 3093',0);
Query OK, 1 row affected (0.00 sec)
```

Confirmar transacción:

```
mysql> commit;
Query OK, 0 rows affected (0.02 sec)
```

11.4.3.3. Cancelar una transacción

Sintaxis

```
ROLLBACK;
```

Esta instrucción cancela las instrucciones que se ejecutaron después de iniciar una transacción con START TRANSACTION.

Ejemplo 11 . 19

En este ejemplo se ilustrará el funcionamiento de la instrucción ROLLBACK.

Inicio de una transacción:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

Cambiemos el saldo de todas las cuentas:

```
mysql> update cuenta
-> set dec_cuensaldo = 50000;
Query OK, 5 rows affected (0.00 sec)
Rows matched: 5    Changed: 5    Warnings: 0
```

Verifiquemos los cambios:

```
mysql> select chr_cuencodigo, dec_cuensaldo  
-> from cuenta;  
+-----+-----+  
| chr_cuencodigo | dec_cuensaldo |  
+-----+-----+  
| 00100001      |      50000.00 |  
| 00100002      |      50000.00 |  
| 00200001      |      50000.00 |  
| 00200002      |      50000.00 |  
| 00200003      |      50000.00 |  
+-----+-----+  
5 rows in set (0.00 sec)
```

Cancelemos la transacción:

```
mysql> rollback;  
Query OK, 0 rows affected (0.02 sec)
```

Verifiquemos que los cambios fueron cancelados:

```
mysql> select chr_cuencodigo, dec_cuensaldo  
-> from cuenta;  
+-----+-----+  
| chr_cuencodigo | dec_cuensaldo |  
+-----+-----+  
| 00100001      |      6900.00 |  
| 00100002      |      4500.00 |  
| 00200001      |      7000.00 |  
| 00200002      |      6800.00 |  
| 00200003      |      6000.00 |  
+-----+-----+  
5 rows in set (0.00 sec)
```

```
mysql>
```



12

Programación

Programar procesos en el servidor es muy importante, ya que ello conlleva a tener rutinas que pueden ser reutilizadas en varias aplicaciones y en la mayoría de casos actualizadas sin tener la necesidad de modificar la aplicación cliente.

Temas a desarrollar

- 12.1. Funciones de usuario
- 12.2. Procedimientos almacenados
- 12.3. Manejo de variables
- 12.4. Sentencias de control de flujo
- 12.5. Manejo de Errores
- 12.6. Cursos

12.1. Funciones de usuario

12.1.1. Crear nuevas funciones

Las funciones solo pueden tener parámetros de entrada, por lo que no es valido especificar si el IN, OUT ó INOUT.

Sintaxis:

```
CREATE FUNCTION nombre_función ( [ parámetros ] )
RETURNS tipo_resultado
cuerpo_función
```

parámetros:

```
nombre_parámetro tipo [, ...]
```

tipo_resultado, tipo:

Puede ser cualquier tipo de dato valido en MySQL.

cuerpo_función:

```
return expresión
```

ó

```
begin
    . . . .
    .
    .
    return expresión;
end
```

12.1.2. Eliminar una función

Sintaxis:

```
DROP FUNCTION [IF EXISTS] nombre_función
```

12.1.3. Modificar una función

En la versión actual (5.1) no es posible modificar el contenido de una función. Si tenemos la necesidad de modificar una función, primero debemos eliminarla y luego volver a crearla.

Ejemplo 12 . 1

En este ejemplo crearemos una función que sume dos números, el script es el siguiente:

```
delimiter $$

create function fn_suma( a int, b int)
returns int
return (a+b) $$

delimiter ;
```

A continuación tenemos el resultado de su ejecución:

```
mysql> delimiter $$

mysql> create function fn_suma( a int, b int)
    -> returns int
    -> return (a+b)$$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Y finalmente, como utilizamos la función:

```
mysql> select fn_suma(35,76);
+-----+
| fn_suma(35,76) |
+-----+
|          111 |
+-----+
1 row in set (0.00 sec)

mysql> set @suma = fn_suma(543,678);
Query OK, 0 rows affected (0.00 sec)

mysql> select @suma;
+-----+
| @suma |
+-----+
| 1221  |
+-----+
1 row in set (0.02 sec)

mysql>
```

Ejemplo 12 . 2

En este ejemplo creamos una función para consultar el saldo de una cuenta, el script es el siguiente:

```
delimiter //

create function fn_consulta_saldo( p_cuenta char(8) )
returns decimal(12,2)
begin
    declare saldo decimal(12,2);
    select dec_cuensaldo into saldo
        from cuenta
        where chr_cuencodigo = p_cuenta;
    return saldo;
end //

delimiter ;
```

A continuación tenemos el resultado de su ejecución:

```
mysql> delimiter //
mysql> create function fn_consulta_saldo( p_cuenta char(8) )
-> returns decimal(12,2)
-> begin
->     declare saldo decimal(12,2);
->     select dec_cuensaldo into saldo
->         from cuenta
->         where chr_cuencodigo = p_cuenta;
->     return saldo;
-> end //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Y finalmente, como ejecutamos la función:

```
mysql> select fn_consulta_saldo( '00100001' );
+-----+
| fn_consulta_saldo( '00100001' ) |
+-----+
|           6900.00 |
+-----+
1 row in set (0.01 sec)

mysql>
```

12.2. Procedimientos almacenados

12.2.1. Creación de procedimientos

Sintaxis:

```
CREATE PROCEDURE nombre_procedimiento ( [ parámetros ] )
cuerpo_procedimiento
```

parámetros:

```
[ IN | OUT | INOUT ] nombre_parámetro tipo [, ...]
```

tipo:

Puede ser cualquier tipo de dato valido en MySQL.

cuerpo_procedimiento:

```
begin
    . . .
    .
    .
end
```

12.2.2. Eliminar un procedimiento

Sintaxis:

```
DROP PROCEDURE [ IF EXISTS] nombre_procedimiento
```

12.2.3. Modificar un procedimiento

En la versión actual (5.1) no es posible modificar el contenido de un procedimiento. Si tenemos la necesidad de modificar una procedimiento, primero debemos eliminarlo y luego volver a crearlo.

Ejemplo 12 . 3

En este ejemplo tenemos un procedimiento para consultar las cuentas de una sucursal, el script es el siguiente:

```
delimiter //
create procedure sp_consultar_cuentas( p_sucursal char(3) )
begin
    select chr_cuencodigo, chr_monecodigo, dec_cuensaldo
    from cuenta
    where chr_sucucodigo = p_sucursal;
end//
delimiter ;
```

A continuación tenemos la ejecución del script:

```
mysql> delimiter //
mysql> create procedure sp_consultar_cuentas( p_sucursal char(3) )
-> begin
->     select chr_cuencodigo, chr_monecodigo, dec_cuensaldo
->     from cuenta
->     where chr_sucucodigo = p_sucursal;
-> end//
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Y finalmente, como ejecutar el procedimiento:

```
mysql> call sp_consultar_cuentas( '001' );
+-----+-----+-----+
| chr_cuencodigo | chr_monecodigo | dec_cuensaldo |
+-----+-----+-----+
| 00100001      | 01           |      6900.00  |
| 00100002      | 02           |      4500.00  |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Ejemplo 12 . 4

En este ejemplo tenemos un procedimiento para consultar el saldo de una cuenta, se está utilizando un parámetro de entrada para comunicar la cuenta a consultar y un parámetro de salida para obtener el saldo de la cuenta.

El script es el siguiente:

```
delimiter //

create procedure sp_consultar_saldo
( in p_cuenta char(8), out p_saldo decimal(12,2) )
begin
    select dec_cuensaldo
    into p_saldo
    from cuenta
    where chr_cuencodigo = p_cuenta;
end//;

delimiter ;
```

A continuación tenemos la ejecución del script:

```
mysql> delimiter //

mysql> create procedure sp_consultar_saldo
-> ( in p_cuenta char(8), out p_saldo decimal(12,2) )
-> begin
->     select dec_cuensaldo
->     into p_saldo
->     from cuenta
->     where chr_cuencodigo = p_cuenta;
-> end//;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> delimiter ;
```

Y finalmente, como ejecutar el procedimiento:

```
mysql> call sp_consultar_saldo( '00100001', @saldo );
Query OK, 0 rows affected (0.06 sec)

mysql> select @saldo;
+-----+
| @saldo |
+-----+
| 6900.00 |
+-----+
1 row in set (0.02 sec)

mysql>
```

12.3. Manejo de variables

12.3.1. Sentencia DECLARE

Esta sentencia se usa para declarar variables locales. Para especificar un valor por defecto para la variable, se debe incluir la cláusula `DEFAULT`. El valor puede especificarse como

expresión, no necesita ser una constante. Si la cláusula DEFAULT no está presente, el valor inicial es NULL.

La visibilidad de una variable local es dentro del bloque BEGIN ... END donde está declarado. Puede usarse en bloques anidados excepto aquéllos que declaren una variable con el mismo nombre.

Sintaxis:

```
DECLARE var_name[,...] type [DEFAULT value]
```

Casos:

```
DECLARE v_saldo DECIMAL(12,2);
DECLARE v_promedio DECIMAL(12,2) DEFAULT 15.0;
```

12.3.2. Sentencia SET

La sentencia SET permite asignarle un valor a una ó varias variables, Se pueden utilizar variables declaradas localmente, globalmente y variables del sistema.

Sintaxis:

```
SET var_name = expr [, var_name = expr] ...
```

Casos:

```
SET v_saldo = 1500.00;
SET v_promedio = 800.00;
```

12.3.3. Sentencia SELECT . . . INTO

La sentencia SELECT almacena el valor de las columnas seleccionadas directamente en variables. Por lo tanto, debe retornar exactamente un registro.

Sintaxis:

```
SELECT col_name [, ... ] INTO var_name [, ... ]
FROM ...
```

Casos:

```
select count(*) into cantCuentas from cuenta;
select dec_cuensaldo into saldo
from cuenta where chr_cuencodigo = '00100001';
```

Ejemplo 12 . 5

En este ejemplo construiremos un procedimiento para consultar los depósitos que están por debajo del monto promedio de todos los depósitos.

El script es el siguiente:

```
delimiter //
create procedure sp_consultar_movimientos()
begin
    declare promedio decimal(12,2);
    select
        avg(dec_moviimporte)
        into promedio
        from movimiento
```

```

        where chr_tipocodigo = '003';
select
    chr_cuencodigo, int_movinumero, dtt_movifecha, dec_moviimporte
from movimiento
where chr_tipocodigo = '003'
and dec_moviimporte < promedio;
end //

delimiter ;

```

A continuación tenemos la ejecución del script:

```

mysql> delimiter //

mysql> create procedure sp_consultar_depositos()
-> begin
->     declare promedio decimal(12,2);
->     select
->         avg(dec_moviimporte)
->         into promedio
->         from movimiento
->         where chr_tipocodigo = '003';
->     select
->         chr_cuencodigo, int_movinumero, dtt_movifecha, dec_moviimporte
->         from movimiento
->         where chr_tipocodigo = '003'
->         and dec_moviimporte < promedio;
-> end //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

```

Finalmente, como ejecutar el procedimiento:

```

mysql> call sp_consultar_depositos();
+-----+-----+-----+-----+
| chr_cuencodigo | int_movinumero | dtt_movifecha | dec_moviimporte |
+-----+-----+-----+-----+
| 00100001      |           4 | 2008-02-14   |      2000.00   |
| 00100001      |           7 | 2008-03-15   |      1000.00   |
| 00100002      |           3 | 2008-02-13   |      2200.00   |
| 00100002      |           4 | 2008-03-08   |      1500.00   |
| 00200001      |           4 | 2008-01-11   |      1000.00   |
| 00200001      |           5 | 2008-01-13   |      2000.00   |
| 00200001      |           7 | 2008-01-19   |      2000.00   |
| 00200001      |          12 | 2008-02-04   |      2000.00   |
| 00200001      |          14 | 2008-02-13   |      2000.00   |
| 00200003      |           2 | 2008-01-17   |      1500.00   |
+-----+-----+-----+-----+
10 rows in set (0.00 sec)

Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql>

```

Ejemplo 12 . 6

En este ejemplo haremos un procedimiento para consultar de una cuenta el total de ingresos, salidas y su saldo respectivo.

El procedimiento deberá tener un parámetro tipo IN para ingresar el número de cuenta y tres parámetros tipo OUT para obtener el resultado.

El script del procedimiento es el siguiente:

```
delimiter //
```

```
create procedure sp_consultar_cuenta
( in p_cuenta char(8), out p_ingreso decimal(12,2),
  out p_salida decimal(12,2), out p_saldo decimal(12,2) )
begin

  -- Consultar saldo
  select dec_cuensaldo into p_saldo
  from cuenta where chr_cuencodigo = p_cuenta;

  -- Consultar ingreso y salida
  select
    sum(case when t.vch_tipoaccion = 'INGRESO'
      then m.dec_moviimporte else 0.0 end),
    sum(case when t.vch_tipoaccion = 'SALIDA'
      then m.dec_moviimporte else 0.0 end)
  into p_ingreso, p_salida
  from movimiento as m
  join tipomovimiento as t
  on m.chr_tipocodigo = t.chr_tipocodigo
  where m.chr_cuencodigo = p_cuenta;

end //

delimiter ;
```

A continuación tenemos la ejecución del script:

```
mysql> delimiter //

mysql> create procedure sp_consultar_cuenta
-> ( in p_cuenta char(8), out p_ingreso decimal(12,2),
->   out p_salida decimal(12,2), out p_saldo decimal(12,2) )
-> begin
->   -- Consultar saldo
->   select dec_cuensaldo into p_saldo
->   from cuenta where chr_cuencodigo = p_cuenta;
->   -- Consultar ingreso y salida
->   select
->     sum(case when t.vch_tipoaccion = 'INGRESO'
->       then m.dec_moviimporte else 0.0 end),
->     sum(case when t.vch_tipoaccion = 'SALIDA'
->       then m.dec_moviimporte else 0.0 end)
->   into p_ingreso, p_salida
->   from movimiento as m
->   join tipomovimiento as t
->   on m.chr_tipocodigo = t.chr_tipocodigo
->   where m.chr_cuencodigo = p_cuenta;
-> end //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Finalmente, tenemos su ejecución:

```
mysql> call sp_consultar_cuenta('00100001',@ingresos,@salidas,@saldo);
Query OK, 0 rows affected (0.00 sec)

mysql> select @ingresos, @salidas, @saldo;
+-----+-----+-----+
| @ingresos | @salidas | @saldo |
+-----+-----+-----+
| 9100.00 | 2100.00 | 7000.00 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

12.4. Sentencias de control de flujo

12.4.1. Sentencia IF

Esta sentencia permite seleccionar una sentencia o grupo de sentencia en base al cumplimiento de una condición.

Sintaxis:

```
IF condición_01 THEN
    lista_sentencias_01
ELSEIF condición_02 THEN
    lista_sentencias_02
ELSEIF condición_03 THEN
    lista_sentencias_03
. . .
. . .
[ELSE
    lista_sentencias_else]
END IF
```

Debemos tener en cuenta que también existe la función IF() que difiere de la sentencia IF.

12.4.2. Sentencia CASE

Se trata también de una sentencia selectiva, tiene dos formatos, uno basado en el valor de una expresión que actúa como selector de la sentencia o sentencias a ejecutar y el segundo formato está basado en condiciones, similar a la sentencia IF.

Sintaxis 01:

```
CASE expresión_selector
    WHEN valor_01 THEN
        lista_sentencias_01
    WHEN valor_02 THEN
        lista_sentencias_02
. .
. .
[ELSE
    lista_sentencias_else]
END CASE
```

Sintaxis 02:

```
CASE
    WHEN condición_01 THEN
        lista_sentencias_01
    WHEN condición_02 THEN
        lista_sentencias_02
. .
. .
[ELSE
    lista_sentencias_else]
END CASE
```

12.4.3. Sentencia LOOP

La sentencia LOOP implementa un bucle simple. Las sentencias dentro del bucle se ejecutan tantas veces hasta que finalice el bucle, usualmente con una sentencia LEAVE.

La sentencia LOOP puede etiquetarse, tal como lo puede ver en su sintaxis.

Sintaxis:

```
[etiqueta:] LOOP  
    lista_sentencias  
END LOOP [etiqueta]
```

12.4.5. Sentencia LEAVE

Esta sentencia se utiliza para salir de cualquier sentencia de control de flujo etiquetada. Puede usarse con BEGIN ... END o bucles.

Sintaxis:

```
LEAVE etiqueta
```

12.4.6. Sentencia ITERATE

Esta sentencia sólo se puede utilizar en sentencia LOOP, REPEAT y WHILE, se utiliza para volver al inicio del bucle.

Sintaxis:

```
ITERATE label
```

Ejemplo 12 . 7

El presente ejemplo es para ilustrar el funcionamiento de las sentencia LOOP, LEAVE e ITERATE.

El script es el siguiente:

```
delimiter //  
  
create procedure sp_demo_loop()  
begin  
    declare n int;  
    drop table if exists demo;  
    create temporary table demo(id int, dato varchar(50) );  
    set n = 0;  
    proceso: loop  
        set n = n + 1;  
        if( n = 5 ) then  
            iterate proceso;  
        end if;  
        insert into demo values(n,'Hola Gustavo');  
        if( n = 8 || n = 5) then  
            leave proceso;  
        end if;  
    end loop proceso;  
    select * from demo;  
end //  
  
delimiter ;
```

A continuación tenemos la ejecución del script:

```
mysql> delimiter //  
  
mysql> create procedure sp_demo_loop()  
-> begin  
->     declare n int;
```

```

->      drop table if exists demo;
->      create temporary table demo(id int, dato varchar(50) );
->      set n = 0;
->      proceso: loop
->          set n = n + 1;
->          if( n = 5 ) then
->              iterate proceso;
->          end if;
->          insert into demo values(n,'Hola Gustavo');
->          if( n = 8 || n = 5 ) then
->              leave proceso;
->          end if;
->      end loop proceso;
->      select * from demo;
-> end //
```

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

El bucle debería finalizar cuando n sea igual a 8 ó 5, sin embargo, cuando es igual a 5 regresa al inicio del bucle y no ejecuta las sentencias que están debajo del primer IF, por lo tanto, nunca finalizará cuando n sea igual a 5, y eso lo podemos comprobar en la ejecución.

Finalmente, tenemos la ejecución del procedimiento:

```

mysql> call sp_demo_loop();
+-----+-----+
| id   | dato    |
+-----+-----+
| 1   | Hola Gustavo |
| 2   | Hola Gustavo |
| 3   | Hola Gustavo |
| 4   | Hola Gustavo |
| 6   | Hola Gustavo |
| 7   | Hola Gustavo |
| 8   | Hola Gustavo |
+-----+-----+
7 rows in set (0.08 sec)

Query OK, 0 rows affected (0.09 sec)
```

mysql>

12.4.7. Sentencia REPEAT

Esta sentencia implementa un bucle que es controlado por una condición que se define en la cláusula UNTIL, el bucle finaliza cuando se cumple la condición.

Su característica principal es que la condición esta al finalizar el bucle.

Sintaxis:

```

[etiqueta:] REPEAT
    lista_sentencias
    UNTIL condición
END REPEAT [etiqueta]
```

12.4.8. Sentencia WHILE

Esta sentencia también implementa un bucle que es controlado por una condición que se define en la entrada del bucle, quiere decir que para ingresar al bucle se debe cumplir la condición y finaliza cuando la condición deje de cumplirse.

Sintaxis:

```
[etiqueta:] WHILE condición DO  
    lista_sentencias  
END WHILE [etiqueta]
```

Ejemplo 12 . 8

En este ejemplo estamos construyendo una función para calcular el factorial de un número.

El script es el siguiente:

```
delimiter //  
  
create function fn_factorial( n int )  
returns int  
begin  
    declare f int default 1;  
    while n > 1 do  
        set f = f * n;  
        set n = n - 1;  
    end while;  
    return f;  
end //  
  
delimiter ;
```

A continuación tenemos la ejecución del script:

```
mysql> delimiter //  
  
mysql> create function fn_factorial( n int )  
-> returns int  
-> begin  
->     declare f int default 1;  
->     while n > 1 do  
->         set f = f * n;  
->         set n = n - 1;  
->     end while;  
->     return f;  
-> end //  
Query OK, 0 rows affected (0.01 sec)  
  
mysql> delimiter ;
```

Finalmente, la ejecución de la función:

```
mysql> select fn_factorial( 5 );  
+-----+  
| fn_factorial( 5 ) |  
+-----+  
|           120 |  
+-----+  
1 row in set (0.00 sec)  
  
mysql>
```

12.5. Manejo de Errores

12.5.1. Condición nombrada

Una condición nombrada representa una condición de error y que necesita un tratamiento específico.

Sintaxis:

```
DECLARE nombre_condición CONDITION FOR condición
```

Donde:

condición:	SQLSTATE [VALUE] valor_sqlstate código_error_mysql
------------	---

12.5.2. Manejador de error

El manejo de errores se realiza declarando un manejador de errores, en el cual especificamos el error o los tipos de errores que queremos controlar.

Sintaxis:

```
DECLARE tipo HANDLER FOR condición [, ...]  
sentencias
```

Donde:

tipo:	CONTINUE EXIT UNDO
condición:	SQLSTATE [VALUE] valor_sqlstate condición_nombreda SQLWARNING NOT FOUND SQLEXCEPTION código_error_mysql

El manejador de error puede tratar una o varias condiciones. Si una de estas condiciones ocurre, la sentencia especificada es ejecutada.

Para un manejador de tipo CONTINUE, continúa la rutina actual tras la ejecución del manejador de error. Para un manejador de tipo EXIT, termina la ejecución de la sentencia compuesta BEGIN..END actual. El manejador de tipo UNDO todavía no está soportado.

- SQLWARNING es una abreviación para todos los códigos SQLSTATE que comienzan con 01.
- NOT FOUND es una abreviación para todos los códigos SQLSTATE que comienzan con 02.
- SQLEXCEPTION es una abreviación para todos los códigos SQLSTATE no tratados por SQLWARNING y NOT FOUND.
- Además de los valores SQLSTATE, los códigos de error de MySQL también son soportados.

Ejemplo 12 . 9

En este ejemplo haremos un procedimiento para registrar un depósito, el script es el siguiente:

```
DELIMITER $$

CREATE PROCEDURE sp_deposito(
    out p_estado varchar(500), -- Parámetro de salida
    p_cuenta char(8),
    p_importe decimal(12,2),
    p_empleado char(4)
```

```
)  
BEGIN  
    -- Declaraciones  
    DECLARE moneda char(2);  
    DECLARE costoMov decimal(12,2);  
    DECLARE cont int;  
    -- Control de Error  
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND  
BEGIN  
    rollback; -- Cancela Transacción  
    set p_estado := 'Error en el proceso de actualización.';  
END;  
-- Iniciar Transacción  
set p_estado = null;  
start transaction;  
-- Tabla Cuenta  
select  
    int_cuencontmov, chr_monecodigo  
    into cont, moneda  
    from cuenta  
    where chr_cuencodigo = p_cuenta;  
-- Tabla CostoMovimiento  
select  
    dec_costimporte  
    into costoMov  
    from costomovimiento  
    where chr_monecodigo = moneda;  
  
-- Registrar el deposito  
update cuenta  
    set dec_cuensaldo = dec_cuensaldo + p_importe,  
        int_cuencontmov = int_cuencontmov + 1  
    where chr_cuencodigo = p_cuenta;  
-- Insertar Movimiento  
set cont := cont + 1;  
insert into movimiento(chr_cuencodigo,int_movinumero,dtt_movifecha,  
    chr_emplcodigo,chr_tipocodigo,dec_moviimporte,chr_cuenreferencia)  
    values(p_cuenta,cont,current_date,p_empleado,'003',p_importe,null);  
-- Confirmar Transacción  
commit;  
set p_estado = 'ok';  
END$$  
  
DELIMITER ;
```

A continuación tenemos la ejecución del script:

```
mysql> DELIMITER $$  
  
mysql> CREATE PROCEDURE sp_deposito(  
    >     out p_estado varchar(500), -- Parámetro de salida  
    >     p_cuenta char(8),  
    >     p_importe decimal(12,2),  
    >     p_empleado char(4)  
    > )  
    > BEGIN  
    >     -- Declaraciones  
    >     DECLARE moneda char(2);  
    >     DECLARE costoMov decimal(12,2);  
    >     DECLARE cont int;  
    >     -- Control de Error  
    >     DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND  
    > BEGIN  
    >         rollback; -- Cancela Transacción  
    >         set p_estado := 'Error en el proceso de actualización.';  
    >     END;  
    >     -- Iniciar Transacción  
    >     set p_estado = null;
```

```

-> start transaction;
-> -- Tabla Cuenta
-> select
->         int_cuencontmov, chr_monecodigo
->         into cont, moneda
->         from cuenta
->         where chr_cuencodigo = p_cuenta;
-> -- Tabla CostoMovimiento
-> select
->         dec_costimporte
->         into costoMov
->         from costomovimiento
->         where chr_monecodigo = moneda;
->
-> -- Registrar el deposito
-> update cuenta
->         set dec_cuensaldo = dec_cuensaldo + p_importe,
->             int_cuencontmov = int_cuencontmov + 1
->             where chr_cuencodigo = p_cuenta;
-> -- Insertar Movimiento
-> set cont := cont + 1;
-> insert into movimiento(chr_cuencodigo,int_movinumero,dtt_movifecha,
->     chr_emplcodigo,chr_tipocodigo,dec_moviimporte,chr_cuenreferencia)
->     values(p_cuenta,cont,current_date,p_empleado,'003',p_importe,null);
-> -- Confirmar Transacción
-> commit;
-> set p_estado = 'ok';
-> END$$
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;

```

Finalmente, tenemos la ejecución del procedimiento:

```

mysql> select chr_cuencodigo, dec_cuensaldo, int_cuencontmov
-> from cuenta
-> where chr_cuencodigo = '00100001';
+-----+-----+-----+
| chr_cuencodigo | dec_cuensaldo | int_cuencontmov |
+-----+-----+-----+
| 00100001      |      7000.00 |          8 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> call sp_deposito( @estado, '00100001', 100.0, '0001' );
Query OK, 0 rows affected (0.13 sec)

mysql> select @estado;
+-----+
| @estado |
+-----+
| ok      |
+-----+
1 row in set (0.00 sec)

mysql> select chr_cuencodigo, dec_cuensaldo, int_cuencontmov
-> from cuenta
-> where chr_cuencodigo = '00100001';
+-----+-----+-----+
| chr_cuencodigo | dec_cuensaldo | int_cuencontmov |
+-----+-----+-----+
| 00100001      |      7100.00 |          9 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

Podemos notar que el saldo de la cuenta se ha incrementado en 100 y el contador se ha incrementado en 1.

12.6. Cursos

MySQL soporta cursos simples dentro de procedimientos y funciones almacenadas. Los cursos son de sólo lectura, y no permiten scrolling.

Los cursos deben declararse antes de declarar los manejadores de error, y las variables y condiciones deben declararse antes de declarar cursos o los manejadores de error.

12.6.1. Declaración de cursos

Sintaxis:

```
DECLARE nombre_cursor CURSOR FOR sentencia_select
```

Se puede definir varios cursos en una rutina, pero cada cursor debe tener un nombre único.

La sentencia **SELECT** **no puede** tener una cláusula **INTO**.

12.6.2. Abrir un cursor

Sintaxis:

```
OPEN nombre_cursor
```

El cursor que se pretende abrir debe estar declarado previamente.

12.6.3. Leer datos del cursor

Sintaxis:

```
FETCH nombre_cursor INTO variable1[, variable2] ...
```

La sentencia **FETCH** lee la siguiente fila (si existe) y guarda los datos recuperados en la lista de variables, y avanza el puntero del cursor.

Es necesario que el cursor se encuentre abierto.

12.6.4. Cerrar el cursor

Sintaxis:

```
CLOSE nombre_cursor
```

La sentencia **CLOSE** cierra un cursor previamente abierto. Si no se cierra explícitamente, un cursor se cierra al final de la sentencia compuesta en que se declara.

Ejemplo 12 . 10

En este ejemplo programaremos un procedimiento para consultas el estado de las cuentas, se debe mostrar el número de cuenta, saldo, el total de ingresos y el total de salidas.

El script del procedimiento es el siguiente:

```
delimiter //
create procedure sp_resumen_cuentas()
```

```

begin
    -- declaraciones
    declare v_cuenta char(8);
    declare v_ingeros decimal(12,2);
    declare v_salidas decimal(12,2);
    declare v_control_cursor int default 1;
    declare c_cuentas cursor for
        select cuenta from resumen;
    declare continue handler for not found
        set v_control_cursor = 0;
    -- tabla temporal para construir el resumen
    create temporary table resumen(
        cuenta char(8) null,
        saldo decimal(12,2) null,
        ingeros decimal(12,2) null,
        salidas decimal(12,2) null
    );
    -- inserto datos en la tabla resumen
    insert into resumen(cuenta,saldo)
        select chr_cuencodigo,dec_cuensaldo from cuenta;
    -- Proceso
    open c_cuentas;
    fetch c_cuentas into v_cuenta;
    while v_control_cursor = 1 do
        select sum(m.dec_moviimporte) into v_ingeros
            from movimiento as m
            join tipomovimiento as t
            on m.chr_tipocodigo = t.chr_tipocodigo
            where t.vch_tipoaccion = 'INGRESO'
            and m.chr_cuencodigo = v_cuenta;
        select sum(m.dec_moviimporte) into v_salidas
            from movimiento as m
            join tipomovimiento as t
            on m.chr_tipocodigo = t.chr_tipocodigo
            where t.vch_tipoaccion = 'SALIDA'
            and m.chr_cuencodigo = v_cuenta;
        update resumen set
            ingeros = v_ingeros,
            salidas = v_salidas
            where cuenta = v_cuenta;
        fetch c_cuentas into v_cuenta;
    end while;
    -- Consulta final
    select * from resumen;
    drop table resumen;
end //
```

delimiter ;

Por lo menos hasta esta versión (Versión 5), MySQL no tiene una manera elegante y fácil de controlar el recorrido por un cursor, debemos implementar un controlador de error, junto con una variable de control, tal como lo puede ver en el script.

A continuación tenemos la ejecución del script:

```

mysql> delimiter //

mysql> create procedure sp_resumen_cuentas()
-> begin
->     -- declaraciones
->     declare v_cuenta char(8);
->     declare v_ingeros decimal(12,2);
->     declare v_salidas decimal(12,2);
->     declare v_control_cursor int default 1;
->     declare c_cuentas cursor for
->         select cuenta from resumen;
->     declare continue handler for not found
```

```
->      set v_control_cursor = 0;
->      -- tabla temporal para construir el resumen
->      create temporary table resumen(
->          cuenta char(8) null,
->          saldo decimal(12,2) null,
->          ingresos decimal(12,2) null,
->          salidas decimal(12,2) null
->      );
->      -- inserto datos en la tabla resumen
->      insert into resumen(cuenta,saldo)
->          select chr_cuencodigo,dec_cuensaldo from cuenta;
->      -- Proceso
->      open c_cuentas;
->      fetch c_cuentas into v_cuenta;
->      while v_control_cursor = 1 do
->          select sum(m.dec_moviimporte) into v_ingresos
->              from movimiento as m
->              join tipomovimiento as t
->              on m.chr_tipocodigo = t.chr_tipocodigo
->              where t.vch_tipoaccion = 'INGRESO'
->              and m.chr_cuencodigo = v_cuenta;
->          select sum(m.dec_moviimporte) into v_salidas
->              from movimiento as m
->              join tipomovimiento as t
->              on m.chr_tipocodigo = t.chr_tipocodigo
->              where t.vch_tipoaccion = 'SALIDA'
->              and m.chr_cuencodigo = v_cuenta;
->          update resumen set
->              ingresos = v_ingresos,
->              salidas = v_salidas
->              where cuenta = v_cuenta;
->          fetch c_cuentas into v_cuenta;
->      end while;
->      -- Consulta final
->      select * from resumen;
->      drop table resumen;
-> end //
```

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

Primero cargamos las dos primeras columnas de la tabla resumen, para todas las cuentas.

El cursor permite hacer un recorrido cuenta x cuenta, y en cada iteración se calcula el total de ingresos y salidas para cada cuenta y se actualiza la tabla resumen.

Para finalizar se realiza un consulta a la tabla resumen y se elimina la tabla.

Finalmente, tenemos la ejecución del procedimiento:

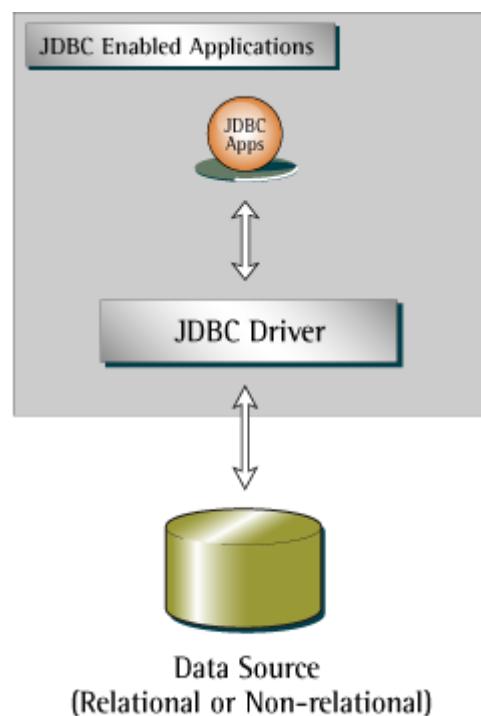
```
mysql> call sp_resumen_cuentas();
+-----+-----+-----+-----+
| cuenta | saldo | ingresos | salidas |
+-----+-----+-----+-----+
| 00100001 | 7100.00 | 9200.00 | 2100.00 |
| 00100002 | 4500.00 | 5500.00 | 1000.00 |
| 00200001 | 7000.00 | 25000.00 | 18000.00 |
| 00200002 | 6800.00 | 8000.00 | 1200.00 |
| 00200003 | 6000.00 | 7500.00 | 1500.00 |
+-----+-----+-----+-----+
```

5 rows in set (0.06 sec)

Query OK, 0 rows affected (0.08 sec)

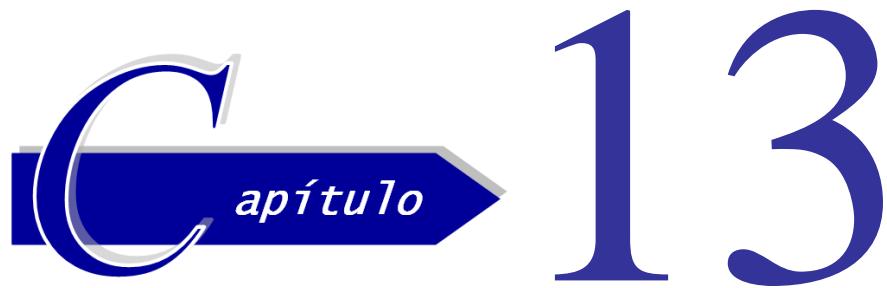
mysql>

Parte III



Programación con JDBC

Página en blanco.



13

Acceso a Bases de Datos

El API JDBC es la Interfaz de Programación de Aplicaciones (API) que proporciona el acceso universal a los datos para el lenguaje de programación Java. Incluye el API JDBC 1.0, que proporciona la funcionalidad básica para el acceso a los datos. El API JDBC 2.0 con funciones más avanzadas. El API JDBC 3.0 que proporciona una forma estándar de acceder a las últimas características objeto - relacionales que los sistemas de gestión de bases de datos hoy en día implementan. Además, el nuevo API incluye características como conjunto de resultados (`ResultSet`) scrollable y actualizables, y un mejor rendimiento. También extiende la tecnología JDBC más allá de cliente - servidor con pool de conexiones y transacciones distribuidas.

Temas a desarrollar:

- 13.1. Introducción
- 13.2. ¿Qué es el API JDBC?
- 13.3. Conexión a una Fuente de Datos

13.1. Introducción

El API Java Database Connectivity (JDBC) proporciona acceso universal a los datos desde el lenguaje de programación Java. Usando el API JDBC 3.0, puede acceder a prácticamente cualquier fuente de datos, desde bases de datos relacionales hasta hojas de cálculo y archivos planos. La tecnología JDBC también proporciona una base común sobre la cual las herramientas y las interfaces alternativas se pueden construir.

El API JDBC 3.0 consta de dos paquetes:

- El paquete `java.sql`
- El paquete `javax.sql`, que agrega capacidades del lado del servidor

Ambos paquetes están disponibles en la Plataforma Java Standard Edition (Java SE).

Para usar el API JDBC con un sistema de gestión de base de datos, necesita un controlador basado en la tecnología JDBC para mediar entre la tecnología JDBC y la base de datos. Dependiendo de varios factores, un controlador podría ser escrito solo en el lenguaje de programación Java, o en una mezcla del lenguaje de programación Java y métodos nativos Java Native Interface (JNI). El sitio Web de JDBC (<http://developers.sun.com/product/jdbc/drivers>) mantiene una lista de proveedores con los controladores actualmente disponibles o en fase de desarrollo.

El JDK incluye un controlador puente JDBC-ODBC que hace que la mayoría de controladores Open Database Connectivity (ODBC) estén disponibles para que los programadores puedan usarlos con la API JDBC. Tenga en cuenta que el controlador puente incluido en la Plataforma Java SE es apropiado sólo para uso experimental o cuando no se dispone de otro controlador.

13.2. ¿Qué es el API JDBC?

El API JDBC es una API de Java para acceder a cualquier fuente de datos tabular. El API JDBC consta de un conjunto de clases e interfaces escritas en el lenguaje Java que provee un API estándar para desarrolladores de aplicaciones y herramientas de base de datos.

El API JDBC facilita el envío de sentencias SQL a los sistemas de la base de datos relacional y soporta todos los dialectos del SQL. Pero el API JDBC 3.0 va más allá del SQL, también te permite interactuar con otros tipos de fuentes de datos, como archivos que contienen datos tabulares.

Lo relevante del API JDBC es que una aplicación puede acceder a cualquier fuente de datos y ejecutarse en cualquier plataforma con una Java Virtual Machine (JVM). En otras palabras, con el API JDBC no hay que escribir un programa para acceder a una base de datos Sybase, otro programa para acceder a una base de datos Oracle, otro programa para acceder a una base de datos DB2 de IBM, etcétera. Uno puede escribir un solo programa usando el API JDBC, y el programa podrá enviar una sentencia SQL u otras sentencias a la fuente de datos. Y, con una aplicación escrita en el lenguaje de programación Java, uno no tiene que preocuparse por escribir aplicaciones diferentes para plataformas diferentes. La combinación de la plataforma Java y el API JDBC permiten al programador escribir una vez y ejecutar la aplicación donde quiera.

El lenguaje de programación Java, siendo robusto, seguro, fácil de usar, fácil de entender, es un lenguaje base excelente para aplicaciones de bases de datos. Lo que se necesita es una manera de que las aplicaciones Java puedan comunicarse con una variedad de diferentes fuentes de datos. JDBC es el mecanismo para hacerlo.

El API JDBC 3.0 extiende lo que se puede hacer con la plataforma Java. Por ejemplo, la API JDBC hace posible la publicación de una página Web que contiene un Applet que utiliza la información obtenida de una fuente de datos remota. O una empresa puede utilizar el API JDBC para conectar todos sus empleados (incluso si están utilizando un conglomerado de Windows, Macintosh y UNIX) a una o más bases de datos internas a través de una Intranet. Con más y más programadores utilizando el lenguaje de programación Java, la necesidad de un fácil y universal acceso a los datos desde Java sigue creciendo.

13.2.1. ¿Qué hace el API JDBC?

En términos simples, un controlador basada en tecnología JDBC (controlador JDBC) permite hacer tres cosas:

1. Establecer una conexión con una fuente de datos.
2. Enviar consultas y sentencias de actualización a la fuente de datos.
3. Procesar los resultados

El siguiente fragmento de código proporciona un ejemplo simple de estos tres pasos:

```
String login = "root";
String password = "";
String url = "jdbc:mysql://localhost/eurekabank";

Class.forName("com.mysql.jdbc.Driver").newInstance();
```

```

cn = DriverManager.getConnection(url,login,password);
st = cn.createStatement();
rs = st.executeQuery("select * from empleado");
while(rs.next())
    System.out.println(rs.getString("vch_emplnombre"));

```

13.2.2. El API JDBC y ODBC frente a UDA

Previo al desarrollo de la API JDBC, Microsoft API ODBC (Open DataBase Connectivity) es el más ampliamente utilizado para acceder a la interfaz de programación de bases de datos relacionales. Ofrece la posibilidad de conectarse a casi todas las bases de datos en casi todas las plataformas. Entonces, ¿por qué no simplemente usar ODBC desde Java?

La respuesta es que sí es posible usar ODBC desde Java, pero esto es aconsejable hacerlo con la ayuda del API JDBC en forma de puente JDBC - ODBC. La pregunta ahora es, "¿Por qué se necesita el API JDBC?"

Hay varias respuestas a esta pregunta:

1. ODBC no es apropiado para usarlo directamente desde el lenguaje de programación Java, ya que utiliza la interfaz de C. Las llamadas desde Java a código nativo C tienen una serie de inconvenientes en la seguridad, la ejecución, la solidez, y portabilidad automática de las aplicaciones.
2. Una traducción literal del API ODBC de C en un API de Java no sería conveniente. Por ejemplo, Java no tiene punteros (variables de la dirección), y ODBC hace abundante uso de ellos. Usted puede pensar en JDBC como ODBC traducido en un interfaz mediante objetos de alto nivel que es natural para programadores que usan el lenguaje de programación Java.
3. ODBC es difícil de aprender. Se mezclan características simples y avanzadas, y tiene opciones complejas incluso para simples consultas. El API JDBC, por otra parte, fue diseñado para mantener sencillas las cosas sencillas, al tiempo que permite capacidades más avanzadas cuando sea necesario. El API JDBC es también más fácil de utilizar, simplemente porque se trata de un API de Java, lo que significa que un programador no tiene que preocuparse por la memoria, ya sea de gestión o de la alineación de bytes de datos.
4. Un API Java como JDBC, es necesario para implementar una solución "pure Java", es decir, una solución que sólo utiliza el API de Java. Cuando se utiliza ODBC, el gestor de controladores ODBC y el controlador deben ser instalados manualmente en cada máquina cliente. Cuando el controlador JDBC está escrito completamente en Java, el código JDBC es instalable automáticamente, portable, y seguro en todas las plataformas Java desde redes de computadoras hasta mainframes.

En resumen, el API JDBC es una interfaz Java natural para trabajar con SQL. Se basa en ODBC en vez de comenzar desde cero, de modo que los programadores familiarizados con ODBC lo encontrará muy fácil de aprender. El API JDBC conserva algunas de las características básicas de diseño de ODBC; de hecho, ambas interfaces están basadas en Open Group (antes X/Open) SQL CLI (Call Level Interface). La gran diferencia es que el API de JDBC aprovecha y refuerza el estilo y virtudes de Java, y va más allá de simplemente enviar sentencias SQL a un sistema de gestión de base de datos relacional.

Microsoft ha introducido nuevas APIs más allá de ODBC como OLE (Object Linking and Embedding) DB, ADO (ActiveX Data Objects) y RDS (Remote Data Service). En muchos aspectos estas API avanzan en la misma dirección que el API JDBC. Por ejemplo, OLE DB y ADO son también interfaces orientados a objetos para bases de datos que se pueden

utilizar para ejecutar comandos SQL. Sin embargo, OLE DB es una interfaz de bajo nivel diseñado para herramientas en lugar de desarrolladores. ADO es más parecido al API JDBC, pero no es Java puro. RDS proporciona funcionalidad similar a las facilidades del API JDBC RowSet, pero también RDS no está escrito en el lenguaje de programación Java, y no es portátil.

Más recientemente, Microsoft ha introducido UDA (Universal Data Access) como un término general que abarca OLE DB, ADO, RDS, y ODBC. El API JDBC 3.0 contiene todas las funcionalidades importantes de UDA más características que no se encuentran en UDA, como soporte SQL3.

13.2.3. Modelos de dos niveles y tres niveles

El API JDBC soporta ambos modelos de dos niveles y tres niveles de acceso a bases de datos.

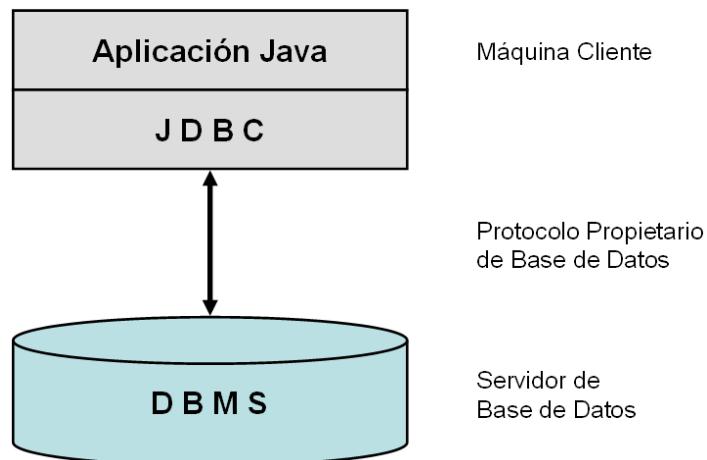


Figura 13 . 1 Arquitectura de dos niveles para el acceso a los datos.

En el modelo de dos niveles, un Applet de Java o una aplicación se comunican directamente con la fuente de datos. Esto requiere un controlador JDBC que pueda comunicarse con la fuente de datos particular. Las instrucciones de los usuarios son entregadas a la base de datos u otra fuente de datos, y los resultados son enviados de vuelta al usuario. La fuente de datos puede estar en otra máquina a la que el usuario está conectado a través de una red. Esto se conoce como configuración cliente/servidor, con la máquina del usuario como cliente, y la máquina que aloja la fuente de datos como servidor. La red puede ser una intranet que, por ejemplo, conecta a los empleados dentro de una empresa, o puede ser Internet.

En el modelo de tres niveles, las instrucciones son enviadas a un "**middle tier**" de servicios, que luego las envía a la fuente de datos. La fuente de datos procesa las instrucciones y envía los resultados al middle tier, que luego los envía al usuario. El modelo de tres niveles es muy atractivo porque el nivel intermedio posibilita mantener el control sobre el acceso y los tipos de actualizaciones que se pueden hacer sobre los datos corporativos. Otra ventaja es que simplifica el despliegue de las aplicaciones. Por último, en muchos casos, la arquitectura de tres niveles puede proporcionar mejoras en el rendimiento.

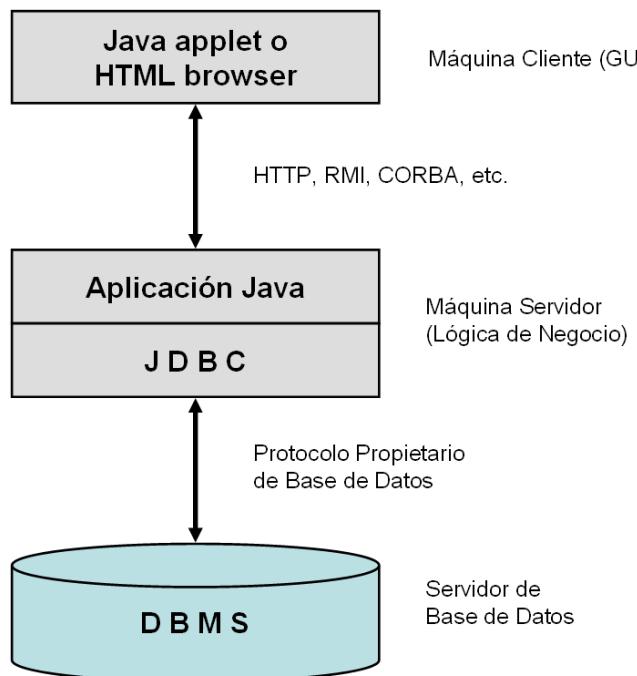


Figura 13 . 2 Arquitectura de tres niveles para el acceso a bases de datos.

Hasta hace poco, el middle tier generalmente ha sido escrito en lenguajes como C o C++, que ofrecen un mayor rendimiento. Sin embargo, con la introducción de compiladores que optimizan la traducción del bytecode de Java en un eficiente código específico de máquina y tecnologías tales como Enterprise JavaBeans, la plataforma Java se está convirtiendo rápidamente en la plataforma estándar para el desarrollo de middle tier. Esta es una gran ventaja, ya que permite aprovechar la robustez, el multi-hilo y características de seguridad propias de Java.

Con las empresas utilizando cada vez más el lenguaje de programación Java para escribir el código del servidor, el API JDBC se está utilizando cada vez más en el nivel intermedio de una arquitectura de tres niveles. Algunas de las características que hacen JDBC una tecnología de servidor son el soporte para la agrupación de conexiones (connection pooling), transacciones distribuidas y conjunto de resultados (rowsets) desconectado. Y, por supuesto, la API JDBC es lo que permite el acceso a una fuente de datos desde un middle tier de Java.

13.2.4. Conformidad de SQL

SQL es el lenguaje estándar de acceso a bases de datos relacionales. Lamentablemente, SQL no es aún el estándar que se quisiera.

Un área de dificultad son los tipos de datos utilizados por diferentes DBMSs (DataBase Management Systems) que algunas veces varían, y las variaciones pueden ser importantes. JDBC aborda esto definiendo un conjunto de tipos de identificadores genéricos SQL en la clase `java.sql.Types`. Tenga en cuenta que, los términos "JDBC SQL type", "JDBC type", y "SQL type" son intercambiables y se refieren a los tipos de identificadores genéricos SQL definidos en el `java.sql.Types`.

Otra área de dificultad con SQL es que aunque la mayoría de DBMSs usan el estándar del SQL para la funcionalidad básica, no se ajustan al estándar más reciente que define la semántica o sintaxis SQL para funcionalidades más avanzadas. Por ejemplo, no todas las bases de datos soportan procedimientos almacenados o combinaciones externas (outer joins), y las que lo hacen no siempre son coherentes entre sí. Asimismo, soporte para características SQL3 y tipos de datos varían mucho. Es de esperar que la parte de SQL,

que es verdaderamente estándar, se ampliará para incluir más y más funcionalidades. En el interín, sin embargo, la API JDBC debe apoyar a SQL como es.

Una forma en que la API JDBC trata este problema es permitiendo cualquier cadena de consulta a través del controlador DBMS subyacente. Esto significa que una aplicación es libre de utilizar tanta funcionalidad SQL como lo desee, pero se corre el riesgo de recibir un error en algunas bases de datos.

De hecho, la consulta de una aplicación puede ser otra cosa que no sea SQL, o puede ser un derivado especializado de SQL diseñado para un DBMS específico (por ejemplo, para consultas de documentos o imágenes).

Una segunda estructura de JDBC se ocupa de los problemas de conformidad de SQL para proporcionar estilo de cláusulas de escape ODBC. La sintaxis de escape proporciona una sintaxis JDBC estándar para varias de las áreas más comunes del SQL divergentes. Por ejemplo, hay escapes para fecha literales y llamadas a procedimientos almacenados.

Para aplicaciones complejas, JDBC trata la conformidad de SQL en una tercera parte. Proporciona información descriptiva sobre el DBMS por medio de la interfaz `DatabaseMetaData`, de tal manera que las aplicaciones pueden adaptarse a los requerimientos y capacidades de cada DBMS. Sin embargo, los usuarios finales no tienen por qué preocuparse por los metadatos.

Debido a que la API JDBC se utiliza como API base para el desarrollo de herramientas de acceso a bases de datos y otras APIs, también tienen que abordar el problema de conformidad para cualquier cosa construida sobre el mismo. Un controlador JDBC debe soportar por lo menos ANSI SQL-92 Entry Level. (ANSI SQL-92 se refiere a las normas adoptadas por el American National Standards Institute en 1992. Entry Level hace referencia a una lista específica de las capacidades de SQL.) Tenga en cuenta, sin embargo, que aunque el API JDBC 3.0 incluye soporte para SQL3 y SQLJ, los controladores JDBC no están obligados a soportarlos.

Debido a la amplia aceptación de la API JDBC por los proveedores de bases de datos, proveedores de conectividad, proveedores de servicios de Internet, y creadores de aplicaciones, se ha convertido en el estándar para el acceso a los datos desde el lenguaje de programación Java.

13.2.5. Productos JDBC

La API JDBC es una elección natural para los desarrolladores que utilizan la plataforma Java, ya que ofrece fácil acceso a bases de datos para las aplicaciones y applets Java.

En la actualidad existe una serie de productos basados en JDBC que ya han sido desplegados o están en desarrollo. La situación de estos productos cambia con frecuencia, por lo que es necesario consultar la página Web de JDBC para obtener información más reciente. Lo puede encontrar en la siguiente URL:

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

13.2.6. Estructura de JDBC

Java proporciona tres componentes JDBC:

- El administrador de controladores JDBC
- La suite de testeo de controladores JDBC
- El puente JDBC-ODBC

La clase `DriverManager` (Gestor de Controladores) ha sido tradicionalmente la columna vertebral de la arquitectura JDBC. Se encuentra representado por la clase

`java.sql.DriverManager`. Es bastante pequeño y simple; su función fundamental es conectar aplicaciones Java con el controlador JDBC correcto, y luego soltarlo.

Se puede considerar que JDBC ofrece dos conjuntos de clases e interfaces bien diferenciados, aquellas de más alto nivel que serán utilizadas por los programadores de aplicaciones para el acceso a bases de datos, y otras de más bajo nivel enfocadas hacia los programadores de controladores (drivers) que permiten la conexión a una base de datos. En el presente libro nos vamos a centrar en el primer subconjunto, el de más alto nivel.

13.2.7. Tipos de Controladores JDBC

Los controladores JDBC nos permiten conectarnos con una base de datos determinada. Existen cuatro tipos de controladores, cada uno presenta una filosofía de trabajo diferente.

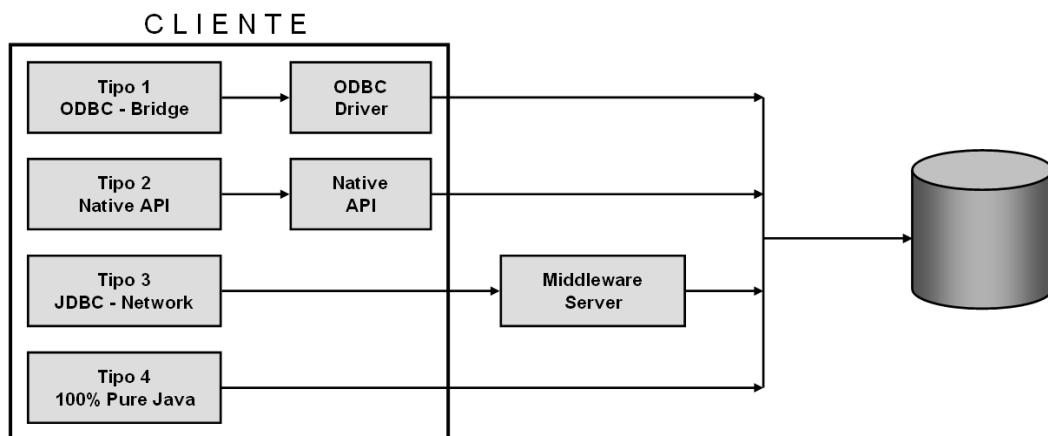


Figura 13 . 3 Tipos de controladores JDBC

Una aplicación desarrollada con Java usa la clase del API JDBC `DriverManager` para acceder a un controlador basado en la tecnología JDBC ("JDBC Driver"). Hay varias alternativas de controladores JDBC. Un controlador JDBC-Net se conecta a un servidor de la capa intermedia y traduce a las llamadas JDBC a un protocolo DBMS - Net independiente. Luego el servidor traduce esto a un protocolo DBMS. Otras implementaciones del controlador se conectan directamente a un DBMS usando protocolos que acceden a las propiedades de la base de datos.

A continuación se pasa a describir cada uno de los tipos de conexión.

13.2.7.1. Tipo 1: JDBC-ODBC bridge plus ODBC driver

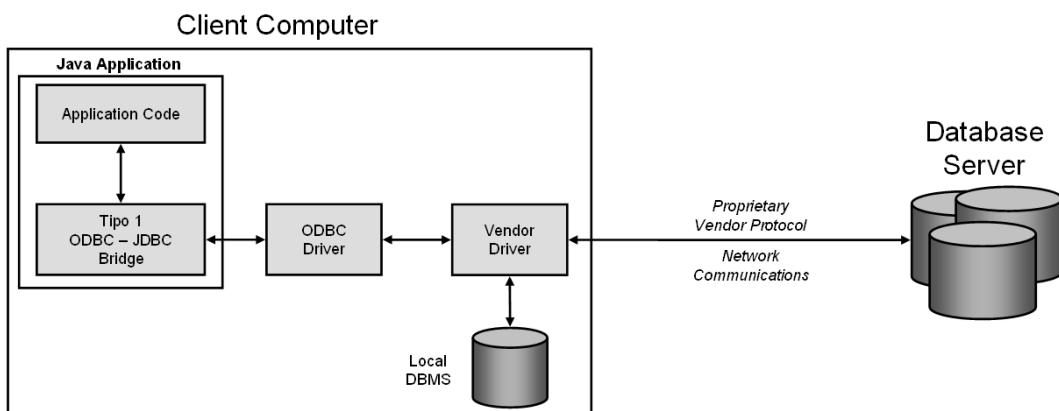


Figura 13 . 4 JDBC Type 1 driver: JDBC-ODBC bridge

Permite a las aplicaciones acceder a fuentes de datos ODBC existentes mediante JDBC. El puente JDBC-ODBC implementa operaciones JDBC traduciéndolas a operaciones ODBC;

se encuentra dentro del paquete `sun.jdbc.odbc` (que se encuentra incluido dentro del JDK a partir de la versión 1.1) y contiene librerías nativas para acceder a ODBC.

Se debe señalar que en cada máquina cliente que utilice el controlador es necesario una configuración previa, es decir, debemos definir la fuente de datos utilizando para ello el administrador de controladores ODBC que lo podemos encontrar dentro del Panel de Control de Windows. Debido a esta configuración en las máquinas clientes, este tipo de controlador no es adecuado para utilizarlo dentro de applets, su utilización está más encaminada a aplicaciones Java dentro de pequeñas intranets en las que la instalaciones en los clientes no sean un gran problema o para aplicaciones Java que pueden actuar de capa intermedia dentro de un modelo de tres capas, como se veía anteriormente.

Además debido a que el puente JDBC-ODBC contiene parte de código nativo, no es posible utilizarlos dentro de applets, debido a las restricciones de seguridad de éstos. Desde Javasoft (una división de Sun) nos sugieren que el uso del puente JDBC-ODBC se limite al tiempo que tarden en aparecer controladores JDBC de otro tipo para la base de datos a la que queremos acceder, esta advertencia es debido a que muchos controladores ODBC no están programados teniendo en cuenta la programación multiproceso (multithreading), por lo que se pueden producir problemas a la hora de utilizar el puente JDBC-ODBC.

13.2.1.2. Tipo 2: Native-API partly-Java driver

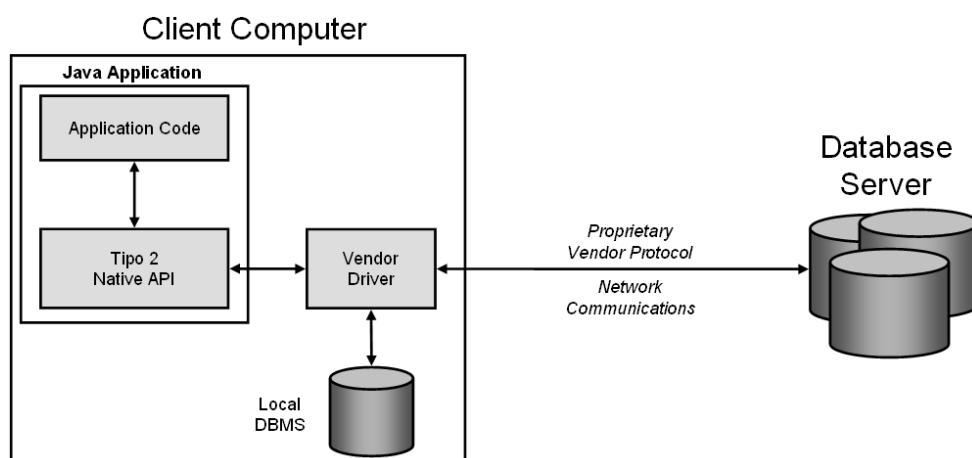


Figura 13 . 5 JDBC Type 2 driver: Native API

Son similares a los controladores de Tipo 1, en cuanto también necesitan una configuración en la máquina cliente. Este tipo de controlador convierte llamadas JDBC a llamadas de Oracle, Sybase, Informix, DB2 u otros Sistemas Gestores de Bases de Datos (SGBD). Tampoco se pueden utilizar dentro de applets al poseer código nativo.

13.2.7.3. Tipo 3: 100% Pure Java, JDBC – Network

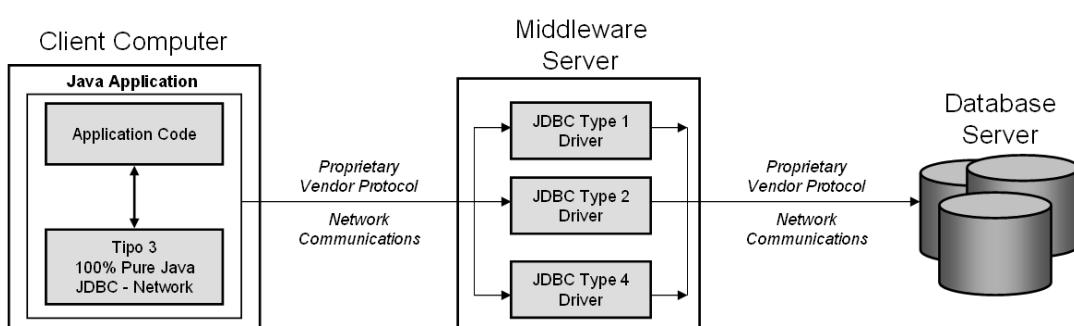


Figura 13 . 6 JDBC Type 3 driver: 100% Pure Java, JDBC - Network

Los controladores Tipo 3 usan una tercera capa para acceder a la bases de datos. Las soluciones J2EE a menudo implementan esta arquitectura. Los clientes usan conectores estándar de red para comunicarse con una aplicación servidor. La información del conector es entonces traducida por la aplicación servidor al formato de llamada requerido por los DBMS, y reenviada al servidor de la base de datos. La Figura 13.6 muestra una configuración típica de tres capas que usa un controlador Tipo 3 para la conectividad a la base de datos.

Se puede pensar en la aplicación servidor como un "proxy" JDBC, quiere decir que éste realiza las llamadas por la aplicación cliente. Como consecuencia, usted necesita algún conocimiento de configuración de la aplicación servidor para un uso eficaz de este tipo de controlador. Por ejemplo, su aplicación servidor podría usar a un controlador Tipo 1, 2, ó 4 para comunicarse con la base de datos, entender estos matices resultará de mucha utilidad.

No obstante, esta configuración puede resultar más flexible. Los proveedores de la aplicación servidor con frecuencia proveen soporte para múltiples bases de datos en el back-end. Esta característica la podemos utilizar para escribir un código base que soporte varias bases de datos de diferentes proveedores. La capa intermedia manejará la sintaxis SQL y los tipos de datos que existen entre bases de datos.

El despliegue del controlador Tipo 3 es más fácil que los Tipos 1 ó 2 porque el cliente no necesita la instalación de algún software adicional, aparte de su aplicación, para acceder a la base de datos.

13.2.7.4. Tipo 4: 100% Java

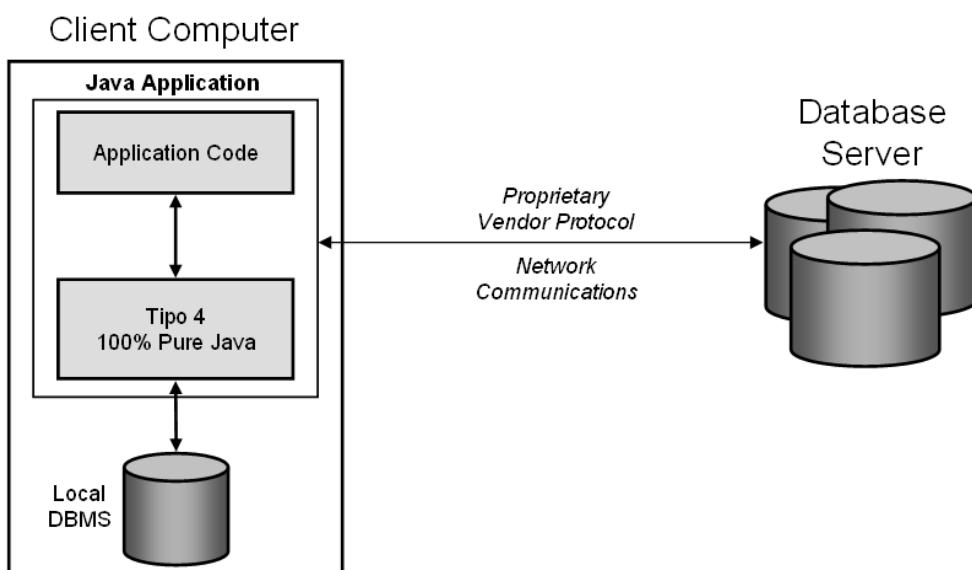


Figura 13 . 7 JDBC Type 4 driver: 100% Java

Este tipo de controlador convierte directamente llamadas JDBC en el protocolo de la red usado por el DBMS. Esto permite llamadas directas desde la máquina cliente al servidor DBMS. Son como los controladores tipo 3 pero sin la figura del middleware y tampoco requieren ninguna configuración en la máquina cliente. La complejidad del programa intermedio es eliminada, pero en el caso de que estemos utilizando el driver de tipo 4 en un applet, debido a las restricciones de seguridad de los applets, la base de datos y el applet se deben encontrar en el mismo servidor, ya que como ya sabíamos un applet solo puede conectarse al servidor del que proviene. Los controladores de tipo 4 se pueden utilizar para servidores Web de tamaño pequeño y medio, así como para intranets.

Estos controladores utilizan protocolos propietarios, por lo tanto serán los propios fabricantes de bases de datos los que ofrecerán estos controladores. También existen controladores de la comunidad de Java que no necesitan ninguna licencia para ser

utilizados. Empresas como **i-net Software** desarrollan controladores JDBC para las bases de datos más conocidas; pueden descargar la versión de evaluación de la siguiente dirección <http://www.inetsoftware.de>.

JavaSoft recomienda el uso de controladores tipo 3 y 4 para acceder a bases de datos a través de JDBC. Los tipos 1 y 2 deberán ser soluciones provisionales hasta que aparezca algún controlador del tipo 3 o 4. Los tipos 3 y 4 ofrecen todas las ventajas de Java, incluyendo la instalación automática; por ejemplo, cargando el controlador JDBC cuando un applet necesite utilizarlo.

Existe un gran número de controladores disponibles, puede consultar en la dirección <http://industry.java.sun.com/products/jdbc/drivers>, en este sitio se ofrece un formulario de búsqueda que permite encontrar el tipo de controlador que estamos buscando junto con el sistema gestor de base de datos al que queremos acceder y la versión de JDBC que implementa el controlador. Este punto es importante, debemos estar seguros que el controlador que vamos a utilizar soporte la versión de JDBC, cada vez existen más controladores disponibles que soportan la última versión de JDBC.

Tabla 13 . 1 Tipos de Controladores JDBC

Tipo Driver	Ventajas	Desventajas
Type 1: JDBC-ODBC bridges	Puede acceder a bases de datos locales como Microsoft Access y FoxPro. Útil para probar características básicas de JDBC en computadoras Windows stand alone.	El usuario debe establecer y mantener fuentes de datos ODBC. Lento. La capa de ODBC extra hace necesario un procesamiento adicional. Puede requerir software adicional del cliente como componentes de conectividad de red de base de datos. No utilizable cuando el despliegue requiere descarga automática y configuración de aplicaciones.
Tipo 2: Native API	Más rápido que el Tipo 1 porque se suprime la capa de ODBC. Código nativo optimizado para su plataforma y DBMS	Un API del proveedor específico debe ser instalado sobre el ordenador de cliente. No utilizable cuando el despliegue requiere descarga automática y configuración de aplicaciones.
Tipo 3: JDBC-Network	No requiere software cliente adicional. El servidor de aplicaciones puede proveer acceso a múltiples DBMS.	Se requiere un servidor de aplicaciones o middleware. Puede requerir configuración adicional para el uso de Internet.
Tipo 4: 100% Java	No requiere software cliente adicional. Comunicación directa con el servidor de base de datos.	Puede requerir configuración adicional para el uso de Internet.

13.3. Conexión a una Fuente de Datos

En esta sección explico como usar los controladores JDBC en sus aplicaciones. Antes de empezar, es necesario determinar el nombre exacto de la clase para el controlador de su proveedor de datos. Típicamente, los proveedores siguen la convención de nombramiento de paquete Java al nombrar a sus controladores. Por ejemplo, el controlador JDBC de Oracle es: `oracle.jdbc.driver.OracleDriver`.

13.3.1. Driver y DriverManager

La interfaz `java.sql.Driver` y la clase `java.sql.DriverManager` proveen las herramientas para trabajar con controladores JDBC. La Figura 13.8 muestra un diagrama de clases UML que ilustra la relación entre la clase `DriverManager` y la interfaz `Driver`.

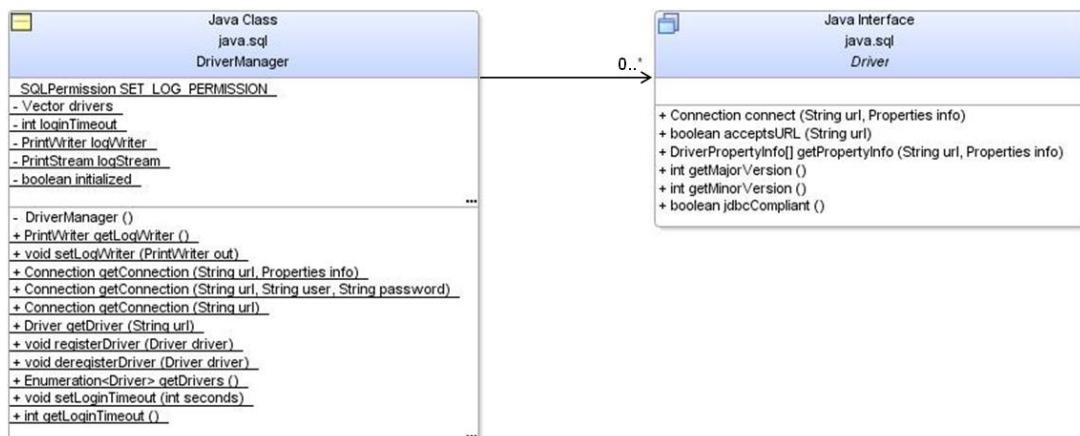


Figura 13 . 8 Diagrama de Clases UML de Driver y DriverManager

Todos los controladores JDBC deben implementar la interfaz `Driver`. En la práctica, sin embargo, usted puede encontrar que trabajar directamente con un objeto tipo `Driver` no es muy útil. Un objeto tipo `Driver` provee un método para obtener conexiones, pero `DriverManager` ofrece alternativas más flexibles de conexión.

Además, un objeto `DriverManager` maneja los controladores JDBC por usted. Con esta clase usted puede explícitamente registrar, seleccionar, o remover cualquier controlador JDBC de su aplicación. Esto le da la flexibilidad para almacenar programáticamente varios controladores y seleccionar el que usted necesita en tiempo de ejecución. El patrón de diseño `Factory` se beneficia de esta característica.

13.3.1.1. Registrar Controladores JDBC

Para usar un controlador JDBC primero debe registrarlo con el objeto `DriverManager`, el cual tiene un método para registrar el controlador. Sin embargo, dos técnicas adicionales están disponibles para registrar los controladores JDBC. La siguiente es una lista de las tres formas diferentes para registrar a un controlador JDBC:

- `Class.forName(String driverName).newInstance()`
- `DriverManager.registerDriver(Driver driverName)`
- `jdbc.drivers` property

`Class.forName(String driverName).newInstance()` es la forma más común para registrar un controlador JDBC. Comparado con la instanciación de un nuevo objeto de tipo JDBC `Driver`, también permite al objeto registrarse por si mismo con `DriverManager`. ¿Cómo puede esto ocurrir cuando `DriverManager` no está involucrado? La especificación JDBC requiere objetos de tipo `Driver` para registrarse con `DriverManager` por un inicializador estático que llama el método `DriverManager.registerDriver ()`. Este mecanismo permite registrar controladores dinámicamente en tiempo de ejecución sin tener en cuenta como usted registra el controlador. El siguiente fragmento de código provee una plantilla para usar `Class.forName().newInstance()`:

```
try {
```

```
        Class.forName("com.mysql.jdbc.Driver").newInstance();  
  
    } catch (Exception ex) {  
  
        ex.printStackTrace();  
  
    }  

```

El método `Class.forName()` provee bastante flexibilidad porque toma un objeto de tipo `String` como parámetro. Se puede definir el parámetro en tiempo de ejecución usando una variedad de métodos. Una técnica es suministrar el nombre del controlador por la línea de comando cuando la aplicación inicia su ejecución. Otro es leerlo desde un archivo de propiedades. Cualquier método le permite cambiar el controlador en el inicio de la aplicación.

Nota:

Para asegurar que su controlador está correctamente registrado con `DriverManager`, siempre use el método `Class.forName().newInstance()`. Esto asegura que el objeto `Driver` es instanciado y el inicializador estático requerido es llamado.

Como usted ya sabe, `DriverManager` provee el método `DriverManager.registerDriver` (`Driver driverName`) para registrar controladores JDBC. El siguiente fragmento de código le muestra qué tan sencillo es este método; note que es necesario crear una nueva instancia de tipo `Driver` y pasarlo como parámetro:

```
try {  
  
    DriverManager.registerDriver(new com.mysql.jdbc.Driver());  
  
} catch(SQLException e) {  
  
    e.printStackTrace();  
  
}  

```

Nota:

El método `DriverManager.registerDriver()` limita la flexibilidad durante el tiempo de ejecución porque toma un objeto de tipo `Driver` como parámetro. `Class.forName().newInstance()` acepta un valor de tipo `String` que se puede obtener de un parámetro de línea de comando o un archivo de propiedades.

Estableciendo los valores del sistema de propiedades de `jdbc.drivers.property` se utiliza fundamentalmente para hacer pruebas. Se puede usar la opción `-D` en el inicio para especificar un controlador válido al ejecutar la aplicación, lo que estará automáticamente registrado con `DriverManager`. Esta técnica también puede resultar útil en producción. Por ejemplo; en sistemas operativos UNIX se puede controlar el acceso a las bases de datos estableciendo variables de entorno para controladores específicos basados en el usuario que ha iniciado sesión. Esto permite establecer bases de datos específicas para usuarios

individuales. También se puede establecer este valor en un archivo de propiedades de un cliente. Lo negativo de esta técnica es que usted debe controlar el ambiente de su cliente. Aquí tenemos un ejemplo de cómo usar `jdbc.drivers.property` en el inicio de una aplicación:

```
D:>java -cp -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver MyJDBCProg
```

Nota:

Recuerde que es necesario ubicar el archivo `.jar` que contiene el controlador en el `CLASSPATH`. Tener el descuido de no hacer esto causará errores de recopilación y ejecución.

13.3.1.2. Seleccionando y Desregistrando Controladores

Durante la ejecución de su aplicación puede que necesite seleccionar o quitar un controlador específico de la lista de `DriverManager`. Estos métodos son realmente útiles; por ejemplo, se podría implementar el patrón de diseño `Factory` para que soporte múltiples bases de datos.

Dentro de su clase `Factory` puede usar `DriverManager` para pre-registrar los controladores JDBC que necesita para conectarse a las bases de datos soportadas. Cuando ocurra el requerimiento de un objeto `Connection`, su fábrica selecciona el controlador correcto de `DriverManager` y luego crea el objeto `Connection`. De esta manera es posible soportar múltiples bases de datos y puede proveer el objeto `Connection` correcto al cliente basado en su requerimiento.

Los dos métodos para seleccionar controladores son `DriverManager.getDriver()` y `DriverManager.getDrivers()`. El primero devuelve un objeto de tipo `Driver` específico cuando se le pasa la URL JDBC identificando su base de datos. (En la siguiente sección se desarrolla el tema de URLs JDBC.) El segundo devuelve un objeto de tipo `Enumeration` a fin de que se pueda iterar a través de una lista de objetos de tipo `Driver` hasta que encuentre el que necesite. El siguiente fragmento de código provee un ejemplo de cómo usar el método `DriverManager.getDrivers()`:

```
// Se necesita un objeto Enumeration para obtener los controladores registrados
Enumeration driverEnum = DriverManager.getDrivers();

while(driverEnum.hasMoreElements()) {
    // Captura la referencia del objeto Driver
    Driver driver = (Driver)driverEnum.nextElement();

    // Obtiene e imprime el nombre del controlador
    String str = "El Nombre del controlador es: " + driver.getClass().getName();
    System.out.println(str);
}
```

Aunque en el fragmento de código solo se realiza un listado del nombre de los controladores, usted puede usar el objeto `Driver` recuperado desde el objeto `Enumeration` para conectarse a una base de datos o usar el controlador con `DriverManager.registerDriver()` explícitamente para removérselo de la lista interna del objeto `DriverManager`.

Si por alguna razón necesita quitar el soporte para una base de datos, entonces puede invocar el método `DriverManager.deregisterDriver()` para quitar a un controlador registrado en `DriverManager`. El método es similar al método `registerDriver()` en que toma un objeto de tipo `Driver` como un parámetro.

13.3.2. Trabajando con Objetos Connection

En JDBC, una instancia de la clase `Connection` es una conexión física con una base de datos. El método `Driver.connect()` realiza la conexión con la base de datos y provee un objeto `Connection`. Es posible usar el objeto de tipo `Driver` directamente, pero `DriverManager` envuelve la llamada en su método `getConnección()`.

Por varias razones, utilizar `DriverManager` es la forma preferida para abrir conexiones con base de datos. Para nombrar una, si usted ha registrado varios controladores, `DriverManager` determinará el controlador apropiado con el cual relacionarse. Además, el método `getConnección()` está sobrecargado para proveerle una variedad de formas de abrir conexiones.

El diagrama de clases UML en la Figura 13.8 muestra que cada método `DriverManager.getConnección()` tiene al parámetro común de tipo `String`, `url`. Este valor es llamado un JDBC URL y tiene un significado especial.

13.3.2.1. Entendiendo JDBC URLs

JDBC requiere un sistema de nombramiento para poder realizar la conexión a una base de datos. El JDBC URL provee este sistema.

A continuación tenemos su estructura general:

```
jdbc:<subprotocol>:<subname>
```

Al crear un JDBC URL es necesario establecer valores para `<subprotocol>` y `<subname>`. El valor de `<subprotocol>` indica cuál es el protocolo específico del proveedor para usarlo cuando nos conectamos a la base de datos. Algunos proveedores DBMS usan múltiples protocolos propietarios para comunicarse con el servidor de base de datos.

El valor de `<subname>` indica la fuente de datos o base de datos con la que queremos conectarnos. Algunos servidores pueden contener más que una base de datos y pueden usar nombres lógicos para representar a cada una de ellas. En general, el valor de `<subname>` es el nombre lógico de la base de datos en el servidor de la base de datos.

Nota:

Los valores exactos de `<subprotocol>` y `<subname>` para el JDBC URL dependen del controlador. Los controladores del mismo proveedor pueden tener diferentes sub-protocolos. No hay formato estándar para cada parámetro.
Debe consultar la documentación específica del proveedor para el formato correcto.

Los siguientes dos ejemplos pueden ayudarle a entender el JDBC URL.

Ejemplo de JDBC-ODBC

```
String url = "jdbc:odbc:MyDB";
```

En este ejemplo, el valor de `<subprotocol>` es `odbc` y queremos conectarnos con un ODBC DSN llamado `MyDB`, que es el valor para `<subname>`. Recuerde que cuando usa ODBC es

responsable de configurar correctamente el DSN en la computadora del cliente antes de usar su aplicación.

Ejemplo de Oracle

```
String url = "jdbc:oracle:thin:@dbServerName:1521:ORCL";
```

En este ejemplo, el valor de <subprotocol> es oracle:thin. La palabra oracle es un estándar en los controladores de Oracle, por su parte, thin se refiere al mecanismo de conexión específico que Oracle utiliza. Algunos proveedores pueden encapsular muchos protocolos diferentes de red para conectarse a sus bases de datos en el controlador. Esto es lo que hace Oracle con su controlador tipo 4. Finalmente, el valor de <subname>, @dbServerName:1521:ORCL, dice al controlador de Oracle el host, port, y la instancia de la base de datos con la que se realizará la conexión.

Nota

El valor para <subprotocol> es único. Los proveedores de controladores deben registrar el nombre de su <subprotocol> con Sun Microsystems, que actúa como un registrador informal.

13.3.2.2. Abriendo Conexiones

Ahora, retomaremos nuevamente los métodos `DriverManager.getConnection()`. Para una fácil referencia, a continuación tenemos los tres métodos sobrecargados de `DriverManager.getConnection()`:

- `getConnection(String url)`
- `getConnection(String url, Properties prop)`
- `getConnection(String url, String user, String password)`

El primer método requiere un parámetro, un JDBC URL, para realizar una conexión. Note que este método carece de información de seguridad. Este método se debe utilizar si la base de datos no ofrece servicios de autenticación de usuario. Sin embargo, algunas bases de datos asumirán que si el usuario está registrado en el cliente, entonces tendrá derechos para usar el sistema.

El siguiente método toma un parámetro adicional, un objeto de tipo `Properties`, además de la cadena `url`. Use este método si necesita pasar información específica a la base de datos cuando realice la conexión. Establezca los pares nombre – valor apropiados en el objeto y suminístrela como un parámetro cuando invoca el método.

El último método, el más común, es bastante sencillo. Requiere de tres cadenas, la `url`, el usuario, y la contraseña.

Cuando llamamos al método `getConnection()`, `DriverManager` devuelve un objeto de tipo `Connection` válido. Internamente, `DriverManager` pasa sus parámetros a cada objeto de tipo `Driver` registrado en su lista. `DriverManager` inicia con el primer controlador JDBC y trata de hacer una conexión; si esto falla, `DriverManager` intenta con el siguiente. Este proceso se repite hasta que encuentre un controlador que puede conectarse a la base de datos especificada en la JDBC URL.

Usar el método `getConnection()` es sencillo. A continuación tenemos un formato básico de cómo usar `DriverManager` para abrir una conexión a una base de datos Oracle:

```
String url = "jdbc:oracle:thin:@egcc:1521:XE";
String user = "gustavo";
String pwd = "cerebro";
Connection conn = DriverManager.getConnection(url, user, pwd);
```

Si la conexión falla, `DriverManager` lanza una excepción de tipo `SQLException` que contiene el mensaje de error específico de la base de datos. Por consiguiente, es de mucha ayuda saber algo acerca de la semántica involucrada en la conexión con su base de datos y así poder interpretar los mensajes de error.

Es posible reemplazar este comportamiento por la recuperación del controlador particular que se requiere de la lista del objeto `DriverManager`. (La sección anterior explica como hacerlo.) Una vez que se obtiene el controlador correcto, solamente invoca su método `connect()` para obtener una conexión.

Nota

A medida que J2EE gana más popularidad en aplicaciones empresariales, el uso de la interfaz `DataSource` se convierte en el modo preferido de abrir conexiones a bases de datos. Por lo general la interfaz `DataSource` trabaja conjuntamente con Java Naming and Directory Interface (JNDI) para proporcionar una conexión. Los objetos `DataSource` no requieren que se suministre el nombre del controlador para hacer la conexión. Esto también permite manejar pool de conexiones y transacciones distribuidas.

De modo parecido, podemos instanciar directamente la clase `Driver` y usarlo para hacer la conexión. El siguiente fragmento de código muestra como hacer la conexión que usa una instancia recién creada de la clase `Driver`:

```
Driver drv = new oracle.jdbc.driver.OracleDriver();
Properties prop = new Properties();
prop.put("user", "gustavo");
prop.put("password", "cerebro");
Connection conn = drv.connect("jdbc:oracle:thin:@egcc:1521:XE", prop);
```

La variable `prop` es usada para mantener un conjunto de parámetros, como el usuario y la contraseña, que se necesita para realizar la conexión a la base de datos, tal como se ilustra en el fragmento de código.

Ejemplo 13 . 1

Este ejemplo, muestra una aplicación que registra un controlador JDBC-ODBC y obtiene una conexión a un DSN local usando cada uno de los tres métodos `getConnection()`.

Para este ejemplo es necesario crear el DSN de nombre `DSNGustavo`, usted puede reemplazarlo por otro, puede ser para cualquier base de datos, ya que se trata de probar solo la conexión.

```
package cap13;

import java.sql.*;
import java.util.Properties;

public class Ejemplo1301 {

    public static void main(String[] args) {

        // Definimos 3 variables de tipo Connection
        Connection conn1 = null;
        Connection conn2 = null;
        Connection conn3 = null;

        // Iniciamos un manejador de errores estándar
        try {
```

```
// Cargamos el driver con Class.forName()
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

//Definimos JDBC URL, usuario, y contraseña
String jdbcUrl = "jdbc:odbc:DSNGustavo";
String user = "sistemas";
String pwd = "pacherrez";
System.out.println("Abriendo las conexiones de base de datos...");

// Uso del primer método getConnection usando solo una JDBC URL
conn1 = DriverManager.getConnection(jdbcUrl + ";UID=" + user + ";PWD=" + pwd);

// Comprobar si la conexión es satisfactoria
if (conn1 != null) {
    System.out.println("Conexión 1 satisfactoria!!!");
}

// Uso del segundo método getConnection. Este requiere
// de un objeto Properties para guardar información específica
Properties prop = new Properties();
prop.put("user", user);
prop.put("password", pwd);
conn2 = DriverManager.getConnection(jdbcUrl, prop);

// Comprobar si la conexión es satisfactoria
if (conn2 != null) {
    System.out.println("Conexión 2 satisfactoria!!!");
}

// Uso del tercer método getConnection que requiere tres
// parámetros String; JDBC URL, usuario, y contraseña
conn3 = DriverManager.getConnection(jdbcUrl, user, pwd);

// Comprobar si la conexión es satisfactoria
if (conn3 != null) {
    System.out.println("Conexión 3 satisfactoria!!!");
}
```

```

// Cerrar explícitamente todas las conexiones. SIEMPRE HACER ESTO!!!
conn1.close();
conn2.close();
conn3.close();

// Comprobar si las conexiones están cerradas
System.out.println("Cerrando todas las conexiones...");
if (conn1.isClosed()) {
    System.out.println("Conexión 1 está cerrada");
}
if (conn2.isClosed()) {
    System.out.println("Conexión 2 está cerrada");
}
if (conn3.isClosed()) {
    System.out.println("Conexión 3 está cerrada");
}

} catch (SQLException se) {
    // Manejo de errores para DriverManager
    se.printStackTrace();
} catch (Exception e) {
    // Manejo de errores para Class.forName y todas las otras excepciones
    e.printStackTrace();
} finally {
    // Bloque finally
    try {
        if (conn1 != null) {
            conn1.close();
        }
        if (conn2 != null) {
            conn2.close();
        }
        if (conn3 != null) {
            conn3.close();
        }
    } catch (SQLException se) {
        se.printStackTrace();
    } // Fin dtry en finally
}

```

```
    } // Fin de try

}

} // Fin de método main

} // Fin de la clase Ejemplo1301
```

El resultado de su ejecución es:

```
Abriendo las conexiones de base de datos...
Conexión 1 satisfactoria!!!
Conexión 2 satisfactoria!!!
Conexión 3 satisfactoria!!!
Cerrando todas las conexiones...
Conexión 1 está cerrada
Conexión 2 está cerrada
Conexión 3 está cerrada
```

13.3.2.3. Cerrando las Conexiones de Base de Datos

En el Listado del ejemplo 13.1 se cierran explícitamente todas las conexiones a la base de datos para terminar cada sesión de base de datos. Sin embargo, si no lo hace explícitamente, el recolector de basura (garbage collector) de Java cerrará la conexión cuando elimina los objetos que no se están usando.

Confiar en el recolector de basura, especialmente en programas de base de datos, es una práctica de programación muy pobre. Se debería tener como hábito cerrar siempre la conexión de manera explícita con el método `close()` para un par de razones.

- Primero, asegurarse que su sesión está correctamente en el servidor de base de datos. Algunas bases de datos se comportan de manera irregular cuando una sesión de usuario finaliza incorrectamente.
- Segundo, el DBMS puede asumir que la sesión de usuario sufrió una interrupción y cancela (rollback) cualquier cambio realizado por el programa. Por ejemplo, las bases de datos Oracle cancelarán todas las sentencias que no han sido confirmadas si la sesión de usuario finaliza en medio de una transacción. Cerrar explícitamente la conexión asegura que la base de datos limpia el entorno del cliente en el lado del servidor de forma normal, como debe ser.

Cerrando explícitamente una conexión se liberan recursos del DBMS, que harán a su administrador de base de datos feliz. Por ejemplo, si su licenciamiento de base de datos está basado en sesiones abiertas, y no cierra una sesión puede impedir a otros usar el sistema. También, cada conexión abierta requiere alguna cantidad de RAM y procesamiento del CPU. Las conexiones abiertas innecesariamente consumen recursos del cliente y del servidor de base de datos. Cuando usted cierra conexiones correctamente usted libera estos recursos para que puedan ser usados por otros usuarios.

Para asegurar que una conexión está cerrada, podemos incluir un bloque `finally` en su código. Un bloque `finally` siempre se ejecuta, sin tener en cuenta si una excepción ocurre o no, entonces esto asegurará que los recursos de base de datos son liberados cerrando la conexión. El Ejemplo 13.2 muestra como usar el bloque `finally` para cerrar conexiones.

Ejemplo 13 . 2

En este ejemplo se ilustra como utilizar el bloque `finally`, por ejemplo para cerrar conexiones abiertas.

```
package cap13;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Ejemplo0302 {

    public static void main(String[] args) {

        // Define la variable de tipo Connection
        Connection conn = null;

        // Inicia el manejo de errores estándar
        try {

            //Load a driver with Class.forName.
            Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
            System.out.println("Abriendo la conexión con la base de datos...");
            conn = DriverManager.getConnection("jdbc:oracle:thin:@egcc:1521:XE", "gustavo", "cerebro");

            // Manejando errores JDBC
        } catch (SQLException se) {
            se.printStackTrace();
            // Manejando otros tipos de excepciones
        } catch (Exception e) {
            e.printStackTrace();
            // Bloque finally usado para cerrar la conexión
        } finally {
            try {
                if (conn != null) {
                    String msg = "Cerrando la conexión desde el bloque finally.";
                    System.out.println(msg);
                    conn.close();
                }
            } catch (SQLException se) {
```

```
        se.printStackTrace();  
    } // Fin de try en el bloque finally  
  
}  
} // Fin del método main  
  
} // Fin de la clase Ejemplo1302
```

El resultado de su ejecución es:

Abriendo la conexión con la base de datos...

Cerrando la conexión desde el bloque finally.



14

Consultas

La operación que más realizamos en una base de datos son las consultas, y la instrucción SELECT es la que utilizamos para desarrollarlas. Existen muchas herramientas que permiten la construcción de consultas sin la necesidad de conocer la sintaxis de la sentencia SELECT, pero el programador si está en la obligación conocer no solo su sintaxis, sino también, como aplicarla en las diferentes casuísticas que se presentan en una empresa.

Temas a desarrollar:

- 14.1. Introducción
- 14.2. Usando Statement
- 14.3. Usando PreparedStatement
- 14.4. Uso de CallableStatement

14.1. Introducción

Una de las tareas que se realizan con más frecuencia a una base de datos son las consultas; algunas más sencillas que otras, pero en todos los casos terminamos ejecutando una o más sentencias SELECT. A continuación tenemos algunos casos en los que necesitamos ejecutar consultas:

- Elaborar el catalogo de los artículos de la empresa.
- Averiguar el stock de un artículo.
- Obtener el estado de una cuenta de crédito.
- Consultar saldo disponible de una cuenta de crédito.
- Averiguar el importe de ventas de un artículo.
- Obtener la deuda de un cliente.
- Elaborar el kardex de un artículo.
- Elaborar el reporte de gastos ejecutados por los diferentes centros de costo de una empresa.
- Elaborar el certificado académico de un alumno.
- Elaborar el record académico de un alumno.
- Etc.

Seguramente estará pensando en todos los posibles casos donde es necesario ejecutar una consulta. Al finalizar este capítulo conocerá los objetos que permiten ejecutar todo tipo de consultas, es necesario que conozca al detalle la sentencia SELECT.

La Figura 14.1 muestra el esquema genérico que se utiliza para ejecutar una consulta, como puede observar el resultado de ejecutar la consulta es un conjunto de filas denominado **Conjunto de Resultados** (ResultSet).

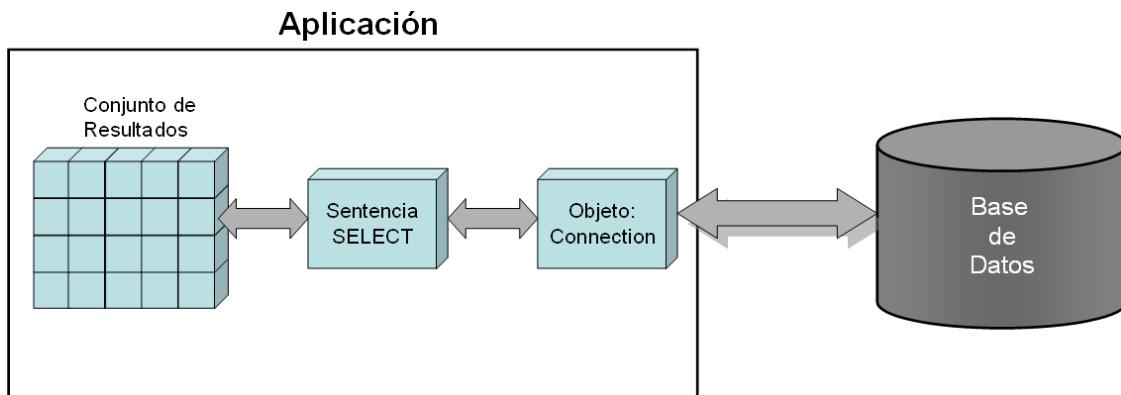


Figura 14 . 1 Esquema genérico de la ejecución de una consulta.

Para ejecutar la consulta podemos utilizar:

- Statement
- PreparedStatement
- CallableStatement

Y para procesar el conjunto de resultados tenemos:

- ResultSet
- RowSet

La Figura 14.2 muestra el esquema de ejecución de una consulta utilizando Statement y ResultSet, note que es necesario el objeto de tipo Connection.

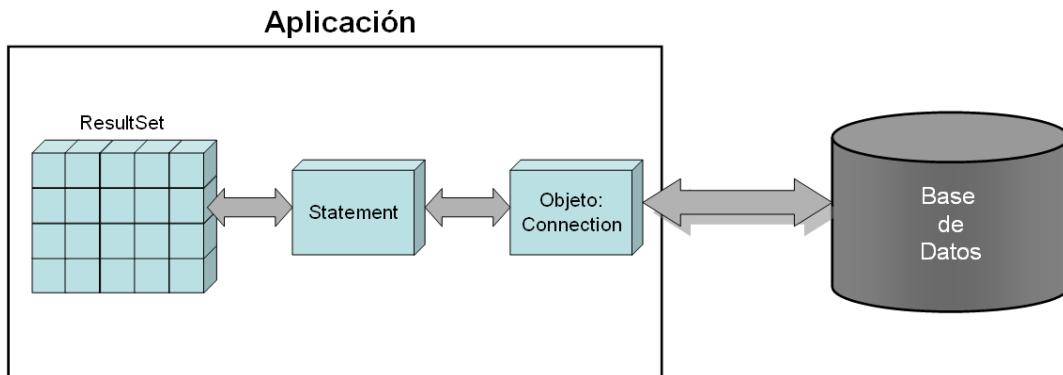


Figura 14 . 2 Esquema de la ejecución de una consulta utilizando Statement y ResultSet.

En la Figura 14.3 tenemos el diagrama UML de las interfaces Statement, PreparedStatement y CallableStatement. Los proveedores JDBC tienen implementaciones de estas interfaces que permiten crear los objetos específicos para cada DBMS.

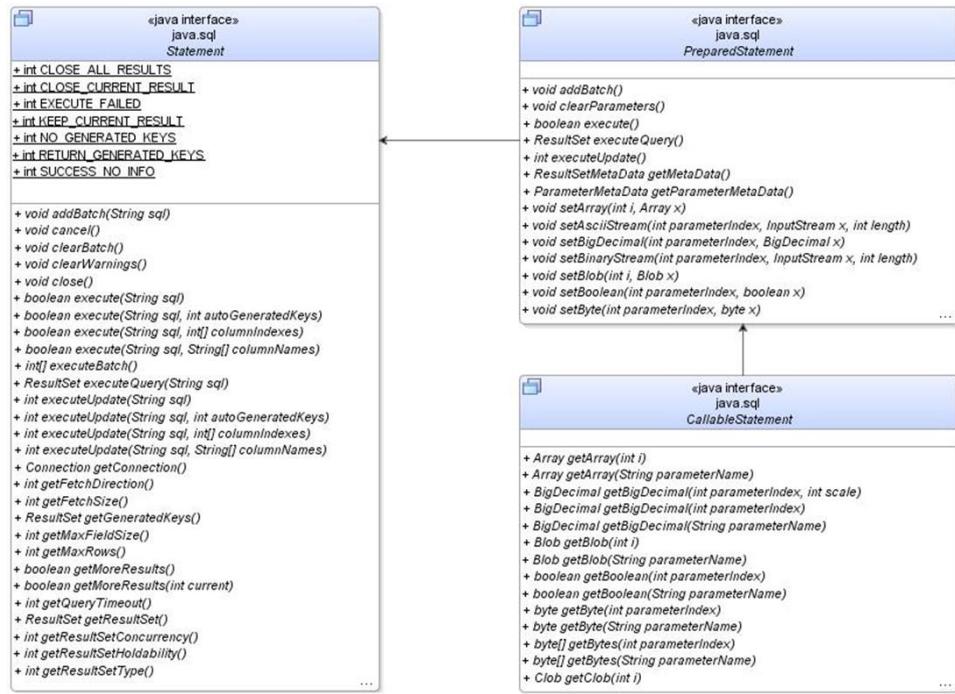


Figura 14 . 3 Diagrama UML de las interfaces Statement, PreparedStatement y CallableStatement.

La Figura 14.4 muestra el diagrama UML de ResultSet y RowSet. Ambos se utilizan para manejar conjunto de resultados, una de las ventajas de RowSet es que puede manejar datos desconectados.

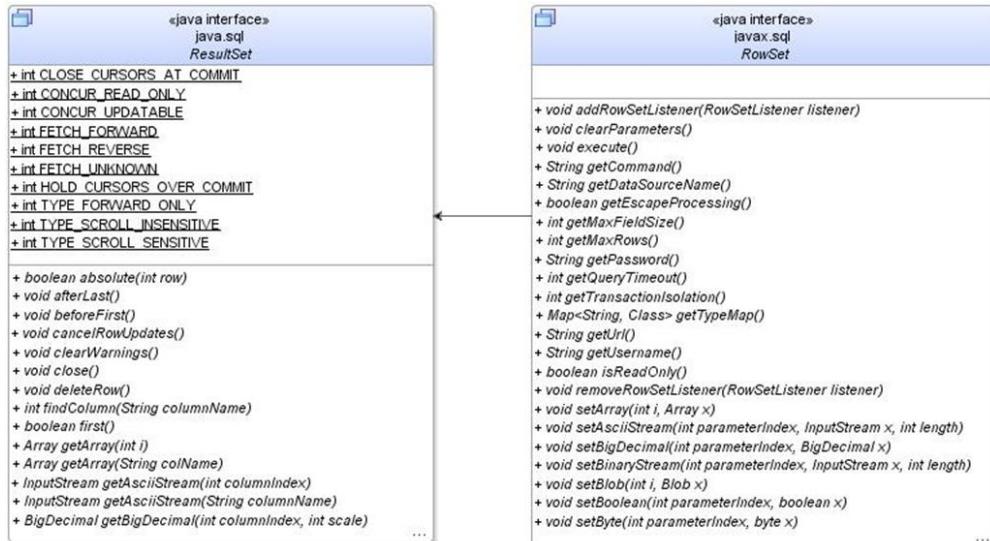


Figura 14 . 4 Diagrama UML de ResultSet y RowSet.

14.2. Usando Statement

Statement permiten ejecutar consultas básicas y recibir los resultados en un ResultSet. Para crear una instancia de Statement, se debe llamar al método `createStatement()` del objeto Connection.

Una vez que se tiene una instancia de Statement, puede ejecutar una consulta SELECT llamando al método `executeQuery(String SQL)`.

También se puede utilizar para actualizar datos en la base de datos, en ese caso se debe utilizar el método `executeUpdate(String SQL)`. Este método retorna el número de registros afectados por el comando de actualización.

Si no sabe de antemano si el comando SQL es un `SELECT`, un `UPDATE` o un `INSERT`, puede usar el método `execute(String SQL)`. Este método retornará `TRUE` si el comando es una sentencia `SELECT`, o `FALSE` si es un comando `UPDATE`, `INSERT` o `DELETE`. Si la consulta es una sentencia `SELECT`, puede recibir los resultados llamando al método `getResultSet()`. Si la consulta es un `UPDATE`, `INSERT` o `DELETE`, puede obtener el número de registros afectados llamando al método `getUpdateCount()` en la instancia de `Statement`.

Ejemplo 14 . 1

En este ejemplo se ilustra el uso de la sentencia `select` para consultar una sola fila, note la forma como es utilizada la sentencia `if` para verificar si la consulta ha tenido éxito. La plantilla genérica es la siguiente:

```
if( rs.next() ) {  
    // script si la consulta tiene éxito.  
}  
else{  
    // script si la consulta no tiene éxito.  
}
```

El script completo de este ejemplo es el siguiente:

```
package cap14;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;  
  
public class Ejemplo1401 {  
  
    public static void main(String[] args) {  
        // Variables  
        Connection cn = null;  
        Statement st = null;  
        ResultSet rs = null;  
        String sql = "select * from empleado where chr_emplcodigo = '0001' ";  
    }  
}
```

```

// Proceso
try {
    // Establecemos la conexión
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
    st = cn.createStatement();
    rs = st.executeQuery(sql);
    if( rs.next() )
        System.out.println( "Empleado: " + rs.getString("vch_emplnombre")
        + " " + rs.getString("vch_emplpaterno") );
    else
        System.out.println("Código no existe.");
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    try {
        if (cn.isClosed() == false)
            cn.close();
    }
} catch (SQLException ex) {
}
}

} // Fin del método main

} // Fin de la clase Ejemplo1401

```

Su ejecución nos da el siguiente resultado:

Empleado: Carlos Alberto Romero

Ejemplo 14 . 2

Este ejemplo también ilustra el uso de la sentencia `select`, pero en este caso retorna mas de una fila, note la forma como se utiliza una variable de tipo `boolean` para verificar si existe o no resultados. La plantilla genérica es la siguiente:

```

existe = false;
while( rs.next() ) {

    existe = true;
    // script si la consulta tiene éxito.
}

```

```
        }
        if( !existe ) {
            // script si la consulta no tiene éxito.
        }
    }
```

El script completo de este ejemplo es el siguiente:

```
package cap14;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Ejemplo1402 {

    public static void main(String[] args) {

        // Variables
        Connection cn = null;
        Statement st = null;
        ResultSet rs = null;
        boolean existe = false;
        String sql = "select * from empleado";

        // Proceso
        try {
            // Establecemos la conexión
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
            st = cn.createStatement();
            rs = st.executeQuery(sql);
            while( rs.next() ) {
                existe = true;
                System.out.println(rs.getString("vch_emplnombre") + " " + rs.getString("vch_emplpaterno"));
            }
            if( !existe )
                System.out.println("No se encontraron datos.");
        } catch (Exception ex) {
```

```

        ex.printStackTrace();
    } finally {
        try {
            if (cn.isClosed() == false) {
                cn.close();
            }
        } catch (SQLException ex) {
        }
    }

}

} // Fin del método main

} // Fin de Ejemplo1402

```

Su ejecución nos da el siguiente resultado:

```

Carlos Alberto Romero
Lidia Castro
Claudia Reyes
Angelica Ramos
Claudia Ruiz
Ricardo Cruz
internet Internet

```

14.3. Usando PreparedStatement

Esta interfaz, al igual que el interfaz Statement, nos permite ejecutar sentencias SQL sobre una conexión establecida con una base de datos. Pero en este caso vamos a ejecutar sentencias SQL precompiladas en las que le podemos definir parámetros de entrada.

El interfaz PreparedStatement hereda del interfaz Statement y se diferencia en dos aspectos:

- Las instancias de PreparedStatement contienen sentencias SQL que ya han sido compiladas. Esto es lo que hace a una sentencia "prepared" (preparada).
- La sentencia SQL contenida en el objeto PreparedStatement puede contener uno o más parámetros de entrada. Un parámetro de entrada es aquél cuyo valor no se especifica cuando la sentencia es creada, en su lugar la sentencia va a tener un signo de interrogación (?) por cada parámetro de entrada. Antes de ejecutarse la sentencia se debe especificar un valor para cada uno de los parámetros a través de los métodos setXXX apropiados. Estos métodos setXXX los añade el interfaz PreparedStatement.

Debido a que las sentencias de los objetos PreparedStatement están precompiladas su ejecución será más rápida que la de los objetos Statement. Por lo tanto, una sentencia SQL que va a ser ejecutada varias veces se recomienda crearla como un objeto

PreparedStatement para ganar en eficiencia. También se utilizará este tipo de sentencias para pasarle parámetros de entrada a las sentencias SQL.

Al heredar de la interfaz Statement, el interfaz PreparedStatement hereda todas sus funcionalidades. Además, añade una serie de métodos que permiten asignar un valor a cada uno de los parámetros de entrada de este tipo de sentencias.

El uso de PreparedStatement es la mejor opción contra el SQL injectado (SQL Injection), que es una vulnerabilidad informática a nivel de la base de datos de una aplicación dinámica. Es, de hecho, un error de una clase más general de vulnerabilidades que pueden ocurrir en cualquier lenguaje de programación o de Script que esté incrustado dentro de otro.

Por ejemplo, supongamos que usted está construyendo una sentencia para consultar los datos de un cliente, la sentencia sería aproximadamente así:

```
sql = "SELECT * FROM cliente WHERE codigo = '" + codigoCliente + "'"
```

Si el usuario ingresa su código, por ejemplo "C001" no habría ningún problema, la sentencia generada sería la siguiente:

```
SELECT * FROM cliente WHERE codigo = 'C001'
```

El problema está en que el usuario puede ingresar un dato malicioso con script SQL que por ejemplo intente dañar la base de datos. Supongamos que el usuario malintencionado ingresa como código "C001'; DROP TABLE cliente; SELECT * FROM nombre like '%", se generaría las siguientes instrucciones SQL:

```
SELECT * FROM cliente WHERE codigo = ' C001';
DROP TABLE cliente;
SELECT * FROM nombre like '%'
```

La primera sentencia es la que se pretende programar, las otras dos son el SQL injectado maliciosamente.

Veamos otro caso, supongamos que usted tiene una ventana de identificación como la que se muestra en la Figura 14.5.

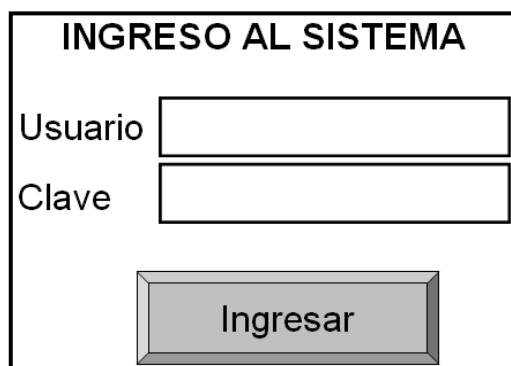


Figura 14 . 5 Ejemplo de una interfaz de acceso a un sistema.

La generación de la sentencia SELECT podría ser de la siguiente manera:

```
sql = "SELECT * FROM usuario WHERE user = '" + usuario + "' AND password = '" + clave + "'"
```

Supongamos que ingresa "gustavo" para el campo **Usuario** y en el campo **Clave** maliciosamente ingresa "123' OR '1='1", se generaría la siguiente sentencia SELECT.

```
SELECT * FROM usuario WHERE user = 'gustavo' AND password = '123' OR '1='1'
```

La condición de esta sentencia es verdadera para todas las filas, con lo cual podrías ingresar al sistema sin importar los datos que contenga el campo **Usuario**.

Una alternativa para evitar esta vulnerabilidad es utilizar `PreparedStatement` para realizar nuestras consultas.

Ejemplo 14 . 3

Este ejemplo ilustra como utilizar `PreparedStatement` para hacer consultas, observe la sentencia donde definimos la consulta SELECT:

```
sql = "select * from cliente where vch_cliedireccion like ?";
```

El signo de interrogación está definiendo un parámetro, se pueden definir varios parámetros. Los parámetros se enumeran a partir de 1 en forma correlativa y en el orden en que se encuentran definidos. Para asignarle datos a los parámetros se utilizan los diferentes métodos `setXXX`, en este caso se está utilizando el método `setString`:

```
ps.setString(1, "%Lince%");
```

Los métodos `setXXX` reciben dos datos, el primero la posición del parámetro y el segundo su valor.

El script completo de este ejemplo es el siguiente:

```
package cap14;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Ejemplo1403 {

    public static void main(String args[]) {

        // Variables
        Connection cn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        boolean existe = false;

        String sql = "select * from cliente where vch_cliedireccion like ?";
```

```
// Proceso
try {
    // Establecemos la conexión
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
    ps = cn.prepareStatement(sql); // Creamos el objeto PreparedStatement
    ps.setString(1, "%Lince%");
    rs = ps.executeQuery();
    while (rs.next()) {
        existe = true;
        System.out.println(rs.getString("vch_clienombre") + " " + rs.getString("vch_cliepaterno"));
    }
    if (!existe) {
        System.out.println("No se encontraron datos.");
    }
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    try {
        if (cn.isClosed() == false) {
            cn.close();
        }
    } catch (SQLException ex) {
    }
}
} // Main
} // class
```

Su ejecución nos da el siguiente resultado:

RICARDO MARCELO
JUAN CARLOS LAY
ENRIQUE MANUEL RODRIGUEZ

14.4. Uso de CallableStatement

Esta interfaz hereda de la interfaz `PreparedStatement` y ofrece la posibilidad realizar llamadas a procedimientos almacenados de la base de datos, también es posible utilizar parámetros de entrada y salida.

Un objeto `CallableStatement` ofrece la posibilidad de realizar llamadas a procedimientos almacenados de una forma estándar para todos los DBMS. Un procedimiento almacenado se encuentra dentro de una base de datos; la llamada a un procedimiento es lo que contiene un objeto `CallableStatement`. Esta llamada está escrita con una sintaxis de escape, esta sintaxis puede tener dos formas diferentes: una con un parámetro de resultado, es un tipo de parámetro de salida que representa el valor devuelto por el procedimiento y otra sin ningún parámetro de resultado. Ambas formas pueden tener un número variable de parámetros de entrada, de salida o de entrada/salida. Cada parámetro está representado por si signo de interrogación.

La sintaxis para realizar la llamada a un procedimiento almacenado es la siguiente:

```
{call nombre_del_procedimiento[(?, ?, ...)]}
```

Si devuelve un parámetro de resultado:

```
{?=call nombre_del_procedimiento[(?, ?, ...)]}
```

La sintaxis de una llamada a un procedimiento sin ningún tipo de parámetros sería:

```
{call nombre_del_procedimiento}
```

Normalmente al crear un objeto `CallableStatement` el programador deberá saber si el DBMS soporta procedimientos almacenados y de qué procedimientos se trata.

El interfaz `CallableStatement` hereda los métodos del interfaz `Statement`, que se encargan de sentencias SQL generales, y también los métodos del interfaz `PreparedStatement`, que se encargan de manejar parámetros de entrada. En la Figura 14.6 se puede ver la relación de herencia que existe entre los tres tipos de interfaces.

Los métodos que define el interfaz `CallableStatement` se encargan de manejar parámetros de salida, registrar los tipos JDBC de los parámetros de salida, recuperar los valores o testeando si un valor devuelto es un JDBC NULL.

Esta interfaz supone un paso más en la especialización de sentencias.

Nota:

`Connection.prepareCall()` es un método costoso, debido a la lectura de metadatos que hace el driver para soportar los parámetros de salida. Por razones de rendimiento, intente minimizar llamadas innecesarias a `Connection.prepareCall()` reutilizando instancias de `CallableStatement` en su código.



Figura 14 . 6 Relación de herencia entre las interfaces Statement, PreparedStatement y CallableStatement.

14.4.1. Utilizando Parámetros

CallableStatement puede tener parámetros de entrada, de salida y de entrada/salida.

Para pasarle parámetros de entrada a un objeto CallableStatement, se utilizan los métodos setXXX que heredaba del interfaz PreparedStatement.

Si el procedimiento almacenado tiene parámetros de salida, el tipo JDBC de cada parámetro de salida debe ser registrado antes de ejecutar el objeto CallableStatement correspondiente.

Para registrar los tipos JDBC de los parámetros de salida se debe ejecutar el método registerOutParameter.

Después de ejecutar el procedimiento, se pueden recuperar los valores de los parámetros de salida llamando al método getXXX adecuado. El método getXXX debe recuperar el tipo Java que se correspondería con el tipo JDBC con el que se registró el parámetro. A los métodos getXXX se le pasará un entero que indicará el valor ordinal del parámetro a recuperar.

El siguiente script crea el procedimiento usp_consulta_telefono_de_cliente que permite consultar el teléfono de un cliente, note que tiene dos parámetros, uno de entrada y otro de salida.

```

DELIMITER $$

CREATE PROCEDURE usp_consulta_telefono_de_cliente
( in codigo char(5), out telefono varchar(20) )
BEGIN

    select vch_clietelefono
    into telefono
    from cliente
    where chr_cliecodigo = codigo;

END$$

```

```
DELIMITER ;
```

El siguiente script ilustra la llamada a un procedimiento almacenado con dos parámetros; uno de entrada y otro de salida:

```
sql = "{call usp_consulta_telefono_de_cliente(?,?)}";
Class.forName("com.mysql.jdbc.Driver").newInstance();
cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
cs = cn.prepareCall(sql);
cs.setString(1, "00003");
cs.registerOutParameter(2, java.sql.Types.VARCHAR);
cs.execute();
telefono = cs.getString(2);
System.out.println("Teléfono: " + telefono);
```

Un parámetro de entrada/salida necesitará que se haga una llamada al método `setXXX` apropiado además de una llamada al método `registerOutParameter()`.

El método `setXXX` asigna un valor al parámetro, tratándolo como un parámetro de entrada, y el método `registerOutParameter()` le asignará al parámetro un tipo JDBC tratándolo como un parámetro de salida. El tipo JDBC indicado en el método `setXXX` para el valor de entrada debe ser el mismo que el tipo JDBC indicado en el método `registerOutParameter()`.

A continuación tenemos el script que crea un procedimiento para consultar el saldo de una cuenta, pero con la particularidad que se utiliza un solo parámetro.

```
DELIMITER $$
```

```
CREATE PROCEDURE usp_consulta_saldo_de_cuenta
( inout dato varchar(15) )
BEGIN

    declare saldo numeric(10,2);

    select dec_cuensaldo into saldo
    from cuenta where chr_cuencodigo = dato;

    if isnull(saldo) then
        set dato = null;
    else
        set dato = cast(saldo as char);
    end if;

END$$
```

```
DELIMITER ;
```

En el siguiente script se está utilizando el procedimiento almacenado `usp_consulta_saldo_de_cuenta` con la finalidad de ilustrar el uso de un parámetro de tipo `inout`.

```
sql = "{call usp_consulta_saldo_de_cuenta(?)}";
Class.forName("com.mysql.jdbc.Driver").newInstance();
cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
```

```
cs = cn.prepareCall(sql);
cs.setString(1, "00200001");
cs.registerOutParameter(1, java.sql.Types.VARCHAR);
cs.execute();
saldo = Double.parseDouble( cs.getString(1) );
System.out.println("Saldo: " + saldo);
```

Puede usted observar que por el mismo parámetro ingresamos el número de cuenta y obtenemos su respectivo saldo.

Ejemplo 14 . 4

Este ejemplo presenta el script completo del uso del procedimiento `usp_consulta_telefono_de_cliente`, el método `wasNull()` permite averiguar si la llamada al último método `getXXX()` fue un valor nulo.

```
package cap14;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Ejemplo1404 {

    public static void main(String args[]) {

        // Variables
        String sql = "{call usp_consulta_telefono_de_cliente(?,?)}";
        Connection cn = null;
        CallableStatement cs = null;
        String telefono = null;

        // Proceso
        try {
            // Establecemos la conexión
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
            // Preparamos la llamada
            cs = cn.prepareCall(sql);
            cs.setString(1, "00888");
            cs.registerOutParameter(2, java.sql.Types.VARCHAR);
```

```

    // Realizamos la llamada
    cs.execute();

    // Obtenemos el valor del parámetro de salida
    telefono = cs.getString(2);

    if (cs.wasNull()) {
        System.out.println("Código no existe.");
    } else {
        System.out.println("Teléfono: " + telefono);
    }

} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    try {
        if (cn.isClosed() == false) { cn.close(); }
    } catch (SQLException ex) { }
}

} // Main

} // Ejemplo1404

```

Ejemplo 14 . 5

Este ejemplo tenemos el código completo del uso del procedimiento usp_consulta_saldo_de_cuenta, note usted como un mismo parámetro se puede utilizar de entrada y salida.

```

package cap14;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Ejemplo1405 {

    public static void main(String args[]) {

        // Variables
        String sql = "{call usp_consulta_saldo_de_cuenta(?)}";
        Connection cn = null;
    }
}

```

```
CallableStatement cs = null;
Double saldo;

// Proceso
try {
    // Establecemos la conexión
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
    // Preparamos la llamada
    cs = cn.prepareCall(sql);
    cs.setString(1, "00200001");
    cs.registerOutParameter(1, java.sql.Types.VARCHAR);
    // Realizamos la llamada
    cs.execute();
    // Obtenemos el valor del parámetro de salida
    saldo = Double.parseDouble(cs.getString(1));
    if (cs.wasNull()) {
        System.out.println("Cuenta no existe.");
    } else {
        System.out.println("Saldo: " + saldo);
    }
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    try {
        if (cn.isClosed() == false) { cn.close(); }
    } catch (SQLException ex) { }
}
}

} // Main

} // Ejemplo1405
```

14.4.2. Manejando Conjunto de Resultados

Los procedimientos almacenados también pueden retornar conjunto de resultados, en este caso contamos con el método `executeQuery()` para ejecutar el procedimiento y retorna un objeto `ResultSet` con el conjunto de resultados.

A continuación tenemos el script que crea el procedimiento almacenado `usp_consultar_clientes_por_distrito` que permite consultar los clientes de un determinado distrito, el nombre del distrito lo ingresamos a través del parámetro `distrito`.

```

DELIMITER $$

CREATE PROCEDURE usp_consultar_clientes_por_distrito
( distrito varchar(50) )
BEGIN

    set distrito = concat('%',distrito,'%');

    select
        chr_cliecodigo as codigo,
        concat(vch_cliepaterno,space(1),vch_cliematerno,
               ',',space(1),vch_clienombre) as nombre
    from cliente
    where vch_cliedireccion like distrito;

END$$

DELIMITER ;

```

El siguiente script muestra el uso del procedimiento almacenado usp_consultar_clientes_por_distrito que retorna un conjunto de resultados.

```

sql = "{call usp_consultar_clientes_por_distrito(?)}";
Class.forName("com.mysql.jdbc.Driver").newInstance();
cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
cs = cn.prepareCall(sql);
cs.setString(1, "LINCE");
rs = cs.executeQuery();
while (rs.next()) {
    System.out.println(rs.getString("codigo") + " " + rs.getString("nombre"));
}

```

Otra alternativa sería utilizar el valor que retorna el método `execute()` (que es de tipo boolean); si el resultado que retorna es true significa que existe por lo menos un conjunto de resultados, podría haber mas de uno.

Para leer un conjunto de resultados se utiliza el método `getResultSet()`, y para averiguar si todavía existe otro conjunto de resultados por leer utilizamos el método `getMoreResults()`.

El siguiente script muestra una plantilla de como se podría procesar varios conjuntos de resultados, y debe ser adaptada al caso específico que necesitemos procesar.

```

boolean hayResultados = cs.execute();

while ( hayResultados ) {

    ResultSet rs = cs.getResultSet();

    // proceso del conjunto de resultados

```

```
hayResultados = cs.getMoreResults();

}
```

Ejemplo 14 . 6

Este ejemplo tenemos el código completo del uso del procedimiento usp_consultar_clientes_por_distrito.

```
package cap14;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Ejemplo1406 {

    public static void main(String args[]) {

        // Variables
        String sql = "{call usp_consultar_clientes_por_distrito(?)}";
        Connection cn = null;
        CallableStatement cs = null;
        ResultSet rs = null;
        boolean existe = false;

        // Proceso
        try {

            // Establecemos la conexión
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "admin");
            cs = cn.prepareCall(sql);
            cs.setString(1, "LINCE");
            rs = cs.executeQuery();

            while (rs.next()) {
                existe = true;
                System.out.println(rs.getString("codigo") + " " + rs.getString("nombre"));
            }

        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
}

if (!existe) {
    System.out.println("No se encontraron datos.");
}

}

} catch (Exception ex) {

    ex.printStackTrace();

}

} finally {

    try {
        if (cn.isClosed() == false) {
            cn.close();
        }
    } catch (SQLException ex) {
    }

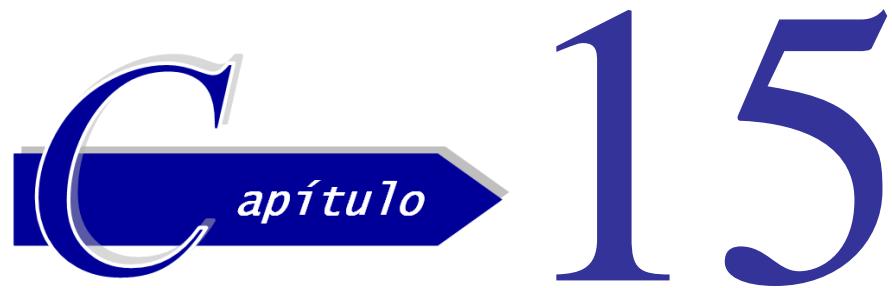
}

}

} // Main

} // Ejemplo1406
```

Página en blanco.



15

Manejo de Transacciones

Todas las aplicaciones manejan transacciones para ejecutar sus procesos, por tal motivo debemos tener mucho cuidado al momento de programarlas. Transacciones mal programadas generan inconsistencias en las bases de datos y por lo tanto los reportes que se obtengan también serán inconsistentes, lo cual sería muy grave para la empresa.

En este libro abordare transacciones controladas desde el cliente y transacciones de base de datos, dejando el tema de transacciones distribuidas para otro texto.

Temas a desarrollar:

- 15.1. Definiciones
- 15.2. Programación de Transacciones

15.1. Definiciones

15.1.1. Transacción

Una transacción es un grupo de acciones que hacen transformaciones consistentes en las tablas preservando la consistencia de la base de datos. Una base de datos está en un estado consistente si obedece todas las restricciones de integridad definidas sobre ella. Los cambios de estado ocurren debido a actualizaciones, inserciones, y eliminación de datos. Se busca asegurar que la base de datos nunca entre en un estado de inconsistencia, sin embargo, durante la ejecución de una transacción la base de datos puede estar temporalmente en un estado inconsistente, tal como se ilustra en la Figura 15.1. El punto importante aquí es asegurar que la base de datos regresa a un estado consistente al finalizar la ejecución de una transacción.

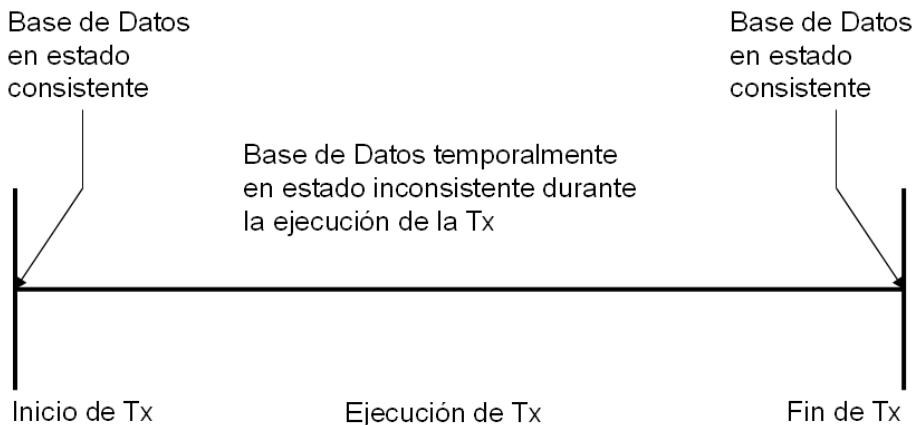


Figura 15 . 1 Estado de la Base de Datos durante la transacción.

Lo que se persigue con el manejo de transacciones es por un lado tener una transparencia adecuada de las acciones concurrentes a una base de datos, y por otro lado tener una transparencia adecuada en el manejo de las fallas que se pueden presentar en una base de datos.

15.1.2. Propiedades de una Transacción

Una transacción debe tener las propiedades ACID, que son las iniciales en inglés de las siguientes características: Atomicity, Consistency, Isolation, Durability.

- #### ▪ Atomicity (Atomicidad)

Una transacción constituye una unidad atómica de ejecución y se ejecuta exactamente una vez; o se realiza todo el trabajo o nada de él en absoluto.

- Consistency (Coherencia)

Una transacción mantiene la coherencia de los datos, transformando un estado coherente de datos en otro estado coherente de datos. Los datos enlazados por una transacción deben conservarse semánticamente.

- Isolation (Aislamiento)

Una transacción es una unidad de aislamiento y cada una se produce aislada e independientemente de las transacciones concurrentes. Una transacción nunca debe ver las fases intermedias de otra transacción.

- Durability (Durabilidad)

Una transacción es una unidad de recuperación. Si una transacción tiene éxito, sus actualizaciones persisten, aun cuando falle el equipo o se apague. Si una transacción no tiene éxito, el sistema permanece en el estado anterior antes de la transacción.

15.1.3. Control de Transacciones

Para controlar transacciones tenemos tres técnicas:

- Transacciones controladas desde el cliente.
 - Transacciones de base de datos.
 - Transacciones distribuidas.

La que se utilice dependerá del tipo de aplicación que se este desarrollando (Cliente-Servidor, Distribuida, Web) y la manera como se ha diseñado el control de transacciones por parte del equipo de trabajo.

15.1.3.1. Transacciones Controladas desde el cliente

Son utilizadas en sistemas Cliente-Servidor; las transacciones son iniciadas y finalizadas con instrucciones que se ejecutan desde la aplicación cliente, tal como se ilustra en la Figura 15.2.

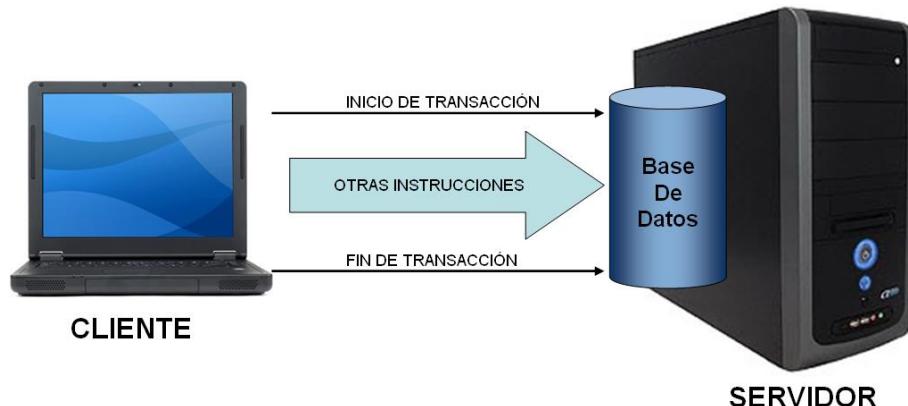


Figura 15 . 2 Transacciones controladas desde el cliente.

Los pasos que se siguen son los siguientes:

1. Se envía la instrucción para iniciar la transacción.
2. Se envían las instrucciones para insertar, modificar o eliminar datos.
3. Se envía la instrucción para confirmar la transacción.

Todos estos pasos deben adaptarse al lenguaje de programación que se esté utilizando, por ejemplo si estamos programando con Java, debemos utilizar JDBC y el esquema general sería:

```

try {

    // Inicio de la Transacción
    cn.setAutoCommit(false);

    // Otras instrucciones

    // Confirmar Transacción
    cn.commit();

} catch (Exception exception) {

    // Cancelar transacción
    cn.rollback();

    // Otras instrucciones de control

}
  
```

15.1.3.2. Transacciones de Base de Datos.

Para este caso se utilizan procedimientos almacenados en la base de datos, y son estos los que toman el control de la transacción. Por lo tanto, las transacciones son iniciadas y finalizadas con instrucciones que se ejecutan desde el procedimiento almacenado, tal como se ilustra en la Figura 15.3.

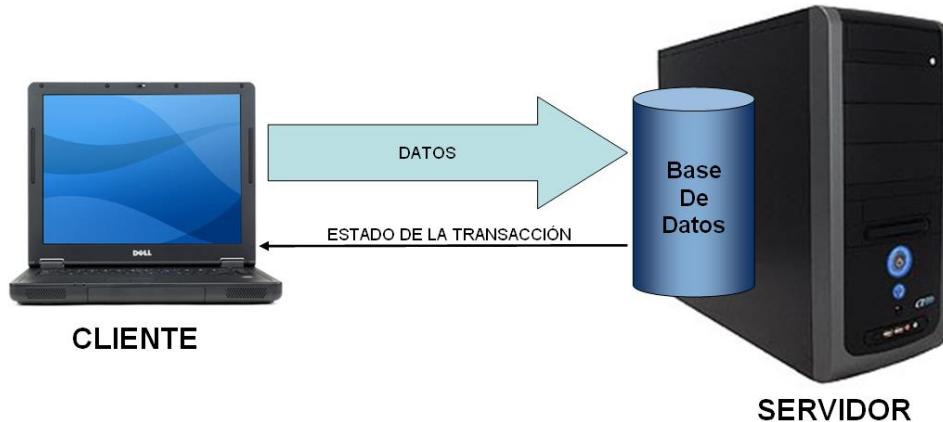


Figura 15 . 3 Transacciones controladas en la base de datos.

El problema está en pasarle los datos al procedimiento, sobre todo cuando estos son fijos, por ejemplo como pasarle los datos de una factura, donde el número de ítems es variable.

A continuación tenemos esquemas generales para las bases de datos más utilizadas:

- MySQL

```
delimiter //  
  
CREATE PROCEDURE nombre_procedimiento ( parámetros )  
BEGIN  
  
    -- Iniciar Transacción  
    START TRANSACTION;  
  
    -- Instrucciones INSERT, UPDATE ó DELETE  
  
    -- Confirmar Transacción  
    COMMIT;  
  
END  
//  
  
delimiter;
```

- SQL Server 2000

```
CREATE PROCEDURE nombre_procedimiento
( parámetros )
AS
BEGIN
    -- Inicio de la Transacción
    BEGIN TRANSACTION

    -- Instrucciones INSERT, UPDATE ó DELETE

    -- Confirmar Transacción
    COMMIT TRANSACTION
END
GO
```

- SQL Server 2005

```
CREATE PROCEDURE esquema.nombre_procedimiento
( parámetros )
AS
BEGIN
    BEGIN TRY
        -- Inicio de la Transacción
        BEGIN TRANSACTION

        -- Instrucciones INSERT, UPDATE ó DELETE

        -- Confirmar Transacción
        COMMIT TRANSACTION

    END TRY
    BEGIN CATCH
        -- Cancelar Transacción
        ROLLBACK TRANSACTION

        -- Instrucciones INSERT, UPDATE ó DELETE

    END CATCH
END
GO
```

- Oracle

```
CREATE OR REPLACE PROCEDURE nombre_procedimiento
( parámetros )
AS
BEGIN

    -- Instrucciones INSERT, UPDATE ó DELETE

    -- Confirmar Transacción
    COMMIT;

EXCEPTION

WHEN OTHERS THEN
    ROLLBACK;
    -- Otras Instrucciones

END;
```

Cada esquema se debe adaptar al caso específico que se quiere programar.

15.1.3.3. Transacciones Distribuidas

En este caso se utiliza un servidor de aplicaciones que se encarga de administrar la transacción, tal como se ilustra en la Figura 15.4.



Figura 15 . 4 Transacciones Distribuidas.

En este tipo de solución el servidor de aplicaciones proporciona los componentes de software que controlan el acceso a los datos y las transacciones que se realizan. Las fuentes de datos pueden ser una ó más bases de datos, incluso de diferentes proveedores.

Si trabajamos con Java debemos usar un servidor JEE y componentes Enterprise Java Beans.

15.2. Programación de Transacciones

15.2.1. Transacciones Controladas desde el cliente

Las transacciones controladas desde el cliente se programan desde el objeto `Connection` que utilizamos para acceder a la base de datos.

Durante el tiempo que dura una transacción, las filas involucradas son automáticamente bloqueadas para los otros usuarios, es por eso que debemos tener especial cuidado en evitar que las transacciones tomen tiempos muy grandes, para lograr esto debemos programarlas en un método y evitar la interacción con el usuario hasta que la transacción finalice.

A continuación tenemos una descripción de los métodos que debemos utilizar:

- **`setAutoCommit()`**

Establece el modo auto-commit de la conexión para tratar las transacciones.

Sintaxis:

```
ObjConnection.setAutoCommit( flag );
```

El parámetro `flag` es de tipo boolean y puede tomar los siguientes valores:

- true** Habilita el modo auto-commit, en cuyo caso cada instrucción SQL que se ejecute será automáticamente confirmada.
- false** Deshabilita el modo auto-commit. Esta opción se utiliza cuando queremos agrupar varias sentencias SQL en una misma transacción o cuando queremos que la transacción sea manejada en la base de datos dentro de un procedimiento almacenado.

JDBC trata las transacciones de forma implícita cuando se deshabilita el auto-commit, lo que significa, que no hay un método que inicie la transacción explícitamente, todas las sentencias SQL son agrupadas automáticamente en una transacción hasta que sean confirmadas con el método `commit()` o canceladas con el método `rollback()`, y desde ese momento nuevamente se agrupan las sentencias SQL en una nueva transacción, tal como se muestra en la Figura 15.5.

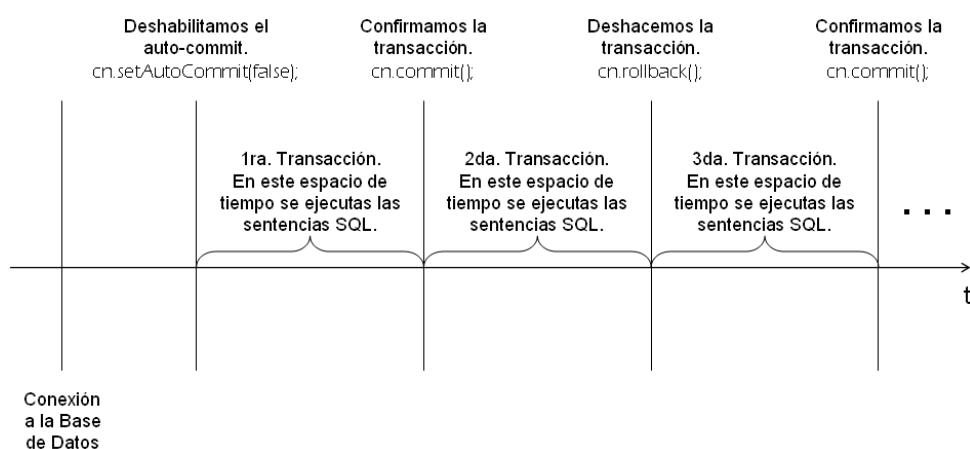


Figura 15 . 5 Esquema de transacciones en el tiempo.

- **`commit()`**

Hace permanente los cambios realizados por las sentencias SQL en la base de datos, y libera los bloqueos mantenidos por la transacción. En la Figura 15.5 tenemos un esquema de cómo se debe utilizar este método.

- **rollback()**

Deshace los cambios realizados por las sentencias SQL en una transacción y libera todos los bloqueos asociados.

En la Figura 15.5 podemos observar que todos los cambios realizados por las sentencias SQL de la 2da transacción son deshechas por el método `rollback()`.

Ejemplo 15 . 1

En este ejemplo ilustraremos el uso de patrones de diseño de software para tener un estándar en el desarrollo de aplicaciones.

Se trata de desarrollar una clase que nos permita registrar un nuevo cliente, a continuación tenemos el diagrama de clases que clarifica la solución:

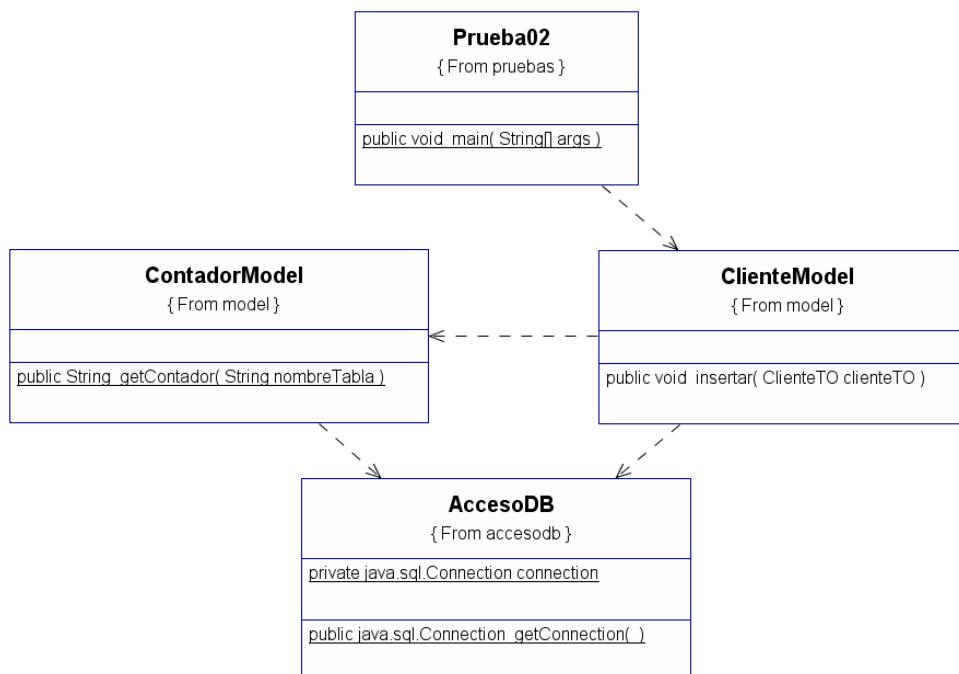


Figura 15 . 6 Diagrama de clases para registrar un nuevo cliente.

1. Acceso a la Base de Datos

Para acceder a la base de datos utilizaremos una clase de nombre `AccesoDB` a través de la cual accederemos a un objeto de tipo `Connection`.

Esta clase implementa el patrón Singleton que nos garantiza que siempre accederemos al mismo objeto `Connection`.

El script es el siguiente:

```

package egcc.accesodb;

import java.sql.Connection;
import java.sql.DriverManager;
  
```

```

public class AccesoDB {

    private static Connection connection = null;

    public static Connection getConnection() throws Exception {

        String url = "jdbc:mysql://localhost/eurekabank";
        String user = "root";
        String pwd = "admin";

        if (connection == null) {

            Class.forName("com.mysql.jdbc.Driver").newInstance();
            connection = DriverManager.getConnection(url, user, pwd);

        }

        return connection;

    } // getConnection

} // AccesoDB

```

2. Clase para Generar el Código del Cliente

La clase ContadorModel sirve para generar el código de cualquier tabla.

El método `getContador` recibe como parámetro el nombre de la tabla y hace la consulta a la tabla `contador` para obtener el código del nuevo registro en la tabla. También debe incrementar el contador para el siguiente código.

Note que el método `getContador` no maneja ninguna transacción; sin embargo, debe ejecutarse dentro de una transacción, porque la generación del código forma parte de un proceso de inserción de un nuevo registro, por ejemplo un nuevo cliente, como se verá mas adelante.

El script es el siguiente:

```

package egcc.model;

import egcc.accesodb.AccesoDB;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class ContadorModel {

    public static String getContador( String nombreTabla) throws Exception{

        Connection cn = AccesoDB.getConnection();
        PreparedStatement ps = null;
        ResultSet rs = null;
        String contador = null;

```

```
String sql = "select right(concat(repeat('0',int_contlongitud)," +
    "int_contitem),int_contlongitud) as item from contador " +
    "where vch_conttabla = ?";

ps = cn.prepareStatement(sql);
nombreTabla = nombreTabla.toUpperCase();
ps.setString(1, nombreTabla);
rs = ps.executeQuery();

if( ! rs.next() ){
    return contador;
}

contador = rs.getString("item");

sql = "update contador " +
    "set int_contitem = int_contitem + 1 " +
    "where vch_conttabla = ?";

ps = cn.prepareStatement(sql);
ps.setString(1, nombreTabla);
ps.executeUpdate();

rs.close();
ps.close();

return contador;

} // getContador

} // ContadorModel
```

3. Clase para Registrar el Nuevo Cliente

En la clase `ClienteModel` se debe programar todos los procesos de negocio referidos a la tabla cliente, en este caso estamos ilustrando la inserción de un nuevo cliente.

Puede comprobar que en el método `insertar` se programa el proceso para insertar un nuevo registro en la tabla cliente, podrá usted darse cuenta que se programa todo el proceso dentro de una transacción.

A continuación tenemos el script completo:

```
package egcc.model;

import egcc.accesodb.AccesoDB;
import egcc.to.ClienteTO;
import java.sql.Connection;
import java.sql.PreparedStatement;
```

```

import java.sql.SQLException;

public class ClienteModel {

    public void insertar(ClienteTO clienteTO) throws Exception {

        Connection cn = null;
        PreparedStatement ps = null;
        String sql = null;
        try {
            // Obtenemos la conexión con la base de datos
            cn = AccesoDB.getConnection();
            // Deshabilitamos el auto-commit
            cn.setAutoCommit(false);
            // Obtenemos el código del nuevo cliente
            clienteTO.setClicodigo(ContadorModel.getContador("Cliente"));
            // Preparamos la sentencia para grabar
            sql = "insert into cliente(chr_clicodigo,vch_cliepaterno,vch_ciematerno," +
                "vch_cienombre,chr_ciedni,vch_clieciudad,vch_ciedireccion,vch_cietefono," +
                "vch_cleemail) values(?,?,?,?,?,?,?,?,?,?)";
            // Creamos el objeto PreparedStatement
            ps = cn.prepareStatement(sql);
            // Asignamos valores a los parámetros
            ps.setString(1, clienteTO.getClicodigo());
            ps.setString(2, clienteTO.getCliepaterno());
            ps.setString(3, clienteTO.getCiematerno());
            ps.setString(4, clienteTO.getCienombre());
            ps.setString(5, clienteTO.getCiedni());
            ps.setString(6, clienteTO.getClieciudad());
            ps.setString(7, clienteTO.getClidireccion());
            ps.setString(8, clienteTO.getCietefono());
            ps.setString(9, clienteTO.getClleemail());
            // Ejecutamos
            ps.executeUpdate();
            // Confirmamos la transacción
            cn.commit();
            // Cerramos el objeto PreparedStatement
            ps.close();
        } catch (Exception e) {
            try {
                // Cancelamos la transacción
                cn.rollback();
            } catch (SQLException ex) { }
            throw e;
        }
    }
}

} // insertar

} // ClienteModel

```

4. Prueba del Método insertar

Para probar el método insertar de la clase ClienteModel crearemos la clase Prueba02, y en esta clase crearemos el método main donde programaremos la prueba respectiva.

El script es el siguiente:

```
package egcc.pruebas;

import egcc.model.ClienteModel;
import egcc.to.ClienteTO;

public class Prueba02 {

    public static void main(String[] args) {

        // Creación de los objetos
        ClienteTO clienteTO = new ClienteTO();
        ClienteModel cliente = new ClienteModel();

        // Preparamos los datos
        clienteTO.setClepaterno("GUARACHI");
        clienteTO.setClematerno("CORONEL");
        clienteTO.setClienombre("ALEJANDRA MARIEL");
        clienteTO.setCledni("10459823");
        clienteTO.setCliciudad("CHICLAYO");
        clienteTO.setCledireccion("BARCELONA 2345");
        clienteTO.setCletelefono("234567");
        clienteTO.setCleemail("aleguarachi@hotmail.com");

        //Grabamos en la base de datos
        try {
            cliente.insertar(clienteTO);
            System.out.println("Código Generado: " + clienteTO.getClecodigo());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
} // main
} // Prueba02
```

Cuando lo ejecutamos obtenemos el siguiente resultado:

```
Código Generado: 00021
```

Si examinamos la tabla CLIENTE en la herramienta MySQL Query Browser el resultado es el siguiente:

The screenshot shows the MySQL Query Browser interface. The query window contains the command: `SELECT * FROM CLIENTE ORDER BY 1 DESC;`. The results window, titled "Resultset 1", displays a table with 21 rows of data from the CLIENTE table. The columns are: chrcodigo, vchcliepaterno, vchcliematerno, vchclienombre, chrciedni, and vchclieciudad. The data includes various names and IDs, such as GUARACHI, CORONEL, ALEJANDRA MARIEL, 10459823, CHICLAYO; ZEGARRA, GARCIA, FERNANDO MOISES, 10772365, LIMA; YAURICASA, BAUTISTA, YESABETH, 10934584, LIMA; etc. The bottom status bar indicates "21 rows fetched in 0.0068s (0.0002s)".

chrcodigo	vchcliepaterno	vchcliematerno	vchclienombre	chrciedni	vchclieciudad
00021	GUARACHI	CORONEL	ALEJANDRA MARIEL	10459823	CHICLAYO
00020	ZEGARRA	GARCIA	FERNANDO MOISES	10772365	LIMA
00019	YAURICASA	BAUTISTA	YESABETH	10934584	LIMA
00018	VALENTIN	COTRINA	JUAN DIEGO	10398247	LIMA
00017	VALDEVIESO	LEYVA	ROXANA	10452682	LIMA
00016	TEJADA	DEL AGUILA	TANIA LORENA	10446791	LIMA
00015	ROJAS	OSCANOA	FELIX NINO	10238943	LIMA
00014	RODRIGUEZ	RAMOS	ENRIQUE MANUEL	10773345	LIMA
00013	RICALDE	RAMIREZ	ROSARIO ESMERALDA	10761324	LIMA
00012	MONTALVO	SOTO	DEYSI LIDIA	10612376	LIMA
00011	LAY	VALLEJOS	JUAN CARLOS	10942287	LIMA
00010	GONZALES	GARCIA	GABRIEL ALEJANDRO	10192376	LIMA
00009	FLORES	SHUTE	CRISTIAN RAFAEL	10346723	LIMA
00008	FLORES	CHAFLIQUE	ROSA LIZET	10773456	LIMA
00007	CHAVEZ	CANALES	EDGAR RAFAEL	10145693	LIMA
00006	AYALA	PAZ	JORGE LUIS	10679245	LIMA
00005	ARANDA	LUNA	ALAN ALBERTO	10875611	LIMA

Figura 15 . 7 Examinando la tabla CLIENTE.

15.2.2. Transacciones de Base de Datos

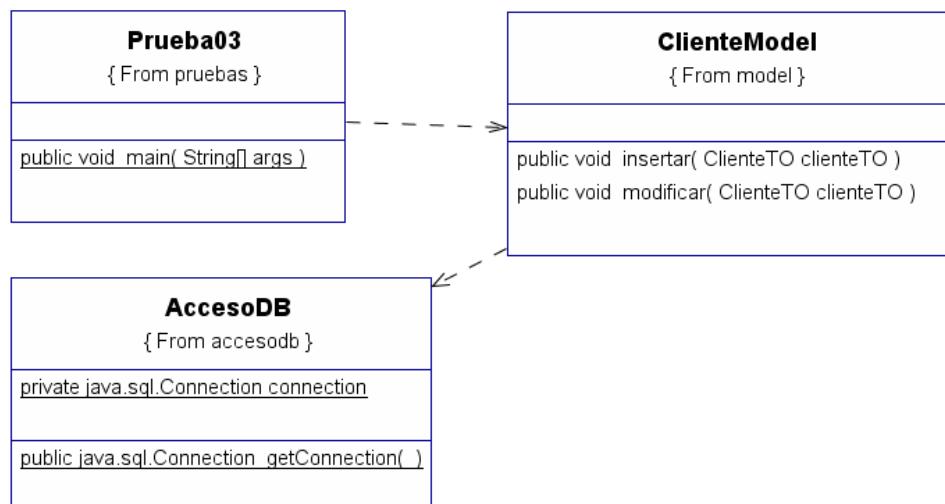
Las transacciones de base de datos se ejecutan íntegramente en el servidor de base de datos y se utilizan procedimientos almacenados para su implementación.

El procedimiento almacenado recibe los datos que son enviados desde la aplicación cliente, luego el procedimiento almacenado inicia su transacción, ejecuta las instrucciones correspondientes al proceso que debe ejecutar y finalmente confirma la transacción o la cancela si hay algún error.

Ejemplo 15 . 2

En este ejemplo ilustraremos como implementar transacciones de base de datos, se trata de desarrollar un método (modificar) en la clase ClienteModel del Ejemplo 15.1 que permita actualizar los datos de un cliente utilizando un procedimiento almacenado.

En la Figura 15.8 puede usted observar el diagrama de clases que aclara la solución.

**Figura 15 . 8** Diagrama de clases para modificar los datos de un cliente.

1. Programación del Procedimiento Almacenado

Nuestro razonamiento para construir el procedimiento podría ser el siguiente:

El procedimiento almacenado recibe los datos para ejecutar el proceso de actualización, y luego de ejecutar su proceso debe retornar un valor indicando el estado de su ejecución.

Un primer análisis de nuestro razonamiento podría ser el siguiente:

El procedimiento almacenado recibe los datos y ejecuta el proceso, y solo en caso de que exista algún error (excepción) se cancela la transacción y el procedimiento lanza una excepción con el mensaje respectivo.

Como pueden leer nuestro análisis tiene sentido y en muchos motores de bases de datos se pueden implementar sin mayor problema, pero MySQL no puede lanzar excepciones desde un procedimiento, entonces debemos plantar otra solución, que podría ser la siguiente:

Implementamos en el procedimiento almacenado un parámetro de salida a través del cual comunicamos el resultado de la ejecución de la siguiente manera: "Si el proceso se ejecuta sin errores el valor del parámetro será nulo (null) y en caso de que se produzca algún error a través del parámetro se comunicará el mensaje de error".

El script que implementa el procedimiento almacenado es:

```
DELIMITER $$

DROP PROCEDURE IF EXISTS usp_actualiza_datos_cliente$$

CREATE PROCEDURE usp_actualiza_datos_cliente(
    out p_estado varchar(200), -- Parámetro de salida
    p_cliecodigo char(5),
    p_cliepaterno varchar(25),
    p_cliematerno varchar(25),
    p_clienombre varchar(30),
    p_cliedni char(8),
    p_clieciudad varchar(30),
    p_cliedireccion varchar(50),
    p_clietelefono varchar(20),
    p_clieemail varchar(50)
)
BEGIN

    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
    BEGIN
        rollback;
        set p_estado = 'Error en el proceso de actualización.';
    END;

    start transaction;
    set p_estado = null;
    update cliente
    set
        vch_cliepaterno = p_cliepaterno,
        vch_cliematerno = p_cliematerno,
        vch_clienombre = p_clienombre,
        chr_cliedni = p_cliedni,
        vch_clieciudad = p_clieciudad,
        vch_cliedireccion = p_cliedireccion,
        vch_clietelefono = p_clietelefono,
        vch_clieemail = p_clieemail
    where
        chr_cliecodigo = p_cliecodigo;

    commit;
END$$

DELIMITER ;
```

A continuación tenemos una prueba del procedimiento:

```
mysql> use eurekabank;
Database changed

mysql> select chr_cliecodigo, vch_cliedireccion, vch_clieemail
-> from cliente order by 1 desc limit 1;
+-----+-----+-----+
| chr_cliecodigo | vch_cliedireccion | vch_clieemail |
+-----+-----+-----+
| 00021         | BARCELONA 274      | aleguarachi@hotmail.com |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> CALL usp_actualiza_datos_cliente( @estado, '00021', 'GUARACHI',
-> 'CORONEL', 'ALEJANDRA MARIEL', '10459823', 'CHICLAYO',
-> 'FRANCISCO CABRERA 1255', '2398456', 'alejandra@perudev.com' );
Query OK, 0 rows affected (0.01 sec)

mysql> select @estado;
+-----+
| @estado |
+-----+
| NULL    |
+-----+
1 row in set (0.00 sec)
```

Puede usted notar que el valor de la variable @estado es NULL, lo que significa que los cambios se ejecutaron correctamente, y lo podemos comprobar en la siguiente consulta:

```
mysql> select chr_cliecodigo, vch_cliedireccion, vch_clieemail
-> from cliente order by 1 desc limit 1;
+-----+-----+-----+
| chr_cliecodigo | vch_cliedireccion | vch_clieemail |
+-----+-----+-----+
| 00021         | FRANCISCO CABRERA 1255 | alejandra@perudev.com |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Otra prueba sería tratar de grabar un valor nulo en la columna correspondiente al apellido materno, deberíamos tener un mensaje de error, tal como se ilustra a continuación:

```
mysql> select chr_cliecodigo, vch_cliematerno
-> from cliente order by 1 desc limit 1;
+-----+-----+
| chr_cliecodigo | vch_cliematerno |
+-----+-----+
| 00021         | CORONEL        |
+-----+-----+
1 row in set (0.00 sec)

mysql> CALL usp_actualiza_datos_cliente( @estado, '00021', 'GUARACHI',
-> NULL, 'ALEJANDRA MARIEL', '10459823', 'CHICLAYO',
-> 'FRANCISCO CABRERA 1255', '2398456', 'alejandra@perudev.com' );
Query OK, 0 rows affected (0.02 sec)

mysql> select @estado;
+-----+
| @estado |
+-----+
| Error en el proceso de actualización. |
+-----+
1 row in set (0.00 sec)

mysql> select chr_cliecodigo, vch_cliematerno
-> from cliente order by 1 desc limit 1;
+-----+-----+
| chr_cliecodigo | vch_cliematerno |
+-----+-----+
| 00021         | CORONEL        |
+-----+-----+
1 row in set (0.00 sec)
```

Como ha podido usted comprobar no es posible establecer en NULL el valor del apellido materno, por el contrario se ha obtenido el mensaje de error respectivo.

2. Programación del Método modificar en la clase ClienteModel

Este método recibe los datos encapsulados en un objeto de tipo ClienteTO para que pueda ejecutar su proceso.

El script es el siguiente:

```
public void modificar(ClienteTO clienteTO) throws Exception {

    Connection cn = null;
    CallableStatement cs = null;
    String estado = null;
    String sql = null;

    try {
        // Obtenemos la conexión con la base de datos
        cn = AccesoDB.getConnection();
        // Deshabilitamos el auto-commit
        cn.setAutoCommit(true);
        // Preparamos el procedimiento
        sql = "{call usp_actualiza_datos_cliente(?,?,?,?,?,?,?,?,?,?)}";
        // Preparamos la sentencia
        cs = cn.prepareCall(sql);
        // Establecemos los parametros
        cs.registerOutParameter(1, java.sql.Types.VARCHAR);
        cs.setString(2, clienteTO.getClecodigo());
        cs.setString(3, clienteTO.getCliepaterno());
        cs.setString(4, clienteTO.getCliematerno());
        cs.setString(5, clienteTO.getClienombre());
        cs.setString(6, clienteTO.getCiedni());
        cs.setString(7, clienteTO.getCieciudad());
        cs.setString(8, clienteTO.getCiedireccion());
        cs.setString(9, clienteTO.getClitelefono());
        cs.setString(10, clienteTO.getClieemail());
        // Ejecutar el proceso
        cs.executeUpdate();
        // Obtenemos el valor de estado
        estado = cs.getString(1);
        // Verificamos el estado
        if( estado != null){
            // Lanzamos la excepción
            throw new Exception(estado);
        }
        cs.close();
    } catch (Exception e) {
        throw e;
    }
}
```

```
    } // modificar
```

3. Prueba del Método modificar

Para probar el método modificar de la clase ClienteModel crearemos la clase Prueba03, y en esta clase crearemos el método main donde programaremos la prueba respectiva.

El script es el siguiente:

```
package egcc.pruebas;

import egcc.model.ClienteModel;
import egcc.to.ClienteTO;

public class Prueba03 {

    public static void main(String[] args) {

        // Creación de los objetos
        ClienteTO clienteTO = new ClienteTO();
        ClienteModel cliente = new ClienteModel();

        // Preparamos los datos
        clienteTO.setClicodigo("00021");
        clienteTO.setClepaterno("GUARACHI");
        clienteTO.setClematerno("CORONEL");
        clienteTO.setClienombre("ALEJANDRA MARIEL");
        clienteTO.setCledni("10459823");
        clienteTO.setCleciudad("CHICLAYO");
        clienteTO.setCledireccion("LUIS GONZALES 1528");
        clienteTO.setCletelefono("234567");
        clienteTO.setCleemail("alejandra@yahoo.es");

        //Grabamos en la base de datos
        try {
            cliente.modificar(clienteTO);
            System.out.println("Proceso ejecutado satisfactoriamente.");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
} // main

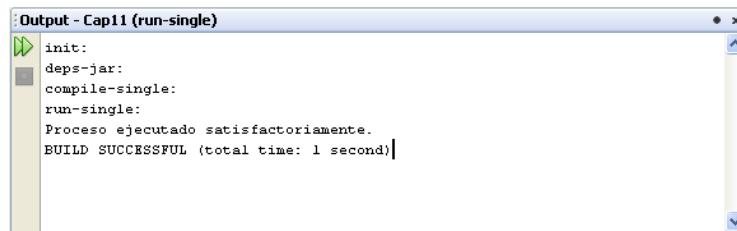
} // Prueba03
```

Los datos antes de ejecutar la clase Prueba03 son:

```
mysql> select chr_clicodigo, vch_cledireccion, vch_cleemail
-> from cliente order by 1 desc limit 1;
+-----+-----+-----+
| chr_clicodigo | vch_cledireccion | vch_cleemail |
+-----+-----+-----+
| 00021         | FRANCISCO CABRERA 1255 | alejandra@perudev.com |
```

```
+-----+-----+
1 row in set (0.00 sec)
```

El resultado de ejecutar la clase Prueba03 es:



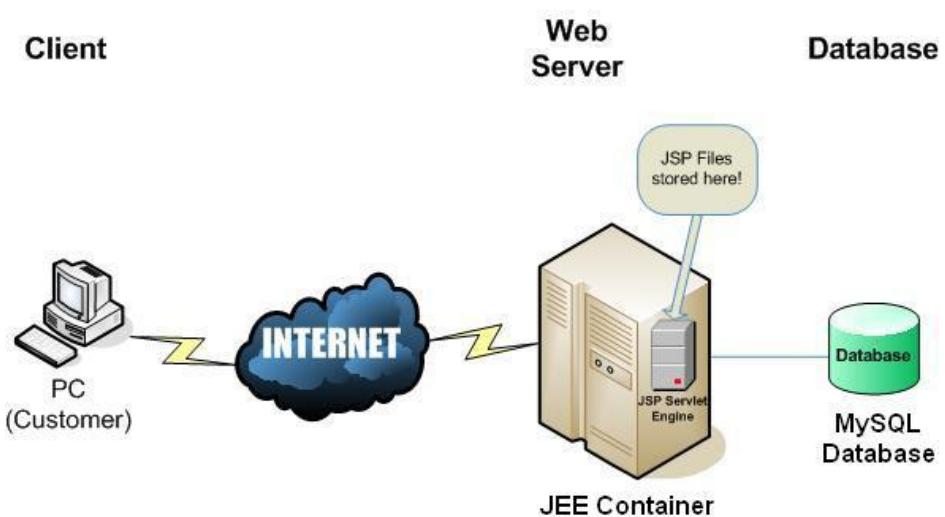
The screenshot shows an IDE's output window titled "Output - Cap11 (run-single)". It displays the following build logs:

```
init:
deps-jar:
compile-single:
run-single:
Proceso ejecutado satisfactoriamente.
BUILD SUCCESSFUL (total time: 1 second)
```

Y los datos son los siguientes:

```
mysql> select chr_cliecodigo, vch_cliedireccion, vch_clieemail
-> from cliente order by 1 desc limit 1;
+-----+-----+
| chr_cliecodigo | vch_cliedireccion | vch_clieemail      |
+-----+-----+
| 00021          | LUIS GONZALES 1528 | alejandra@yahoo.es |
+-----+-----+
1 row in set (0.00 sec)
```

Parte IV



Desarrollo Web con JSP

Página en blanco.



16

Servlets

En el desarrollo de aplicaciones web con Java, la base son los servlets, es por eso que debemos dedicarle el tiempo necesario para entender su funcionamiento y como podemos aprovecharlos en la construcción de soluciones.

Los servlets son bastante utilizados en la implementación del controlador cuando aplicamos el patrón de diseño MVC en la construcción de aplicaciones web.

Temas a desarrollar:

- 16.1. Introducción General
- 16.2. Introducción a los Servlets
- 16.3. Arquitectura del Paquete Servlet
- 16.4.- Un Servlet Sencillo
- 16.5.- Interacción con los Clientes
- 16.6.- Programación de Servlets
- 16.7. Servlets y JavaBeans
- 16.8. Interacción con un Servlet
- 16.9. Sesiones

16.1. Introducción General

En cualquier aplicación enfocada a la Web, es necesario programar el servidor, es decir, realizar una serie de programas que respondan a los requerimientos de los usuario, y generen páginas web dinámicas.

Sobre esta base, se han desarrollado diferentes formas de realizar dicha programación. Una de las más populares en el entorno Windows son las ASP (Active Server Pages), que consisten en una serie de etiquetas incluidas en páginas web, que usan Visual Basic como lenguaje.

Dada la popularidad del lenguaje Java, existen en la actualidad varias formas de usarlo dentro de un servidor Web, entre ellas destacamos dos: mediante servlets, pequeños programas en Java que se ejecutan de forma persistente en el servidor, y que, por lo tanto, tienen una activación muy rápida, y una forma más simple de hacerlo, los JSP (Java Server Pages), que consisten en pequeños trozos de código en Java que se insertan dentro de páginas web, de forma análoga a los ASPs anteriores. Ambas opciones, hoy en día, son muy populares en sitios de comercio electrónico. Frente a los ASPs, la ventaja que presentan los servlets y JSP es que son independientes del sistema operativo y del procesador de la máquina.

Por su parte, PHP es un lenguaje cuyos programas se insertan también dentro de las páginas web, al igual que los ASPs y JSPs; es mucho más simple de usar, y el acceso a bases de datos desde él es muy sencillo. Es tremadamente popular en sitios de comercio electrónico con poco tráfico, por su facilidad de desarrollo y rapidez de implantación.

Finalmente, los CGI (Common Gateway Interface) era el único método disponible originalmente, y consiste en programas que se lanzan desde el servidor, y que, por lo tanto, pueden estar escritos en cualquier lenguaje, compilados o en código fuente. También son independientes del SO, y presentan la ventaja de que, dado un programa escrito en un lenguaje cualquiera, es fácil adaptarlo a un CGI. Entre los lenguajes que se usan para CGIs, el más popular es el Perl.

16.2. Introducción a los Servlets

16.2.1. ¿Qué es un Servlets?

Los Servlets son módulos que extienden los servidores orientados a requerimiento-respuesta, como los servidores Web compatibles con Java. Por ejemplo, un servlet podría ser responsable de tomar los datos de un formulario de entrada de pedidos en HTML y aplicarle la lógica de negocios utilizada para actualizar la base de datos de pedidos de una compañía.

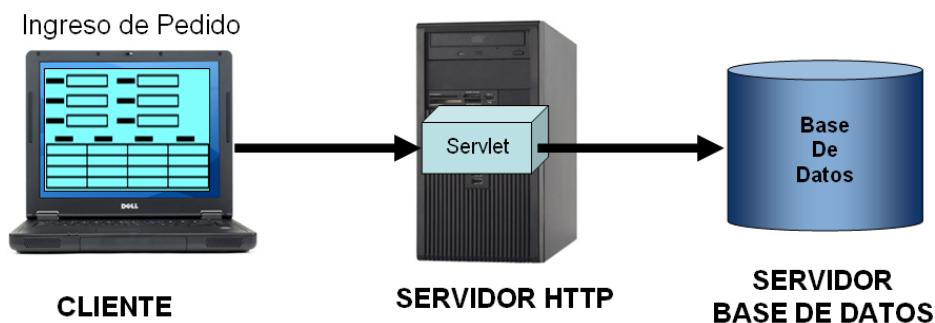


Figura 16 . 1 Esquema de funcionamiento de un servidor HTTP.

Los Servlets son para los servidores lo que los applets son para los navegadores. Sin embargo, al contrario de los applets, los servlets no tienen interfaz gráfica de usuario.

Los servlets pueden ser incluidos en diferentes servidores porque la API Servlet, la que se utiliza para escribir Servlets, no asume nada sobre el entorno o protocolo del servidor. Los servlets se utilizan ampliamente dentro de servidores HTTP; muchos servidores Web soportan la API Servlet.

16.2.2. Ventajas de los Servlets

Los Servlets Java son más eficientes, fáciles de usar, más poderosos, más portables, y más baratos que el CGI tradicional y otras muchas tecnologías del tipo CGI.

- **Eficiencia**

Con CGI tradicional, se inicia un nuevo proceso para cada requerimiento HTTP. Si el programa CGI hace una operación relativamente rápida, la sobrecarga de inicio del proceso puede dominar el tiempo de ejecución. Con los Servlets, la máquina Virtual Java permanece iniciada, y cada petición es manejada por un Thread Java de peso ligero, no un pesado proceso del sistema operativo. De forma similar, en CGI tradicional, si hay N peticiones simultáneas para el mismo programa CGI, el código de este proceso se cargará N veces en memoria. Sin embargo, con los Servlets, hay N Threads pero sólo una instancia de la clase Servlet. Adicionalmente los Servlets tienen más alternativas que los programas normales CGI para optimizaciones como los caches de cálculos previos, mantener abiertas las conexiones de bases de datos, etc.

- **Conveniencia**

Junto con la conveniencia de poder utilizar un lenguaje familiar y sencillo (Java), los Servlets tienen una gran infraestructura para análisis automático y decodificación de datos de formularios HTML, leer y seleccionar cabeceras HTTP, manejar cookies, seguimiento de sesiones, y muchas otras utilidades.

- **Potencia**

Los Servlets nos permiten fácilmente hacer muchas cosas que son difíciles o imposibles con CGI normal. Por algo, los servlets pueden hablar directamente con el servidor Web. Esto simplifica las operaciones que se necesitan para buscar imágenes y otros datos almacenados en situaciones estándares. Los Servlets también pueden compartir los datos entre ellos, haciendo las cosas útiles como almacenes de conexiones a bases de datos fáciles de implementar. También pueden mantener información de requerimiento en requerimiento, simplificando cosas como seguimiento de sesión y el caché de cálculos previos.

- **Portable**

Los Servlets están escritos en Java y siguen un API estandarizado. Consecuentemente, los servlets escritos, digamos en el servidor I-Planet Enterprise, se pueden ejecutar sin modificarse en Apache, Microsoft IIS o WebStar. Los Servlets están soportados directamente o mediante plug-in en la mayoría de los servidores Web.

- **Barato**

Hay un número de servidores Web gratuitos o muy baratos que son buenos para el uso "personal" o el uso empresarial. Sin embargo, la mayoría de los servidores Web comerciales son relativamente caros. Una vez que tengamos un servidor Web, no importa el coste del servidor, añadirle soporte para Servlets (si no viene pre-configurado para soportarlos) es gratuito o muy barato.

16.2.3. Utilizar Servlets en lugar de Scripts CGI!

Los Servlets son un reemplazo efectivo para los scripts CGI. Proporcionan una forma de generar documentos dinámicos que son fáciles de escribir y rápidos en ejecutarse. Los Servlets también solucionan el problema de hacer la programación del lado del servidor con APIs específicos de la plataforma: están desarrollados con el API Java Servlet, una extensión estándar de Java.

Por eso se utilizan los servlets para manejar peticiones de cliente HTTP. Por ejemplo, tener un servlet procesando datos POSTeados sobre HTTP utilizando un formulario HTML, incluyendo datos del pedido o de la tarjeta de crédito. Un servlet como este podría ser parte de un sistema de procesamiento de pedidos, trabajando con bases de datos de productos e inventarios, y quizás un sistema de pago on-line.

16.2.4. Otros usos de los Servlets

Tenemos algunos de los muchos usos para los servlets:

- *Permitir la colaboración entre las personas*

Un servlet puede manejar múltiples peticiones concurrentes, y puede sincronizarlas. Esto permite a los servlets soportar sistemas como conferencias on-line.

- *Reenviar peticiones*

Los Servlets pueden reenviar peticiones a otros servidores y servlets. Con esto los servlets pueden ser utilizados para balancear la carga desde varios servidores que reflejan el

mismo contenido, y para particionar un único servicio lógico en varios servidores, de acuerdo con los tipos de tareas compartidas o la organización.

16.3. Arquitectura del Paquete Servlet

El paquete `javax.servlet` proporciona clases e interfaces para escribir servlets. La arquitectura de este paquete se describe a continuación.

16.3.1. El Interfaz Servlet

La abstracción central en la API Servlet es la interfaz `Servlet`. Todos los servlets implementan este interfaz, bien directamente, o más comúnmente extendiendo una clase que lo implemente como `HttpServlet`.

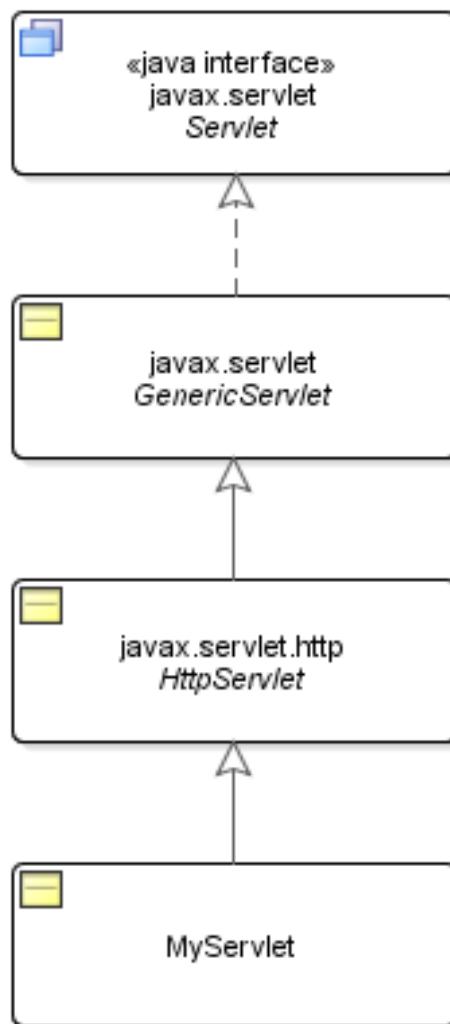


Figura 16 . 2 Estructura de la API Servlet

La interfaz `Servlet` declara, pero no implementa, métodos que manejan el Servlet y su comunicación con los clientes. Los desarrolladores de Servlets proporcionan algunos de esos métodos cuando programan un servlet.

16.3.2. Interacción con el Cliente

Cuando un servlet acepta una llamada de un cliente, recibe dos objetos.

- Un `ServletRequest`, que encapsula la comunicación desde el cliente al servidor.

- Un `ServletResponse`, que encapsula la comunicación de vuelta desde el servlet hacia el cliente.

`ServletRequest` y `ServletResponse` son interfaces definidas en el paquete `javax.servlet`.

16.3.3. La Interfaz `ServletRequest`

La Interfaz `ServletRequest` permite al servlet acceder a:

- Información como los nombres de los parámetros pasados por el cliente, el protocolo (esquema) que está siendo utilizado por el cliente, el nombre del host remoto que ha realizado la petición y el nombre del servidor que la ha recibido.
- El stream de entrada, `ServletInputStream`. Los Servlets utilizan este stream para obtener los datos desde los clientes que utilizan protocolos como los métodos http POST y GET.

Las interfaces que extienden la interfaz `ServletRequest` permiten al servlet recibir datos específicos del protocolo. Por ejemplo, la interfaz `HttpServletRequest` contiene métodos para acceder a información específica de la cabecera HTTP.

16.3.4. La Interfaz `ServletResponse`

La Interfaz `ServletResponse` le da al servlet los métodos para responder al cliente:

- Permite al servlet seleccionar la longitud del contenido y el tipo MIME de la respuesta.
- Proporciona un stream de salida, `ServletOutputStream`, y un `Writer` a través del cual el servlet puede responder datos.

Las interfaces que extienden la interfaz `ServletResponse` le dan a los servlets más capacidades específicas del protocolo. Por ejemplo, el interfaz `HttpServletResponse` contiene métodos que permiten al servlet manipular información específica de cabecera HTTP.

16.3.5. Capacidades Adicionales de los Servlets http

Las clases e interfaces descritas anteriormente construyen un servlet básico. Los servlets HTTP tienen algunos objetos adicionales que proporcionan capacidades de seguimiento de sesión. El desarrollador se servlets pueden utilizar esos APIs para mantener el estado entre el servlet y el cliente a través de múltiples conexiones durante un periodo de tiempo. Los servlets HTTP también tienen objetos que proporcionan cookies. El API cookie se utiliza para guardar datos dentro del cliente y recuperar posteriormente esos datos.

16.4.- Un Servlet Sencillo

16.4.1. Creación de un Servlet Sencillo

Paso 01

Crear un proyecto de tipo **Web Application** de nombre **Cap16**.

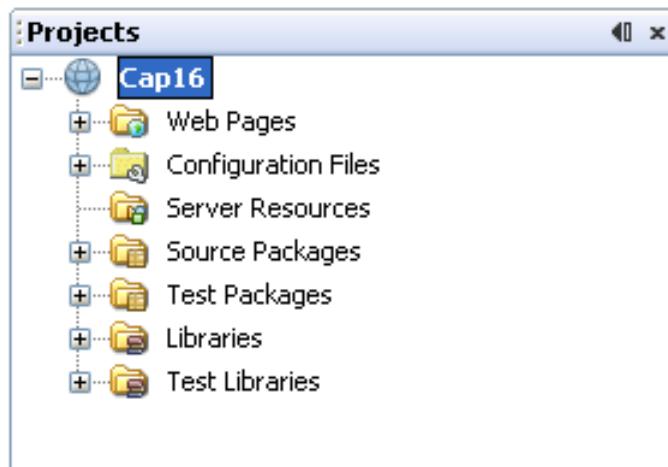


Figura 16 . 3 Estructura de un proyecto web.

Paso 02

Proceda a crear un servlet de nombre **SimpleServlet** en el paquete **servlets**.

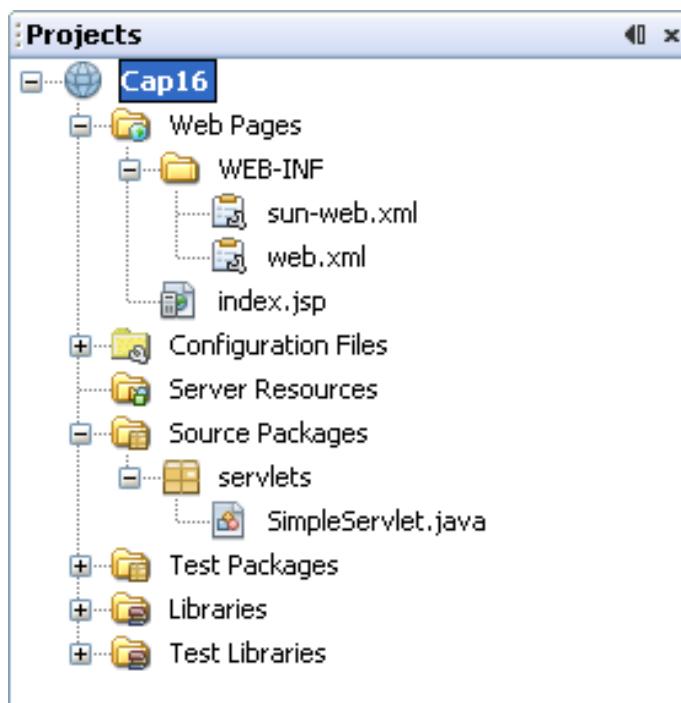


Figura 16 . 4 Ubicación del servlet **SimpleServlet** en la estructura del proyecto.

Paso 03

En el archivo **web.xml** verifique:

- La definición del servlet

```
<servlet>
    <servlet-name>SimpleServlet</servlet-name>
    <servlet-class>servlets.SimpleServlet</servlet-class>
</servlet>
```

- El mapeo del servlet

```
<servlet-mapping>
    <servlet-name>SimpleServlet</servlet-name>
    <url-pattern>/SimpleServlet</url-pattern>
</servlet-mapping>
```

Paso 04

A continuación tenemos el script del servlet.

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SimpleServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet SimpleServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<p>Esta es la salida desde SimpleServlet.</p>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
throws ServletException, IOException {  
    processRequest(request, response);  
}  
  
protected void doPost(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
    processRequest(request, response);  
}  
  
public String getServletInfo() {  
    return "Short description";  
}// </editor-fold>  
  
}
```

16.4.2. Descripción del Servlet

- **SimpleServlet** extiende la clase `HttpServlet`, que implementa el interfaz `Servlet`.
- **SimpleServlet** sobre-escribe el método `doGet` y `doPost` de la clase `HttpServlet`. Estos métodos son invocados cuando un cliente hace un requerimiento `GET` (el método de requerimiento por defecto de HTTP) y `POST` respectivamente.
- Los métodos `doGet` y `doPost` invocan al método `processRequest`, donde se genera una sencilla página HTML devuelta al cliente.
- Dentro del método `processRequest`
 - La petición del usuario está representada por un objeto `HttpServletRequest`.
 - La respuesta al usuario está representada por un objeto `HttpServletResponse`.
 - Como el texto es devuelto al cliente, la respuesta se envía utilizando el objeto `Writer` obtenido desde el objeto `HttpServletResponse`.

Nota

El método `processRequest` lo genera automáticamente el asistente de NetBeans, no todos los IDEs generan un método para centralizar los requerimientos de los usuarios, se podría programar solamente `doGet` ó `doPost` dependiendo del tipo de llamada del cliente.

16.4.3. Ejecución del Servlet

Para ejecutar el servlet hacemos clic con el botón derecho del Mouse sobre el archivo **SimpleServlet.java** y ejecutamos el comando **Run File**.

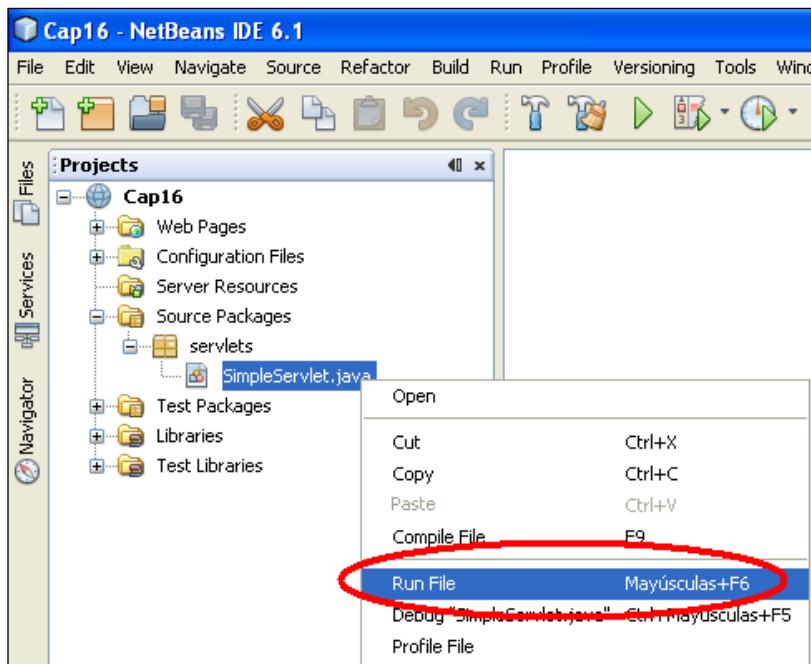


Figura 16 . 5 Ejecución del servlet SimpleServlet.

A continuación tenemos el resultado de su ejecución:

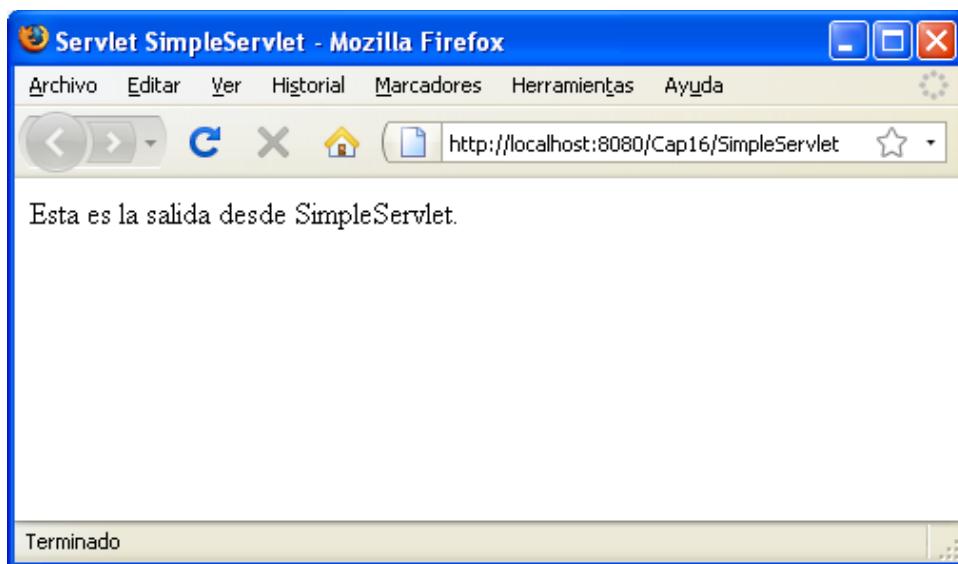


Figura 16 . 6 Salida de la ejecución de SimpleServlet.

16.5.- Interacción con los Clientes

Un Servlet HTTP maneja peticiones de los clientes a través de su método `service`. Este método soporta peticiones estándar de clientes HTTP enviando cada requerimiento a un método designado para manejar esa petición. Por ejemplo, el método `service` llama al método `doGet` mostrado anteriormente en el ejemplo del servlet sencillo.

16.5.1. Requerimientos y Respuestas

Esta sección explica el uso de los objetos que representan requerimientos de los clientes (un objeto `HttpServletRequest`) y las respuestas del servlet (un objeto `HttpServletResponse`). Estos

objetos se proporcionan al método `service` y a los métodos que `service` llama para manejar peticiones HTTP.

16.5.2. Manejar Requerimientos GET y POST

Los métodos a los que delega el método `service` las peticiones HTTP, incluyen:

- `doGet`, para manejar requerimientos GET, GET condicional, y requerimientos de HEAD.
- `doPost`, para manejar requerimientos POST.
- `doPut`, para manejar peticiones PUT.
- `doDelete`, para manejar peticiones DELETE.

Por defecto, estos métodos devuelven un error `BAD_REQUEST` (400). Nuestro servlet debería sobrescribir el método o métodos diseñados para manejar las interacciones HTTP que soporta. Esta sección muestra cómo implementar métodos para manejar los requerimientos HTTP más comunes: GET y POST.

El método `service` de `HttpServlet` también llama al método `doOptions` cuando el servlet recibe un requerimiento `OPTIONS`, y a `doTrace` cuando recibe un requerimiento `TRACE`. La implementación por defecto de `doOptions` determina automáticamente que opciones HTTP son soportadas y devuelve esa información. La implementación por defecto de `doTrace` realiza una respuesta con un mensaje que contiene todas las cabeceras enviadas en el requerimiento `TRACE`. Estos métodos normalmente no se sobre-escriben.

Adicionalmente, el asistente de NetBeans para crear servlets crea el método `processRequest` que es invocado desde los métodos `doGet` y `doPost`.

16.5.3. Problemas con los Threads

Los Servlets HTTP normalmente pueden servir a múltiples clientes concurrentes. Si los métodos de nuestro Servlet no funcionan con clientes que acceden a recursos compartidos, debemos:

- Sincronizar el acceso a estos recursos, o
- Crear un servlet que maneje sólo una petición de cliente a la vez.

En este capítulo solo implementaremos la segunda opción.

Ejemplo 16 . 1

En este ejemplo desarrollaremos un formulario que permita ingresar dos números, estos números serán enviados a un servlet, y el servlet retornará la suma de estos dos números.

Los elementos que construiremos son los siguientes:

Elemento	Nombre
Proyecto	Cap16Ejemplo01
Página HTML	suma.html
Servlet	Suma.java
Página de inicio	index.jsp

Paso 01

Crear el proyecto web Cap16Ejemplo01.

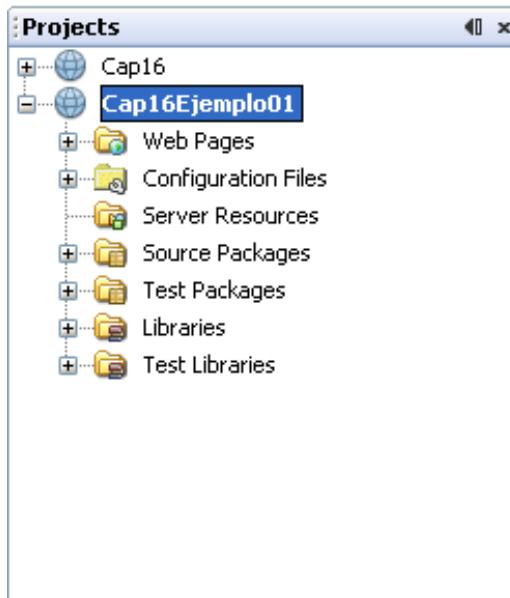


Figura 16 . 7 Estructura del proyecto Cap16Ejemplo01.

Paso 02

Crear el servlet **Suma.java** en el paquete **servlets**, la programación del método `processRequest` es el siguiente:

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        // Captura los datos
        int n1 = Integer.parseInt(request.getParameter("num1"));
        int n2 = Integer.parseInt(request.getParameter("num2"));
        // Proceso
        int s = n1 + n2;
        // Respuesta al cliente
        out.println("<html>");
        out.println("  <head><title>Suma de Dos Números</title></head>");
        out.println("  <body>");
        out.println("    <h1>Suma de Dos N&uacute;meros</h1>");
        out.println("    <p>Número 1 = " + n1 + "</p>");
        out.println("    <p>Número 2 = " + n2 + "</p>");
        out.println("    <p>Suma = " + s + "</p>");
        out.println("  </body>");
        out.println("</html>");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
        out.println("</body></html>");

        out.close();

    } finally {

        out.close();

    }

}
```

Paso 03

Crear la página web suma.html, en esta página crearemos el formulario para que el usuario ingrese los números a procesar. A continuación tenemos el script:

```
<html>

<head>

    <title>Cap16Ejemplo01</title>

    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

</head>

<body>

    <h1>Suma de Dos Números</h1>

    <form action="Suma" method="post">

        <table width="256">

            <tr>

                <td width="85">Número 1</td>

                <td width="58">

                    <input type="text" name="num1" maxlength="5" size="8" id="num1"/>

                </td>

                <td width="85">&ampnbsp</td>

            </tr>

            <tr>

                <td>Número 2</td>

                <td><input type="text" name="num2" maxlength="5" size="8" id="num2"/></td>

                <td><input name="procesar" type="submit" id="procesar" value="Procesar"/></td>

            </tr>

        </table>

    </form>

</body>

</html>
```

Paso 04

La página index.jsp es la que se ejecuta por defecto, haremos que esta página realice forward hacia la página suma.html. La única línea que debe tener esta página es la siguiente:

```
<jsp:forward page="suma.html"/>
```

Paso 05

Finalmente, estamos listos para ejecutar el proyecto y lo primero que obtenemos es el formulario de ingreso de datos, tal como se observa en la Figura 16.8.

Figura 16 . 8 Formulario de ingreso de datos.

Después de ingresar dos números y hacer clic en el botón **Procesar** obtenemos el resultado del servlet, tal como se ilustra en la Figura 16.9.

Figura 16 . 9 Resultado del servlet.

Si quiere probar otra suma haga clic en el enlace **Otra Suma**.

16.5.4. Servlet Recursivo

El mismo servlet puede presentar el formulario y a la vez procesar los datos del mismo, esta posibilidad hace que el servlet se le llame recursivo.

Ejemplo 16 . 2

En este ejemplo veremos el mismo caso del Ejemplo 16.1, pero a través de un servlet recursivo.

Los elementos que construiremos son los siguientes:

Elemento	Nombre
Proyecto	Cap16Ejemplo02
Servlet	Suma.java
Pagina de inicio	index.jsp

Paso 01

Crear el proyecto Cap16Ejemplo02.

Paso02

Proceda a crear el servlet **Suma.java** en el paquete **servlets**. Este servlet manejará 3 estados:

- **FORMULARIO**, en este estado se mostrará el formulario para que el usuario ingrese los números a sumar.
- **RESPUESTA**, en este estado el servlet mostrará la respuesta obtenida después de sumar los dos números ingresados en el formulario y haber hecho clic en el botón **Procesar**.
- **ERROR**, en este estado se mostrará el mensaje obtenido en caso exista algún error en el proceso, por ejemplo cuando los datos ingresados no sean números.

El script de la clase completa es la siguiente:

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author Gustavo Coronel
 */
public class Suma extends HttpServlet {

    // Estados
    enum Estados { FORMULARIO, RESPUESTA, ERROR };

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        // Proceso General
    }

}
```

```

Estados estado = Estados.FORMULARIO;
int n1 = 0;
int n2 = 0;
int suma = 0;
String msgError = "";
try {
    if (request.getParameterMap().size() > 0) {
        estado = Estados.RESPUESTA;
        // Datos
        n1 = Integer.parseInt(request.getParameter("num1"));
        n2 = Integer.parseInt(request.getParameter("num2"));
        // Proceso
        suma = n1 + n2;
    }
} catch (Exception e) {
    estado = Estados.ERROR;
    msgError = e.getMessage();
}
// Reporte al usuario
out.println("<html>");
out.println("<head><title>Cap16Ejemplo02</title></head>");
out.println("<body>");
out.println("<h1>Suma de Dos N&uacute;meros</h1>");
if (estado == Estados.FORMULARIO) {
    out.println("<form action='Suma' method='post'>");
    out.println("<table width='256'>");
    out.println("<tr>");
    out.println("<td width='85'>Número 1</td>");
    out.println("<td width='58'>");
    out.println("<input type='text' name='num1' maxlength='5' size='8' id='num1' />");
    out.println("</td>");
    out.println("<td width='85'>&ampnbsp</td>");
    out.println("</tr>");
    out.println("<tr>");
    out.println("<td>Número 2</td>");
    out.println("<td>");
    out.println("<input type='text' name='num2' maxlength='5' size='8' id='num2' />");
    out.println("</td>");
    out.println("<td>");
}

```

```
out.println("<input name='procesar' type='submit' id='procesar' value='Procesar' />");  
out.println("</td>");  
out.println("</tr>");  
out.println("</table>");  
out.println("</form>");  
} else if (estado == Estados.RESPUUESTA) {  
    out.println("<p>Número 1 = " + n1 + "</p>");  
    out.println("<p>Número 2 = " + n2 + "</p>");  
    out.println("<p>Suma = " + suma + "</p>");  
    out.println("<A HREF='Suma'>Otra Suma</A>");  
} else {  
    out.println("<h3>Error</h3>");  
    out.println("<p>Dotos no son correctos.</p>");  
    out.println("<p>Mensaje: " + msgError + "</p>");  
    out.println("<A HREF='Suma'>Otra Suma</A>");  
}  
out.println("</body></html>");  
out.close();  
}  
  
protected void doGet(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
    processRequest(request, response);  
}  
  
protected void doPost(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
    processRequest(request, response);  
}  
  
public String getServletInfo() {  
    return "Servlet para sumar dos números";  
}
```

Paso 03

La página index.jsp es la que se ejecuta por defecto, haremos que esta página realice forward hacia el servlet Suma. La única línea que debe tener esta página es la siguiente:

```
<jsp:forward page="Suma"/>
```

Paso 04

Finalmente, estamos listos para ejecutar el proyecto y lo primero que obtenemos es el formulario de ingreso de datos, tal como se observa en la Figura 16.10.

The screenshot shows a Mozilla Firefox window with the title bar 'Cap16Ejemplo02 - Mozilla Firefox'. The address bar displays 'http://localhost:8080/Cap16Ejemplo02/'. The main content area has a heading 'Suma de Dos Números'. Below it, there are two input fields labeled 'Número 1' and 'Número 2', followed by a button labeled 'Procesar'. At the bottom of the page, there is a status bar with the text 'Terminado'.

Figura 16 . 10 Formulario que presenta el servlet.

Después de ingresar dos números, por ejemplo 15 y 23, y hacer clic en el botón **Procesar** obtenemos el resultado que se muestra en la Figura 16.11.

The screenshot shows a Mozilla Firefox window with the title bar 'Cap16Ejemplo02 - Mozilla Firefox'. The address bar displays 'http://localhost:8080/Cap16Ejemplo02/Suma'. The main content area has a heading 'Suma de Dos Números'. Below it, the text 'Numero 1 = 15', 'Numero 2 = 23', and 'Suma = 38' are displayed. There is also a link 'Otra Suma'. At the bottom of the page, there is a status bar with the text 'Terminado'.

Figura 16 . 11 Resultado que muestra el servlet.

En caso de ingresar datos incorrectos obtendremos el mensaje error similar al que se muestra en la Figura 16.12.

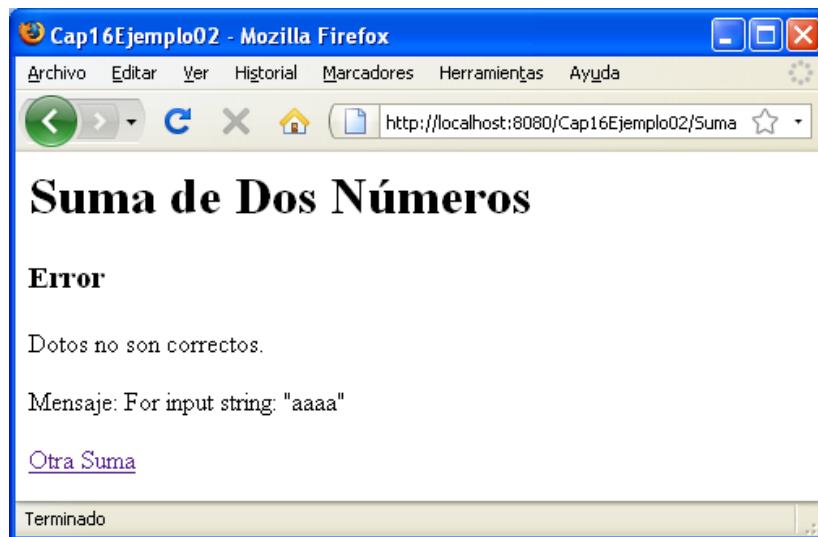


Figura 16 . 12 Mensaje de error mostrado por el servlet.

16.6.- Programación de Servlets

16.6.1. Programación

Todo servlet implementa la interfaz `Servlet` del paquete `javax.servlet`, no directamente, sino heredando de la clase `GenericServlet`, o heredando de la clase `HttpServlet`, que es subclase de `GenericServlet`, tal como se ilustra en la Figura 16.13.



Figura 16 . 13 Diagrama de clases base de los servlets.

Los servlets que se estudian en este libro son los que admiten peticiones basadas en el protocolo HTTP.

Un servlet HTTP se construirá heredando de la clase `HttpServlet` y seguirá un modelo de ejecución basado en un ciclo de vida similar al de los applets y ejecutado por el servidor JEE donde se hospeda. Como esta clase implementa la interfaz citada anteriormente, todo servlet va a poder usar directamente los métodos de esta interfaz, los de la propia clase y los de su superclase `GenericServlet`.

La interfaz `Servlet` declara los métodos del ciclo de vida de un servlet, que son:

- `void init(ServletConfig config)`: es invocado una sola vez, por el contenedor del servidor JEE compatible donde se hospeda el servlet y se emplea para inicializarlo. Se ejecuta cuando se realiza el primer requerimiento del servlet. Este método y el resto del ciclo de vida se tratarán con más detalle en el siguiente apartado.
- `void destroy()`: es invocado por el contenedor antes de que el servlet se descargue de memoria y deje de prestar servicio.
- `void service(ServletRequest request, ServletResponse response)`: es invocado por el contenedor para procesar el requerimiento, una vez que el servlet se ha inicializado (equivale a ejecutar el `init`). Es el llamado método de servicio. Sus argumentos son instancias de las interfaces `javax.servlet.ServletRequest` y `javax.servlet.ServletResponse` que modelan, respectivamente, el requerimiento del cliente y la respuesta del servlet. En un servlet HTTP, este método suele sustituirse por los métodos de `javax.servlet.http.HttpServlet`.
 - `void doPost(HttpServletRequest request, HttpServletResponse response)` para gestionar requerimientos tipo POST.
 - `void doGet(HttpServletRequest request, HttpServletResponse response)` para gestionar requerimientos tipo GET.

Se realiza un requerimiento tipo GET a un recurso (un servlet, en nuestro caso) cuando se lleva a cabo una de las siguientes acciones:

- Se digita en la barra de direcciones del navegador la URL del recurso.
- Se hace clic en un enlace HTML que apunte al recurso.
- Enviar al recurso un formulario HTML cuyo atributo `method` de la etiqueta `<form>` tenga el valor de GET:

```
<form action="..." method="get">
```

También es válido:

```
<form action="...">
```

Ya que, si no se especifica `method`, este atributo asume el valor GET.

En cambio, se realiza un requerimiento de tipo POST si se envía al recurso un formulario HTML cuyo atributo `method` de la etiqueta `<form>` es POST.

Es decir:

```
<form action="..." method="post">
```

Por tanto, un servlet se crea heredando de:

- `GenericServlet`: si admite requerimientos no basadas en ningún protocolo especial. Empleado cuando se programan servlets que se ejecutan en servidores de correo, noticias, ficheros, etc. Poco habitual.
- `HttpServlet`: es una subclase de la anterior, si admite requerimientos de clientes basadas en el protocolo HTTP. Es el modo de crear los servlets que se va a estudiar.

La declaración de la clase asociada a un servlet HTTP genérico sería:

```

import javax.servlet.*;
import javax.servlet.http.*;

public class Suma extends HttpServlet {

    ...
    ...
    ...

}

}

```

Nota

Si utiliza NetBeans, su asistente para la creación de servlets los registra automáticamente en el archivo web.xml y le agrega el método **processRequest** que invocado desde los métodos **doPost** y **doGet**.

16.6.2. Esquema de funcionamiento

La Figura 16.14 muestra el esquema de funcionamiento de un servlet.

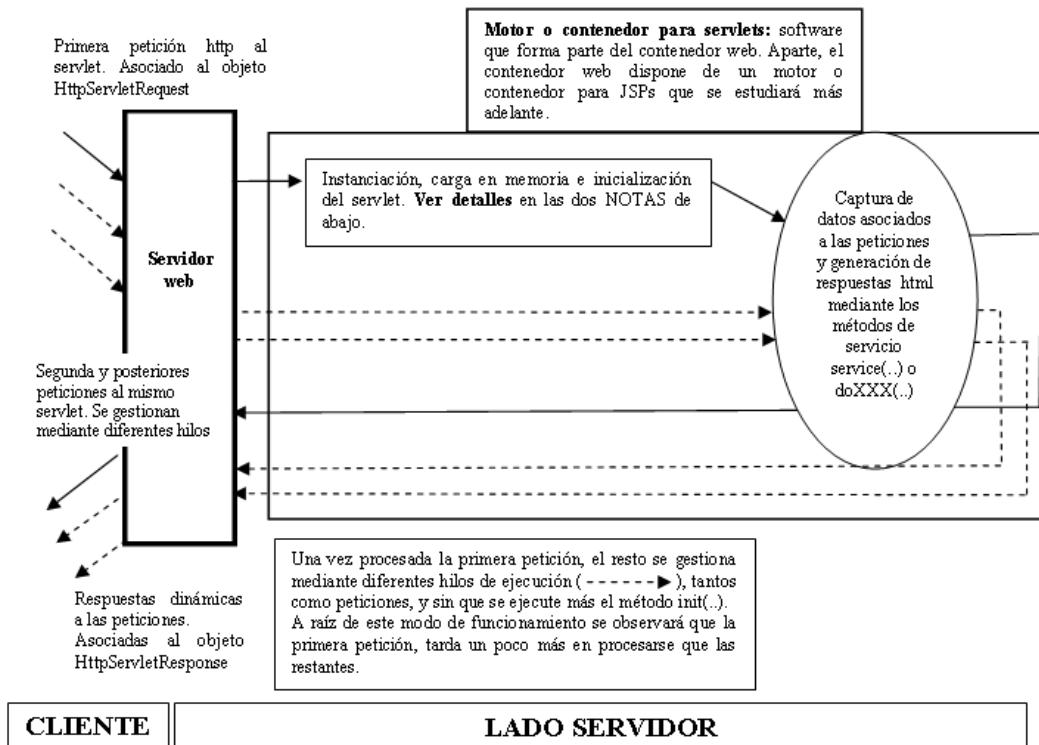


Figura 16 . 14 Esquema de funcionamiento de un servlet.

El funcionamiento se puede resumir de la siguiente manera:

1. Cuando el servlet es requerido por primera vez se instancia, carga en memoria e inicializa.
2. A partir del segundo requerimiento la gestión del servlet es mediante hilos (threads).

16.6.2.1. Instanciación de un Servlet

La instancia y carga en memoria se produce tras levantar el servidor y leer el descriptor de despliegue web.xml de la aplicación a la que pertenece. Implica la creación, por parte del contenedor, de un objeto servlet mediante el método estático de java.lang.Class, forName(String s) y el método newInstance() de Class.

Como argumento del método forName(..) se utiliza el nombre de la clase obtenida tras compilar el servlet. Esta información se obtiene del valor especificado en la etiqueta XML <servlet> del descriptor de despliegue. Por cada elemento <servlet> se instancia un objeto servlet diferente.

Por ejemplo, a continuación se muestra el descriptor de despliegue web.xml de una aplicación que contiene un servlet.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <servlet>
    <servlet-name>SimpleServlet</servlet-name>
    <servlet-class>servlets.SimpleServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>SimpleServlet</servlet-name>
    <url-pattern>/SimpleServlet</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

16.6.2.2. Nota sobre Inicialización de un Servlet

Tras la instancia y carga en memoria del servlet, viene su inicialización. Comienza cuando se realiza la primera petición al servlet. El proceso está regido por su ciclo de vida, que se basa en los métodos de la interfaz javax.servlet..Servlet:

- init(ServletConfig config)
- service(ServletRequest request, ServletResponse response)
- destroy()

La inicialización implica la ejecución ÚNICA, por parte del contenedor, del método `init(ServletConfig config)`. Se produce cuando se realiza el primer requerimiento. El objeto `javax.servlet.ServletConfig` es creado por el contenedor durante la carga en memoria y “recoge” información sobre el servlet, previa lectura del descriptor de despliegue. Este método no es necesario escribirlo (mejor dicho, redefinirlo) en el código del servlet.

Lo habitual, si se opta por hacerlo, es redefinir el método `init()` de la clase `GenericServlet`. El contenedor garantiza que después de invocar al método `init(ServletConfig config)`, invocará al método `init()`. **Modo de trabajo habitual.** Para obtener el objeto `ServletConfig`, se utilizará el método `getServletConfig()` de `GenericServlet`.

Si se redefine el `init(ServletConfig config)`, `config` estará directamente disponible en el cuerpo del método pero, si se desea utilizar algún método de la clase `GenericServlet`, debe incluirse como primera línea `super.init(config)`, con el fin de pasarle a `GenericServlet` la interfaz `ServletConfig` (recordar que la implementa por construcción). Si no se hace esto, se produce un error de ejecución, lanzándose una excepción de tipo `NullPointerException` en la línea donde se intenta emplear el método de `GenericServlet`.

Tras la inicialización, el servlet ya está disponible para recibir requerimientos, gestionarlos y generar una respuesta, con el método de su ciclo de vida `service(ServletRequest request, ServletResponse response)`. Los objetos asociados a los argumentos del método `service` son creados también por el contenedor.

Cuando se realiza un requerimiento a un servlet, el contenedor invoca al método `service(ServletRequest request, ServletResponse response)` de `javax.servlet.http.HttpServlet`, que invoca al `service(HttpServletRequest request, HttpServletResponse response)` de la misma clase (está sobrecargado). Este método `service`, en función del tipo del requerimiento, delega automáticamente su procesamiento al `doXXX(..)` correspondiente. Si el requerimiento de tipo, a `doPost(..)`, si es de tipo GET a `doGET(..)`, etc.

Si se redefine en el código del servlet cualquiera de los métodos `service(..)` anteriores, debe invocarse explícitamente al método `doXXX(..)` correspondiente, o procesar el requerimiento usando sólo el método `service` redefinido.

Lo habitual es no redefinir el método `service(..)` y trabajar con los métodos `doXXX(..)`. **Modo de trabajo habitual.**

Para el caso específico de que se encuentre trabajando con NetBeans, como lo es en este libro, el asistente de creación de servlets registra automáticamente el servlet en el archivo `web.xml` y crea el método `processRequest` que es invocado de los métodos `doPost()` y `doGet()`.

16.6.2.3. Explicación Resumida del Proceso

Una vez levantado el servidor, se carga la clase asociada al servlet, se instancia invocando a su constructor sin argumentos por defecto y se carga en memoria. Cuando se realiza el primer requerimiento se inicializa, esta inicialización es única e implica la ejecución, por parte del contenedor, del método `init(ServletConfig config)` del ciclo de vida del servlet.

Este método no suele aparecer en el código del servlet. Lo habitual es redefinir el método `init()` y emplear el método `getServletConfig()` de `GenericServlet` para obtener un objeto `ServletConfig`.

Finalizada la inicialización, el servlet ya está disponible para procesar los requerimientos y generar una respuesta a los mismos, con el método `service(ServletRequest request, ServletResponse response)`.

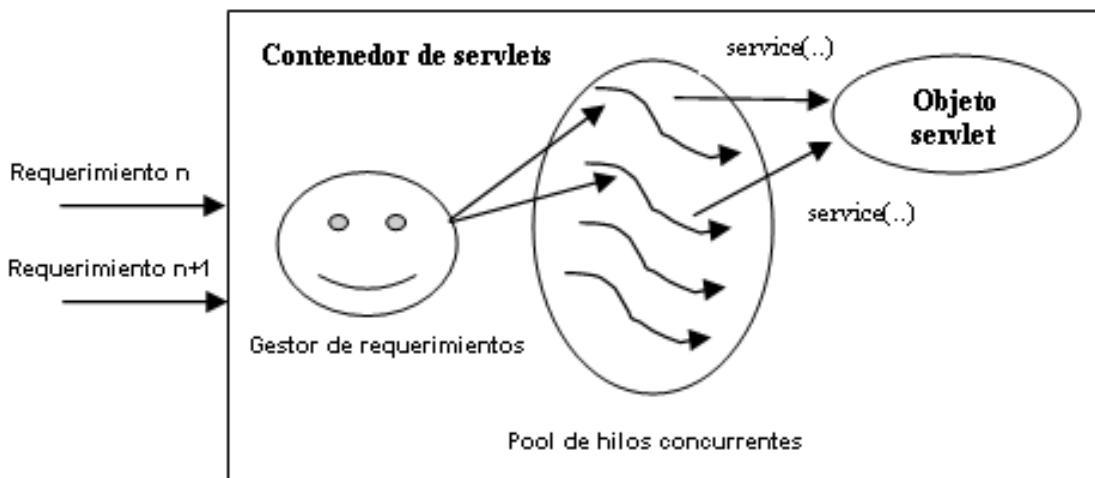


Figura 16 . 15 Gestión de múltiples requerimientos mediante hilos.

Una vez procesado el primer requerimiento, el resto de requerimientos se gestiona mediante diferentes hilos de ejecución, tantos como requerimientos existan, tal como se puede apreciar en la Figura 16.15, y sin que se ejecute más el método `init(..)`.

CONCLUSIÓN

Un objeto servlet en memoria procesa múltiples requerimientos simultáneamente mediante hilos.

16.6.2.4. Respeto al Método `destroy()` del Ciclo de Vida

Conviene decir lo siguiente: si se cae el servidor, o se sobrepasa un tiempo límite de inactividad del servlet, o si el servidor está trabajando con insuficiente memoria, el contenedor podría eliminar el objeto servlet de la memoria. Su ejecución es competencia del contenedor.

Antes de que suceda, se garantiza la ejecución del método `destroy()`. No es muy habitual redefinirlo en el código del servlet, pero si se hace, su código típicamente se compondrá de líneas que permitan liberar los recursos empleados por el servlet. Ejemplo: cierre de conexiones a bases de datos, vaciado y cierre de flujos de lectura y escritura, etc.

Ejemplo 16 . 3

En este ejemplo ilustrare el uso de los métodos `init` y `destroy` de un servlet con acceso a la base de datos EurekaBank de MySQL, utilizando JDBC.

Se trata de hacer un programa que permita consultar las cuentas de una sucursal. Los elementos que construiremos son los siguientes:

Elemento	Nombre	Descripción
Proyecto	Cap16Ejemplo03	Nombre del proyecto.
Página HTML	sucursal.html	Página que contiene un formulario para ingresar el código de la sucursal a consultar.
Servlet	Cuentas.java	Servlet que recibe el código de la sucursal y muestra un listado de las cuentas que le pertenecen.
Página de inicio	index.jsp	Página que inicia la ejecución del proyecto.

Paso 01

Como primer paso crearemos el proyecto Cap16Ejemplo03, la estructura del proyecto se muestra en la Figura 16.16.

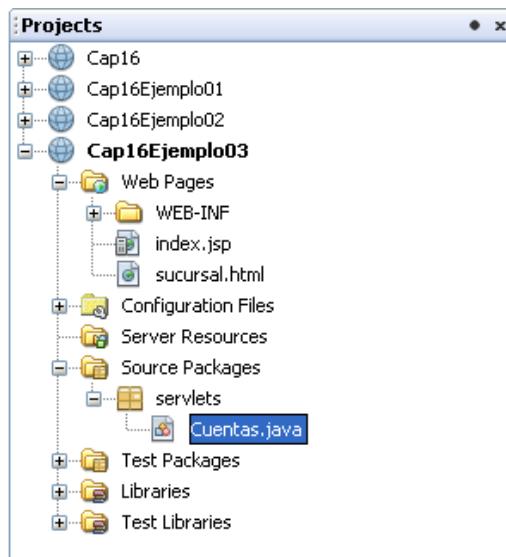


Figura 16 . 16 Estructura del proyecto Cap16Ejemplo03.

Paso 02

En este proyecto accederemos a la base de datos EurekaBank, por lo tanto será necesario cargar la librería para MySQL, esta tarea la realiza en la ventana de propiedades del proyecto, en la categorías **Librerías**, tal como se muestra en la Figura 16.17.

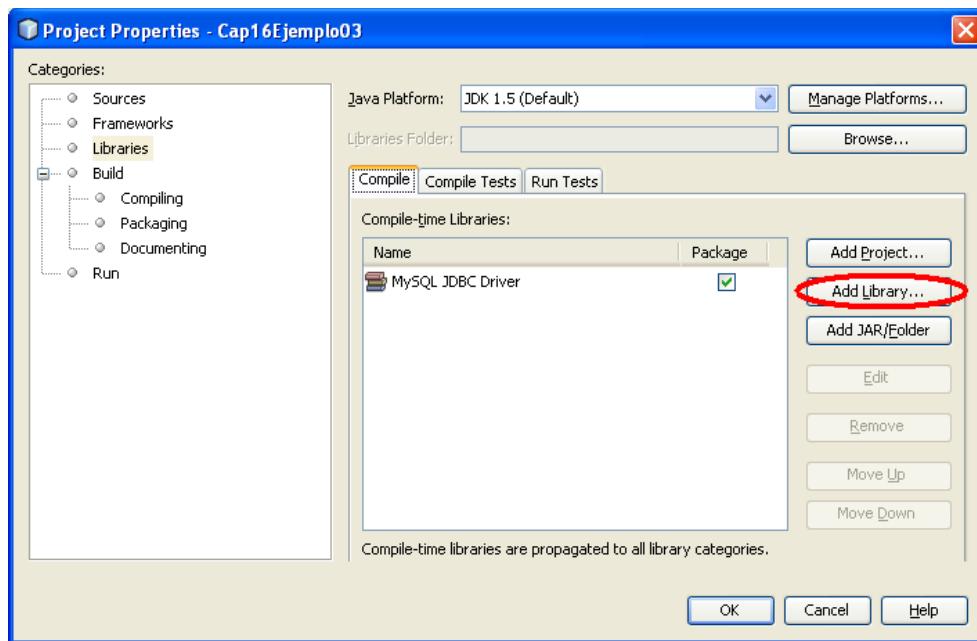


Figura 16 . 17 Cargar la librería JDBC para MySQL.

Paso 03

Procederemos a crear el documento **sucursal.html**, en este documento crearemos el formulario para el ingreso del código de la sucursal de la que queremos consultar sus cuentas, el script es el siguiente:

```

<html>
    <head>
        <title>Consultar Cuentas</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>EurekaBank</h1>
        <form name="form1" method="post" action="Cuentas">
            <table width="255">
                <tr>
                    <td width="68">
                        Sucursal
                    </td>
                    <td width="47">
                        <input name="codigo" type="text" id="codigo" size="5" maxlength="3">
                    </td>
                    <td width="124">
                        <input type="submit" name="consultar" id="consultar" value="Consultar">
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>

```

Paso 04

Es hora de programar el servlet, debemos tener en cuenta que la conexión con la base de datos se debe programar en el método `init()` y la desconexión en el método `destroy()`.

Se necesita una variable que represente la conexión con la base de datos y que se defina en la clase con alcance de instancia, a esta variable la llamaremos `cn` y será de tipo `java.sql.Connection`; crearemos un método privado de nombre `crearConexion()` que se encargara de hacer la conexión con la base de datos, este método se ejecutará desde el método `init()`.

El script del servlet es el siguiente:

```

package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

```

```
import java.sql.ResultSet;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author Gustavo Coronel
 */
public class Cuentas extends HttpServlet {

    private Connection cn = null;
    ArrayList<String> listaErrores = new ArrayList<String>();

    private void crearConexion() {
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            this.cn = DriverManager.getConnection("jdbc:mysql://localhost/eurekabank", "root", "");
        } catch (Exception e) {
            this.listaErrores.add("Error en la conexión con la base de datos");
            this.listaErrores.add(e.getMessage());
        }
    }

    @Override
    public void init() throws ServletException {
        super.init();
        this.crearConexion();
    }

    @Override
    public void destroy() {
        super.destroy();
        try {
            if (this.cn != null) {
                this.cn.close();
            }
        }
    }
}
```

```

} catch (Exception e) {
    this.cn = null;
}
}

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    // Variables
    boolean hayError = false;
    PreparedStatement pst = null;
    ResultSet rs = null;
    String sucursal = null;
    String sql = "select * from cuenta where chr_sucucodigo = ?";
    String datos = "";
    // Dato
    sucursal = request.getParameter("codigo");
    // Proceso
    try {
        pst = this.cn.prepareStatement(sql);
        pst.setString(1, sucursal);
        rs = pst.executeQuery();
        while (rs.next()) {
            String cuenta = rs.getString("chr_cuencodigo");
            String cliente = rs.getString("chr_cliecodigo");
            Double saldo = rs.getDouble("dec_cuensaldo");
            datos += "<tr>";
            datos += "<td>" + cuenta + "</td>";
            datos += "<td>" + cliente + "</td>";
            datos += "<td>" + saldo + "</td>";
            datos += "</tr>";
        }
    } catch (Exception e) {
        hayError = true;
        this.listaErrores.add("Error en el proceso");
        this.listaErrores.add(e.getMessage());
    }
    if (datos.length() == 0) {

```

```
        datos = "<p>No hay cuentas para la sucursal " + sucursal + "</p>";
    } else {
        datos = "<table border='1' width='200'>" +
            "<tr>" +
            "<td>Cuenta</td>" +
            "<td>Cliente</td>" +
            "<td>Saldo</td>" +
            "</tr>" +
            datos +
            "</table>";
    }

    // Reporte
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Consulta de cuentas</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>EurekaBank</h1>");

    if (hayError) {
        String errores = "";
        for (String error : listaErrores) {
            errores += error + "<br>";
        }
        out.println("<p>ERRORES</p>");
        out.println("<p>" + errores + "<p>");
    } else {
        out.println(datos);
    }

    out.println("<a href='sucursal.html'>Otra Consulta</a>");
    out.println("</body>");
    out.println("</html>");
    out.close();
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
```

```

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

@Override
public String getServletInfo() {
    return "Consulta de Cuentas.";
} // </editor-fold>
}

```

Paso 05

La página index.jsp es la que se ejecuta por defecto, haremos que esta página realice forward hacia la página sucursal.html. La única línea que debe tener esta página es la siguiente:

```
<jsp:forward page="sucursal.html"/>
```

Paso 06

Antes de ejecutar el proyecto consultemos las conexiones activas:

```

mysql> show processlist;
+----+-----+-----+-----+-----+-----+-----+
| Id | User | Host      | db   | Command | Time | State  | Info
+----+-----+-----+-----+-----+-----+-----+
| 1  | root | localhost:1773 | NULL | Query   | 0    | NULL   | show processlist |
+----+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)

mysql>

```

Ejecutamos el proyecto, y obtenemos el formulario para ingresar el código de la sucursal que queremos consultar, tal como se ilustra en la Figura 16.18.

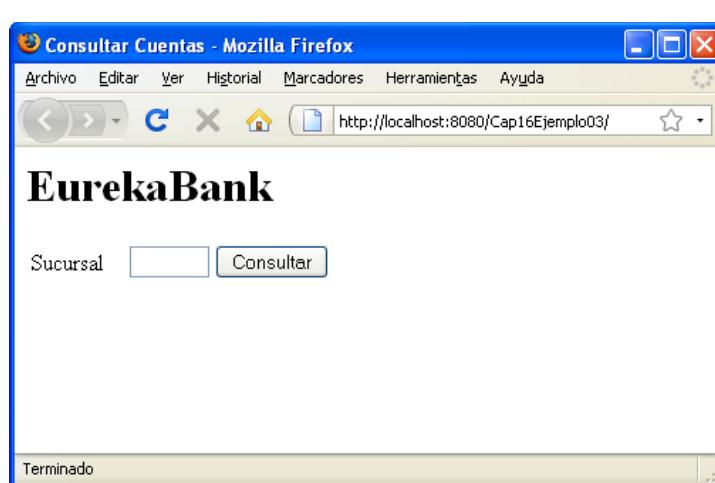


Figura 16 . 18 Formulario de ingreso del código de la sucursal a consultar.

Si ingresamos por ejemplo el código 001 y hacemos clic en el botón **Consultar** obtenemos el resultado que se muestra en la Figura 16.19.



Figura 16 . 19 Resultado de la consulta de una sucursal.

En caso de ingresar un código de sucursal que no existe o que no tenga ninguna cuenta se muestra un mensaje, tal como se ilustra en la Figura 16.20.



Figura 16 . 20 Mensaje cuando la sucursal no existe ó no tiene sucursales.

Consultemos nuevamente las conexiones activas:

```
mysql> show processlist;
+----+----+-----+----+----+-----+----+----+
| Id | User | Host      | db   | Command | Time | State | Info          |
+----+----+-----+----+----+-----+----+----+
| 1 | root | localhost:1773 | NULL | Query   | 0    | NULL  | show processlist |
| 2 | root | localhost:1868 | eurekabank | Sleep  | 4    | NULL  |                         |
+----+----+-----+----+----+-----+----+----+
2 rows in set (0.00 sec)
```

```
mysql>
```

Finalmente, pienso que esta no es la mejor manera de programar el acceso a la base de datos, algunas de las razones son las siguientes:

1. Cada servlet estaría consumiendo una conexión a la base de datos.
2. Estaríamos juntando en una sola clase la lógica de control de la aplicación, la lógica del negocio y la capa de acceso a la base de datos, y de lo que se trata es de desacoplar estas capas para un mejor control, desarrollo y reutilización de componentes.

3. Para la capa de acceso a la fuente de datos deberíamos utilizar el patrón de diseño DAO y si tenemos varias fuentes de datos deberíamos utilizar Factory Patterns, ó quizás un framework ORM.
4. También podemos aplicar Factory Patterns a la lógica de negocio.

Recuerden que este ejemplo es ilustrativo de como funciona los métodos `init()` y `destroy()`, su aplicación depende del sistema que se encuentre desarrollando.

16.7. Servlets y JavaBeans

16.7.1. ¿Qué es un JavaBeans?

Un JavaBean o Bean es un componente software que se puede reutilizar.

Aunque los Beans individuales pueden variar ampliamente en funcionalidad desde los más simples a los más complejos, todos ellos comparten las siguientes características:

- **Introspection:** Permite a la herramienta de programación o IDE analizar como trabaja el bean.
- **Customization:** El programador puede alterar la apariencia y la conducta del bean.
- **Events:** Informa al IDE de los sucesos que puede generar en respuesta a las acciones del usuario o del sistema, y también los sucesos que puede manejar.
- **Properties:** Permite cambiar los valores de las propiedades del bean para personalizarlo (customization).
- **Persistence:** Se puede guardar el estado de los beans que han sido personalizados por el programador, cambiando los valores de sus propiedades.

Podemos concluir que los Bean son componentes software que pueden ser manipulados visualmente por las herramientas de desarrollo (IDE).

Uno de los usos principales para los que se crearon los Bean es crear componentes visuales que pudieras añadirse en una paleta y usar en cualquier IDE de Java. La forma de conseguirlo es imponer una serie de reglas para que los IDE puedan "descubrir" automáticamente sus propiedades y puedan manipularlas de forma fácil. Estos Beans están pensados para ser ejecutados en el cliente, ya que forman parte de la interfaz.

Sin embargo, estas mismas convenciones que servían para los IDE, sirven para otras aplicaciones interesantes, como mapear clases Java a XML, clases Java a filas de tablas, manipular objetos a través de tags desde JSP, etc. y entonces los Beans pueden usarse por todo.

Adicionalmente, a un Bean se le suele llamar también "**Componente Java Bean**", suena mucho mejor que "**clase Java**" ó "**Bean**", pero debemos tener cuidado en no confundirlo con "**Componente EJB**".

16.7.2. Propiedades

Una propiedad es un atributo del Bean que afecta a su apariencia o a su conducta. Por ejemplo, un botón puede tener las siguientes propiedades: el tamaño, la posición, el título, el color de fondo, el color del texto, si está o no habilitado, etc.

Las propiedades de un bean pueden examinarse y modificarse mediante métodos o funciones miembros, que acceden a dicha propiedad, y pueden ser de dos tipos:

- **getter method:** lee el valor de la propiedad.
- **setter method:** cambia el valor de la propiedad.

Ejemplo 16 . 4

En este ejemplo desarrollaremos un programa que permita consultar los empleados de un departamento.

Para solucionar este problema aplicaremos el patrón de diseño Data Access Object.

Data Access Object (DAO) es un patrón del diseño de integración tal como está catalogado en el libro **Core J2EE Design Pattern**¹. DAO encapsula el código de acceso y manipulación al almacén persistente dentro de una capa separada. El almacén persistente en nuestro contexto es una base de datos de MySQL.

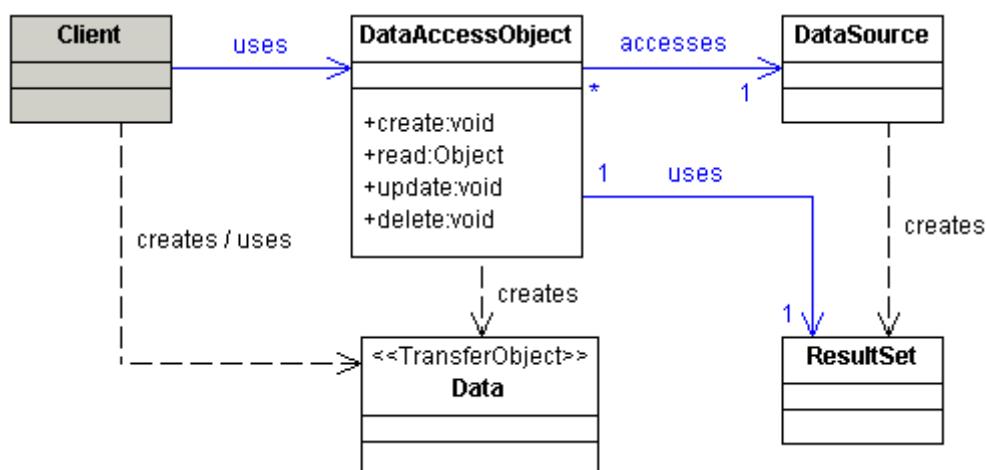


Figura 16 . 21 Patrón de Diseño DAO

Este patrón introduce una capa de abstracción entre la capa de lógica del negocio y la capa de persistencia almacenamiento, como se muestra en las Figuras 16.21 y 16.22. Los objetos de negocio acceden a la base de datos (Data Source) a través de los objetos de acceso a datos (DAO). Esta capa de abstracción simplifica el código de la aplicación e introduce flexibilidad. Idealmente, los cambios realizados a la fuente de datos, tales como cambio de proveedor de base de datos o del modelo, sólo sería necesario modificar los objetos de acceso a datos y deben tener un impacto mínimo en los objetos de negocio.

¹ Para acceder al libro **Core J2EE Design Pattern** en Internet puede consultar <http://www.corej2eepatterns.com/> y <http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html>

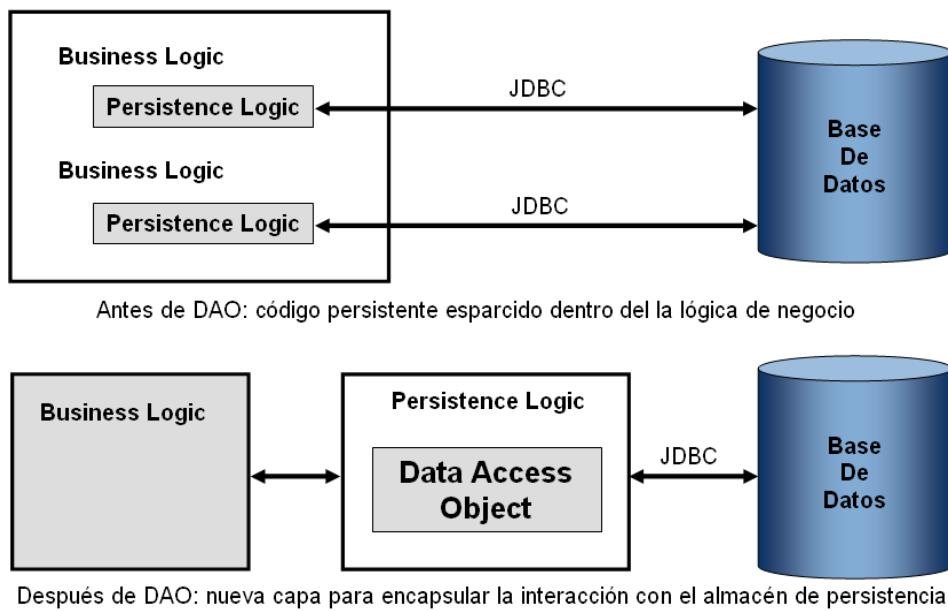


Figura 16 . 22 Estructura de una aplicación, antes y después de DAO.

Ahora que conocemos los fundamentos del patrón de diseño DAO, es el momento de diseñar nuestra solución.

La Figura 16.23 representa el diagrama de clases de la implementación del patrón de diseño DAO para nuestro problema, los elementos que intervienen son los siguientes:

- **AccesoDB**

Se trata de una clase que permite el acceso a una instancia (siempre la misma) del objeto Connection con el acceso a la base de datos EurekaBank.

- **EmpleadoTO**

Se trata de una clase que encapsula los datos de un empleado, se utiliza para comunicar los datos hacia el objeto DAO y viceversa.

- **IEmpleadoDAO**

Se trata de la interfaz donde se define las operaciones que debe implementar la clase **EmpleadoDAO**.

- **EmpleadoDAO**

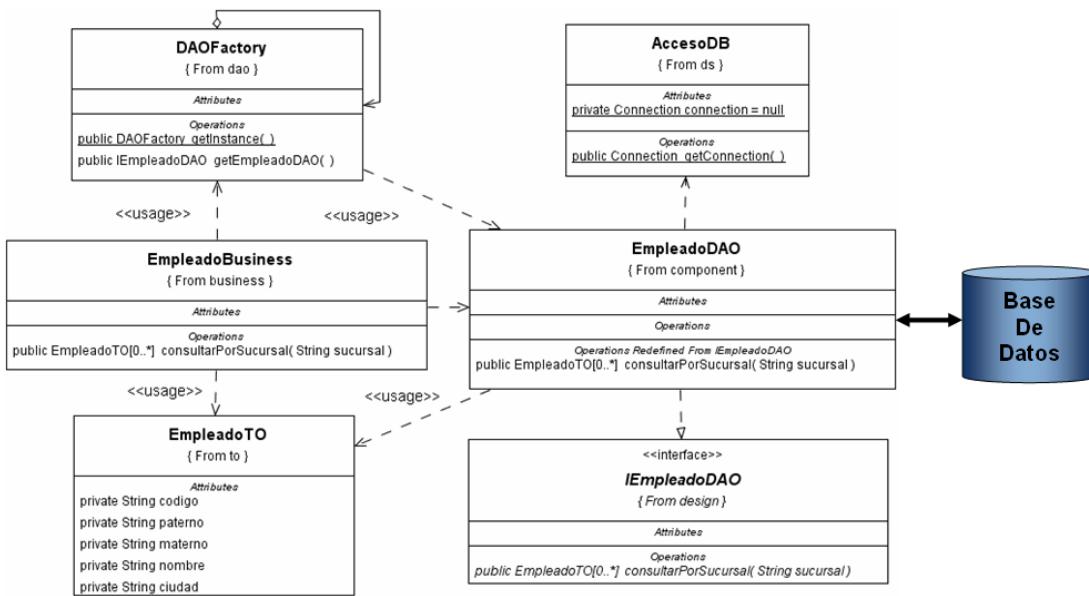
Encapsula todas las operaciones relacionadas a la tabla empleado. Se comunica con la fuente de datos, en este caso la base de datos EurekaBank.

- **DAOFactory**

Permite un acceso transparente a los objetos DAO.

- **EmpleadoBusiness**

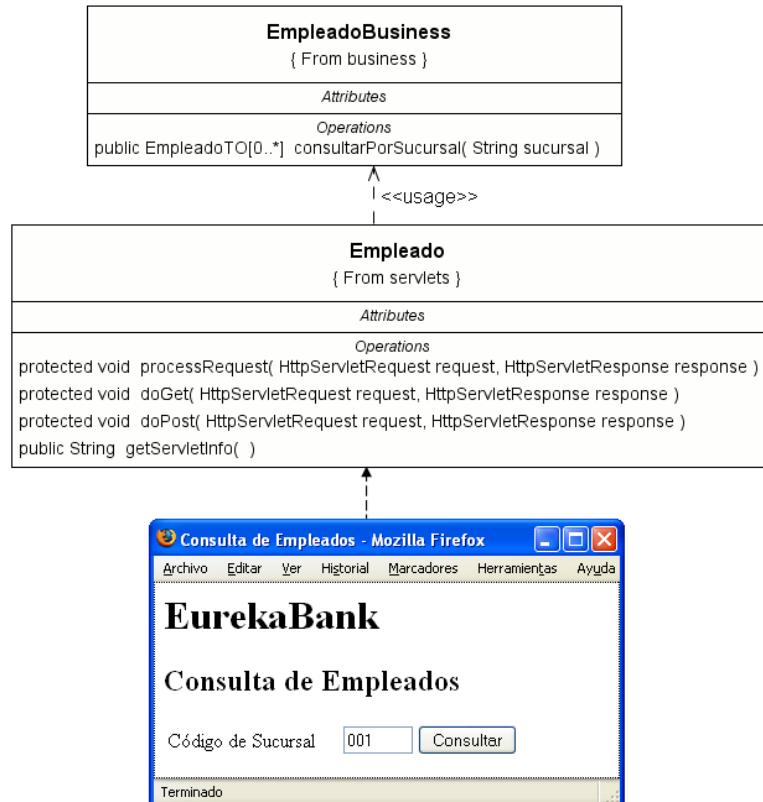
Implementa la lógica de negocio para la tabla empleado.

**Figura 16 . 23** Diseño del patrón DAO

La aplicación cliente interactúa con la capa DAO a través del objeto de negocio, tal como se ilustra en la Figura 16.24.

El servlet sólo interactúa con la capa de negocio y no tiene por qué enterarse que tipo de fuente de datos se está utilizando, de esta manera podemos cambiar de proveedor de datos y el impacto sería mínimo.

Si hay cambios en la regla de negocio el impacto sería también mínimo, esto debido a que las capas están bien definidas y totalmente desacopladas.

**Figura 16 . 24** Diseño de la aplicación.

Ahora es el momento de hacer la implementación.

Paso 01

Primero debemos crear el proyecto web **Cap16Ejemplo04**, y debemos agregarle la librería JDBC para MySQL.

Paso 02

Crear la clase **AccesoDB.java** para acceder a la base de datos, el script de esta clase es el siguiente:

```
package dao.ds;

import java.sql.Connection;
import java.sql.DriverManager;

/**
 *
 * @author Gustavo Coronel
 */
public class AccesoDB {

    private static Connection connection = null;

    public static Connection getConnection() throws Exception {
        String driver = "com.mysql.jdbc.Driver";
        String url = "jdbc:mysql://localhost/eurekabank";
        String user = "root";
        String pwd = "";
        if (connection == null) {
            Class.forName(driver).newInstance();
            connection = DriverManager.getConnection(url, user, pwd);
        }
        return connection;
    }
}
```

Esta clase implementa una variante del patrón singleton para acceder a solamente una instancia de la clase `Connection`.

Paso 03

Implementemos la clase **EmpleadoTO.java**, fundamental para encapsular los datos de un empleado. El script es el siguiente:

```
package dao.to;

/**
 *
 * @author Gustavo Coronel
 */
public class EmpleadoTO {

    private String codigo;
    private String paterno;
    private String materno;
    private String nombre;
    private String ciudad;
    private String direccion;
    private String usuario;
    private String clave;

    public String getCiudad() {
        return ciudad;
    }

    public void setCiudad(String ciudad) {
        this.ciudad = ciudad;
    }

    public String getClave() {
        return clave;
    }

    public void setClave(String clave) {
        this.clave = clave;
    }

    public String getCodigo() {
        return codigo;
    }

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

```
}

public String getDireccion() {
    return direccion;
}

public void setDireccion(String direccion) {
    this.direccion = direccion;
}

public String getMaterno() {
    return materno;
}

public void setMaterno(String materno) {
    this.materno = materno;
}

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getPaterno() {
    return paterno;
}

public void setPaterno(String paterno) {
    this.paterno = paterno;
}

public String getUsuario() {
    return usuario;
}

public void setUsuario(String usuario) {
```

```
        this.usuario = usuario;
    }
}
```

Paso 04

Definamos la interfaz **IEmppleadoDAO.java** para el objeto DAO de empleados, el script es el siguiente:

```
package dao.design;

import dao.to.EmpleadoTO;
import java.util.ArrayList;

/**
 *
 * @author Gustavo Coronel
 */
public interface IEmppleadoDAO {

    public abstract ArrayList<EmpleadoTO> consultarPorSucursal(String sucursal) throws Exception;
}
```

Paso 05

Implementemos la clase **EmpleadoDAO.java** para las operaciones con la tabla empleado, el script es el siguiente:

```
package dao.component;

import dao.design.IEmpleadoDAO;
import dao.ds.AccesoDB;
import dao.to.EmpleadoTO;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;

/**
 *
 * @author Gustavo Coronel
 */
public class EmpleadoDAO implements IEmpleadoDAO{
```

```

public ArrayList<EmpleadoTO> consultarPorSucursal(String sucursal) throws Exception {
    String query = "select * from empleado where chr_emplcodigo " +
        "in (select chr_emplcodigo from asignado where chr_sucucodigo = ?)";
    PreparedStatement ps = AccesoDB.getConnection().prepareStatement(query);
    ps.setString(1, sucursal);
    ResultSet rs = ps.executeQuery();
    ArrayList<EmpleadoTO> lista = new ArrayList<EmpleadoTO>();
    EmpleadoTO empl = null;
    while( rs.next() ){
        empl = new EmpleadoTO();
        empl.setCodigo(rs.getString("chr_emplcodigo"));
        empl.setPaterno(rs.getString("vch_emplpaterno"));
        empl.setMaterno(rs.getString("vch_emplmaterno"));
        empl.setNombre(rs.getString("vch_emplnombre"));
        empl.setCiudad(rs.getString("vch_emplciudad"));
        empl.setDireccion(rs.getString("vch_empldireccion"));
        empl.setUsuario(rs.getString("vch_emplusuario"));
        empl.setClave(rs.getString("vch_emplclave"));
        lista.add(empl);
    }
    rs.close();
    ps.close();
    return lista;
}

}

```

Paso 06

Implementemos la clase **DAOFactory.java** que permite el acceso transparente a los objetos DAO, el script es el siguiente:

```

package dao;

import dao.component.EmpleadoDAO;
import dao.design.IEmpleadoDAO;

/**
 *
 * @author Gustavo Coronel
 */

```

```
public class DAOFactory {  
  
    private static DAOFactory daoFac;  
  
    static {  
        daoFac = new DAOFactory();  
    }  
  
    public static DAOFactory getInstance() {  
        return daoFac;  
    }  
  
    public IEmpleadoDAO getEmpleadoDAO() {  
        return new EmpleadoDAO();  
    }  
}
```

Paso 07

Es hora de implementar la clase **EmpleadoBusiness.java** que implementa lo lógica de negocio de la tabla empleado. El script es el siguiente:

```
package business;  
  
import dao.DAOFactory;  
import dao.design.IEmpleadoDAO;  
import dao.to.EmpleadoTO;  
import java.util.ArrayList;  
  
/**  
 *  
 * @author Gustavo Coronel  
 */  
public class EmpleadoBusiness {  
  
    public ArrayList<EmpleadoTO> consultarPorSucursal(String sucursal) throws Exception {  
        IEmpleadoDAO empleadoDAO = DAOFactory.getInstance().getEmpleadoDAO();  
        return empleadoDAO.consultarPorSucursal(sucursal);  
    }  
}
```

Paso 08

A continuación tenemos la implementación del servlet **Empleado.java**, el script es el siguiente:

```
package servlets;

import business.EmpleadoBusiness;
import dao.to.EmpleadoTO;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author Gustavo Coronel
 */
public class Empleado extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        // Variables
        String sucursal = null;
        ArrayList<EmpleadoTO> lista = null;
        boolean hayError = false;
        String msgError = null;
        EmpleadoBusiness empleadoBusiness = new EmpleadoBusiness();
        // Proceso
        try {
            // Dato
            sucursal = request.getParameter("sucursal");
            lista = empleadoBusiness.consultarPorSucursal(sucursal);
        } catch (Exception e) {
            hayError = true;
            msgError = e.getMessage();
        }
    }

    // ... (resto del código)
}
```

```
        }

        // Reporte

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Consulta de Empleados</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>EurekaBank</h1>");
        out.println("<h2>Consulta de Empleados</h2>");

        if(hayError){

            out.println("<p>ERROR en el proceso</p>");

            out.println("<p>" + msgError + "</p>");

        }else if(lista.size() == 0){

            out.println("<p>No hay empleados.</p>");

        }else{

            out.println("<table width='400' border='1'> " +
                "<tr> " +
                "<th>Código</th> " +
                "<th>Paterno</th> " +
                "<th>Materno</th> " +
                "<th>Nombre</th> " +
                "</tr>");

            for (EmpleadoTO empleadoTO : lista) {

                out.println("<tr> " +
                    "<td>" + empleadoTO.getCodigo() + "</td> " +
                    "<td>" + empleadoTO.getPaterno() + "</td> " +
                    "<td>" + empleadoTO.getMaterno() + "</td> " +
                    "<td>" + empleadoTO.getNombre() + "</td> " +
                    "</tr>");

            }

            out.println("</table>");

        }

        out.println("<a href='empleado.html'>Otra Consulta</a>");

        out.println("</body>");

        out.println("</html>");

    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
```

```

        processRequest(request, response);
    }

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

public String getServletInfo() {
    return "Short description";
}// </editor-fold>
}

```

Paso 09

La página HTML que muestra el formulario para el ingreso del código de la sucursal se llama **empleado.html** y el script es el siguiente:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title>Consulta de Empleados</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>EurekaBank</h1>
        <h2>Consulta de Empleados</h2>
        <form name="form1" method="post" action="Empleado">
            <table width="317">
                <tr>
                    <td width="137">
                        Código de Sucursal
                    </td>
                    <td width="56">
                        <input name="sucursal" type="text" id="sucursal" size="5" maxlength="3">
                    </td>
                    <td width="108">
                        <input type="submit" name="consultar" id="consultar" value="Consultar">
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>

```

```
</form>
<p>&nbsp; </p>
</body>
</html>
```

Paso 10

La página **index.jsp** es la que se ejecuta por defecto, modifiquemosla para que realice un forward hacia la página **empleado.html**. El script es el siguiente:

```
<jsp:forward page="empleado.html"/>
```

Paso 11

Es momento de ejecutar el proyecto, la pantalla inicial nos muestra el formulario para ingresar el código de la sucursal a consultar, tal como lo puede ver en la Figura 16.25.

Luego de ingresar el dato solicitado y hacer clic en el botón **Consultar** obtenemos la lista de empleados correspondiente, tal como lo puede ver en la Figura 16.26.



Figura 16 . 25 Formulario de ingreso del código de la sucursal.



Figura 16 . 26 Resultado de la consulta.



Figura 16 . 27 Respuesta del servlet cuando no hay empleados.

Lo más recomendable sería programar el patrón DAO en un proyecto independiente, por ejemplo EurekaDAO y luego los proyectos que necesiten acceder a la fuente de datos importen el proyecto EurekaDAO, este es el caso que estaremos implementando posteriormente.

16.8. Interacción con un Servlet

16.8.1. Consideraciones Previas

Para hacer referencia a un servlet debemos tener en cuenta como se mapeado en el descriptor de despliegue (archivo web.xml), por ejemplo en el proyecto del Ejemplo 16.4 el servlet se ha mapeado como se muestra a continuación:

```
<servlet>
    <servlet-name>Empleado</servlet-name>
    <servlet-class>servlets.Empleado</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Empleado</servlet-name>
    <url-pattern>/Empleado</url-pattern>
</servlet-mapping>
```

La etiqueta **url-pattern** representa el **alias** con que debemos hacer referencia al servlet, normalmente se utiliza el mismo nombre de la clase pero no tiene que ser así.

Esta técnica nos permite cambiar el nombre del Servlet y del paquete libremente en un futuro sin que la aplicación en su conjunto se vea afectada por estos cambios, ya que se seguirá utilizando el mismo alias.

También debemos tener en cuenta como se ha definido el path de la aplicación, la Figura 16.28 nos muestra el path de la aplicación para el Ejemplo 16.4.



Figura 16 . 28 Definición del path de la aplicación.

16.8.2. Escribiendo la URL del servlet en un Navegador Web

Los servlets pueden ser llamados directamente escribiendo su URL en el campo dirección del navegador Web.

Para el Ejemplo 16.4 podríamos digitar en el campo dirección del navegador la siguiente URL:

```
http://localhost:8080/Cap16Ejemplo04/Empleado?sucursal=001
```

Y estaríamos consultando todos los empleados de la sucursal 001, tal como lo estaríamos haciendo desde el formulario de ingreso.

16.8.3. Llamar a un Servlet desde dentro de una página HTML

Las URLs de los servlets pueden utilizarse en etiquetas HTML, por ejemplo en un formulario o en un enlace, y pueden presentarse diferentes casos, que se explican a continuación:

1. Si el servlet está en otro servidor, debemos utilizar la URL completa.

```
<form method="post" action="http://localhost:8080/Cap16Ejemplo04/Empleado">
```

...

...

```
</form>
```

ó

```
<a href="http://localhost:8080/Cap16Ejemplo04/Empleado?sucursal=001">
```

Consultar

```
</a>
```

2. Si el servlet está en la misma aplicación sólo debemos hacer referencia al **alias** del servlet.

```
<form method="post" action="Empleado">
```

...

```

    ...
</form>

ó

<a href="Empleado?sucursal=001">
    Consultar
</a>

```

16.8.4. Llamada a un Servlet desde otro Servlet

Tenemos dos posibilidades, ejecutar un `sendRedirect()` o un `forward()`, que tienen el mismo objetivo, pero que funcionan diferente.

A continuación tenemos sus diferencias:

- `forward()` se ejecuta completamente en el servidor. Mientras que `sendRedirect()` conlleva a responder con un mensaje HTTP y esperar a que el navegador cliente acuda a la URL especificada. Es por ello que `forward()` es más rápido. Y es por ello que `sendRedirect()` modifica la URL del navegador.
- `forward()` permite llamar a un servlet o página JSP. Por el contrario en `sendRedirect()` se indica una URL que puede ser incluso una URL externa como "<http://www.google.com>" o cualquier otra.
- En un `forward()` se pasan dos argumentos: `request` y `response`. Esto permite pasar objetos en el scope `request`, por ejemplo. Mientras que en `sendRedirect()` los únicos parámetros que se pueden pasar son los de una URL "...?parametro1=valor1....". Obviamente también se podría usar otro scope, pero no el scope `request`.

Supongamos que tenemos dos servlets de nombre **Datos** y **Respuesta**. A continuación tenemos dos ejemplos, uno utilizando `sendRedirect()` y otro utilizando `forward()`.

- Desde el servlet **Datos** se realiza un `sendRedirect()` al servlet **Respuesta**:

```
response.sendRedirect("Respuesta");
```

- Desde el servlet **Datos** se realiza un `forward()` al servlet **Respuesta**:

```
RequestDispatcher rd = request.getRequestDispatcher("Respuesta");
rd.forward(request, response);
```

16.9. Sesiones

16.9.1. Introducción

Hay un número de problemas que vienen del hecho de que HTTP es un protocolo "**sin estado**". En particular, cuando estamos haciendo una compra on-line, es una molestia real que el servidor Web no puede recordar fácilmente transacciones anteriores. Esto hace que las aplicaciones como las canastas de compras sean muy problemáticas: cuando añadimos una entrada en nuestra canasta, ¿cómo sabe el servidor que es realmente nuestra canasta? Incluso si los servidores no retienen información contextual, todavía tendríamos problemas con comercio electrónico. Cuando nos movemos desde la página donde hemos especificado que queremos comprar (almacenada en un servidor Web normal) a la página que toma nuestro número de la tarjeta de crédito y la dirección de envío (almacenada en un servidor seguro que usa SSL), ¿cómo recuerda el servidor lo que hemos comprado?

Existen tres soluciones típicas a este problema:

1. **Cookies.** Podemos usar cookies HTTP para almacenar información sobre una sesión de compra, y cada conexión subsecuente puede buscar la sesión actual y luego extraer la información sobre esa sesión desde una localización en la máquina del servidor. Esta es una excelente alternativa, y es la aproximación más ampliamente utilizada. Sin embargo, aunque los servlets tienen un Interfaz de alto nivel para usar cookies, existen unos tediosos detalles que necesitan ser controlados:
 - Extraer el cookie que almacena el identificador de sesión desde los otros cookies (puede haber muchos).
 - Seleccionar un tiempo de expiración apropiado para el cookie (las sesiones interrumpidas durante 24 horas probablemente deberían ser reseteadas).
 - Asociar la información en el servidor con el identificador de sesión (podría haber demasiada información que se almacena en el cookie, pero los datos sensibles como los números de las tarjetas de crédito nunca deben ir en cookies).
2. **Reescribir la URL.** Podemos añadir alguna información extra al final de cada URL que identifique la sesión, y el servidor puede asociar ese identificador de sesión con los datos que ha almacenado sobre la sesión. Esta también es una excelente solución, e incluso tiene la ventaja que funciona con navegadores que no soportan cookies o cuando el usuario las ha desactivado. Sin embargo, tiene casi los mismos problemas que los cookies, a saber, que los programas del lado del servidor tienen mucho proceso que hacer, pero tedioso. Además tenemos que ser muy cuidadosos con que cada URL que le devolvamos al usuario tiene añadida la información extra. Y si el usuario deja la sesión y vuelve mediante un bookmark ó un enlace, la información de sesión puede perderse.
3. **Campos de formulario ocultos.** Los formularios HTML tienen una entrada que se parece a esto: <INPUT TYPE="HIDDEN" NAME="session" VALUE="...>. Esto significa que, cuando el formulario se envíe, el nombre y el valor especificado se incluirán en los datos GET o POST. Esto puede usarse para almacenar información sobre la sesión. Sin embargo, tiene la mayor desventaja en que sólo funciona si cada página se genera dinámicamente, ya que el punto negro es que cada sesión tiene un único identificador.

Los servlets proporcionan una solución técnica. La API HttpSession. Esta es una interfaz de alto nivel construida sobre las cookies y la reescritura de URL. De hecho, muchos servidores, usan cookies si el navegador las soporta, pero automáticamente se convierten a reescritura de URL cuando las cookies no son soportadas o están desactivadas. Pero el programador de servlets no necesita molestarte con muchos detalles, no tiene que manipular explícitamente las cookies o la información añadida a la URL, y tiene automáticamente un lugar conveniente para almacenar los datos asociados con cada sesión.

16.9.2. La API de Seguimiento de Sesión

Usar sesiones en servlets es bastante sencillo, la búsqueda del objeto de sesión asociado con el requerimiento actual, crear un nuevo objeto de sesión cuando sea necesario, buscar la información asociada con una sesión, almacenar la información de una sesión, y descartar las sesiones completas o abandonadas.

16.9.2.1. Buscar el Objeto HttpSession de la Sesión Actual

Esto se hace llamando al método `getSession()` de `HttpServletRequest`. Si devuelve `null`, podemos crear una nueva sesión, pero es tan comúnmente usado que hay una opción que crea automáticamente una nueva sesión si no existe una. Sólo pasamos `true` al método `getSession()`. Así, nuestro primer paso normalmente se parecerá a esto:

```
HttpSession session = request.getSession(true);
```

16.9.2.2. Buscar la Información Asociada con un Sesión.

Los objetos `HttpSession` viven en el servidor; son asociados automáticamente con el requerimiento mediante un mecanismo background como los cookies o la reescritura de URL. Estos objetos sesión tienen una estructura de datos interna que nos permite almacenar un número de claves y valores asociados.

En la versión 2.1 y anteriores del API servlet, usamos `getValue("key")` para buscar un valor previamente almacenado. El tipo de retorno es `Object`, por eso tenemos que forzarlo a un tipo más específico de datos. El valor de retorno es `null` si no existe dicho atributo. En la versión 2.2 `getValue()` está obsoleto en favor de `getAttribute()`, por el mejor nombrado correspondiente con `setAttribute()` (el correspondiente para `getValue()` es `putValue()`, no `setValue()`), y porque `setAttribute()` nos permite usar un `HttpSessionBindingListener` asociado para monitorizar los valores, mientras que `putValue()` no.

16.9.2.3. Asociar Información con una Sesión

Cómo se describió en la sección anterior, leemos la información asociada con una sesión usando `getValue()` (o `getAttribute()` desde la versión 2.2 de las especificaciones de los Servlets). Observa que `putValue()` ó `setAttribute()` reemplaza cualquier valor anterior. Algunas veces esto será lo que queremos pero otras veces queremos recuperar un valor anterior y aumentarlo.

A continuación tenemos un script que podría ayudarnos al momento de manipular la información de sesión:

```
HttpSession session = request.getSession(true);
int cant = (Integer) session.getAttribute("A001");
int n = Integer.parseInt(request.getParameter("num"));
cant += n;
session.setAttribute("A001", cant);
```

16.9.2.4. Finalizar una Sesión

Una sesión de usuario puede ser finalizada manual ó automáticamente.

Automáticamente una sesión es finalizada cuando no hay requerimiento a la aplicación por un periodo de tiempo de 30 minutos. Este valor es por defecto, se puede modificar en el descriptor de despliegue (archivo `web.xml`).

Para invalidar manualmente una sesión, se utiliza el método `invalidate()` del objeto de tipo `HttpSession`.

A continuación tenemos un script que ilustra como eliminar una sesión de usuario:

```
 HttpSession session = request.getSession(true);
 session.invalidate();
```

Invalidar una sesión significa eliminar el objeto de tipo `HttpSession` y con él todos sus valores del sistema.

Ejemplo 16 . 5

En este ejemplo simularemos una canasta de compra muy básica para ilustrar el tema de sesiones.

Tenemos que ingresar el nombre del artículo, su precio y la cantidad que se quiere comprar, y luego se debe mostrar la lista de artículos ingresados.

La Figura 16.29 muestra el diseño de las clases para registrar la canasta.

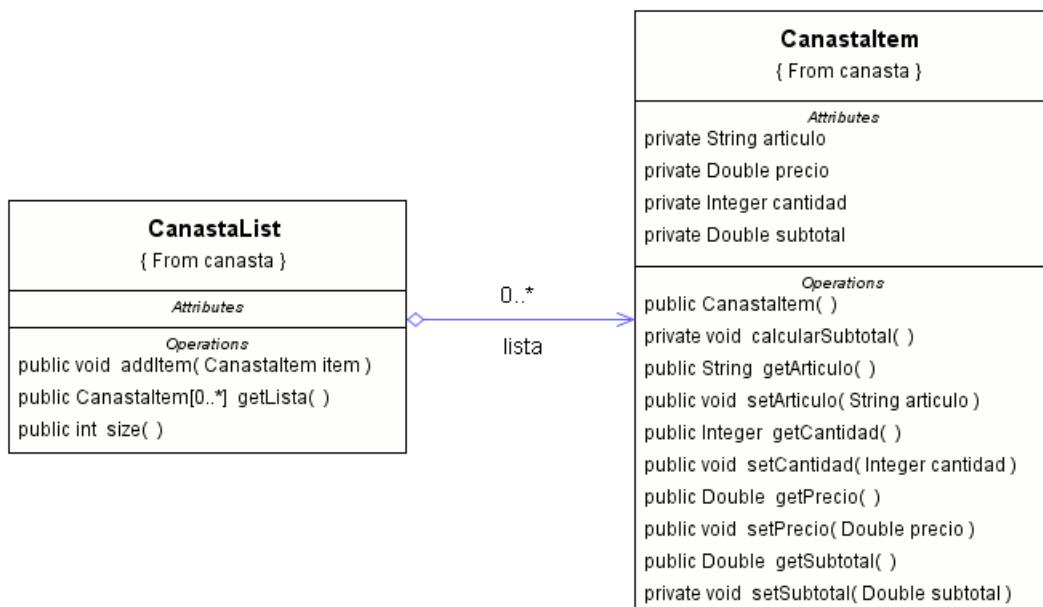


Figura 16 . 29 Diseño de clases para la canasta.

La clase **CanastaItem** representa un ítem de la canasta, por lo tanto la clase **CanastaList** representa a todos los ítems de la canasta. En sesión se debe guardar una instancia de la clase **CanastaList**.

Adicionalmente, debemos tener los siguientes recursos:

Recurso	Descripción
registrarVenta.html	Página principal que muestra el formulario de ingreso de datos y las opciones para operar con la canasta.
paginaError.html	Página que muestra un mensaje de error cuando el usuario ingresa datos incorrectos.
AgregarItem.java	Servlet que agrega un nuevo ítem a la canasta, luego transfiere el control al servlet ListarCanasta.java .

ListarCanasta.java	Servlet que muestra un listado del contenido de la canasta.
NuevaVenta.java	Servlet que elimina la sesión para iniciar una nueva venta.

La Figura 16.30 nos muestra la relación entre los diferentes recursos de la aplicación, por ejemplo el formulario definido en **registrarVenta.html** envía los datos al servlet AgregarItem.java, luego hacer su proceso hace un forward hacia el servlet ListarCanasta.java.

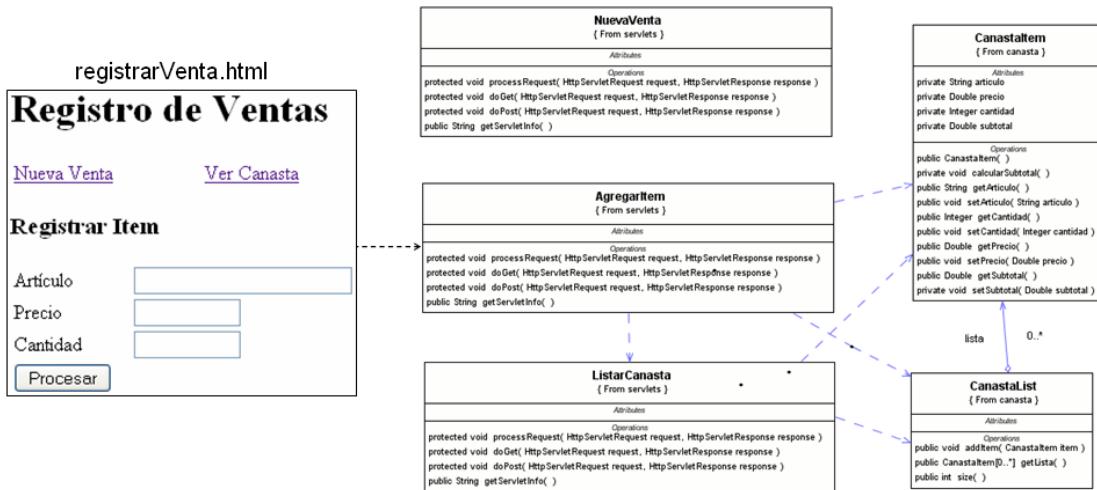


Figura 16 . 30 Relación entre los diferentes recursos de la aplicación.

Paso 01

Como primer paso debemos crear un nuevo proyecto web de nombre **Cap16Ejemplo05**.

Paso 02

Crear la clase **Canastaltem**, el script es el siguiente:

```

package canasta;

/**
 *
 * @author Gustavo Coronel
 */
public class Canastaltem {

    private String articulo;
    private Double precio;
    private Integer cantidad;
    private Double subtotal;

    public Canastaltem() {
        setArticulo(null);
        setCantidad(0);
        setPrecio(0.0);
    }
}
  
```

```
}

private void calcularSubtotal() {
    Double st;
    try {
        st = getPrecio() * getCantidad();
    } catch (Exception e) {
        st = 0.0;
    }
    setSubtotal(st);
}

public String getArticulo() {
    return articulo;
}

public void setArticulo(String articulo) {
    this.articulo = articulo;
}

public Integer getCantidad() {
    return cantidad;
}

public void setCantidad(Integer cantidad) {
    this.cantidad = cantidad;
    calcularSubtotal();
}

public Double getPrecio() {
    return precio;
}

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

public Double getSubtotal() {
```

```

        return subtotal;
    }

    private void setSubtotal(Double subtotal) {
        this.subtotal = subtotal;
    }
}

```

Note usted que cuando cambia el precio ó cantidad automáticamente se actualiza el subtotal, además, el método `setSubtotal()` es privado.

Paso 03

Crear la clase `CanastaList`, el script es el siguiente:

```

package canasta;

import java.util.ArrayList;

/**
 *
 * @author Gustavo Coronel
 */
public class CanastaList {

    private ArrayList<CanastaItem> lista = new ArrayList<CanastaItem>();

    public void addItem(CanastaItem item) {
        lista.add(item);
    }

    public ArrayList<CanastaItem> getList() {
        return lista;
    }

    public int size() {
        return lista.size();
    }
}

```

Esta clase es la que contiene la lista de items que el usuario va ingresando a la canasta, todos los items los guarda en un objeto de tipo `ArrayList`.

Paso 04

A continuación tenemos el listado del servlet **AgregarItem.java**:

```
package servlets;

import canasta.CanastaItem;
import canasta.CanastaList;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 *
 * @author Gustavo Coronel
 */
public class AgregarItem extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // Variables
        HttpSession session = null;
        String articulo = null;
        Double precio = null;
        Integer cantidad = null;
        CanastaList canasta = null;
        CanastaItem item = null;
        // Datos
        try {
            articulo = request.getParameter("articulo");
            precio = Double.parseDouble(request.getParameter("precio"));
            cantidad = Integer.parseInt(request.getParameter("cantidad"));
        } catch (Exception e) {
            response.sendRedirect("paginaError.html");
            return;
        }
    }
}
```

```

// Proceso
item = new CanastaItem();
item.setArticulo(articulo);
item.setPrecio(precio);
item.setCantidad(cantidad);
session = request.getSession(true);
if (session.getAttribute("canasta") == null) {
    canasta = new CanastaList();
} else {
    canasta = (CanastaList) session.getAttribute("canasta");
}
canasta.addItem(item);
session.setAttribute("canasta", canasta);
// Reporte
RequestDispatcher rd = request.getRequestDispatcher("ListarCanasta");
rd.forward(request, response);
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}

public String getServletInfo() {
    return "Short description";
} // </editor-fold>
}

```

Puede usted notar que si hay un error en los datos ingresados hay un redireccionamiento a la página de error **paginaError.html**.

Paso 05

El servlet **ListarCanasta.java** muestra el contenido de la canasta, a continuación tenemos su script:

```
package servlets;
```

```
import canasta.CanastaItem;
import canasta.CanastaList;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 *
 * @author Gustavo Coronel
 */
public class ListarCanasta extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<h1>Canasta</h1>");
        HttpSession session = request.getSession(true);
        if(session.getAttribute("canasta") == null){
            out.println("<h3>No existe canasta</h3>");
            out.println("<a href='registrarVenta.html'>Regresar</a>");
            out.close();
            return;
        }
        CanastaList canasta = (CanastaList)session.getAttribute("canasta");
        if(canasta.size() == 0){
            out.println("<h3>Canasta vacia</h3>");
            out.println("<a href='registrarVenta.html'>Regresar</a>");
            out.close();
            return;
        }
        out.println("<table border='1'>" +
                    "<tr>" +
                    "<th>Articulo</th>" +
```

```

        "<th>Precio</th>" +
        "<th>Cantidad</th>" +
        "<th>Subtotal</th>" +
        "</tr>");

Double total = 0.0;
for (CanastaItem item : canasta.getLista()) {
    total += item.getSubtotal();
    out.println("<tr>" +
        "<td>" + item.getArticulo() + "</td>" +
        "<td>" + item.getPrecio() + "</td>" +
        "<td>" + item.getCantidad() + "</td>" +
        "<td>" + item.getSubtotal() + "</td>" +
        "</tr>");
}
out.println("</table>");
out.println("<h2>Total: " + total + "</h2>");
out.println("<a href='registrarVenta.html'>Regresar</a>");
out.println("<a href='NuevaVenta'>Nueva Venta</a>");
out.close();
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

public String getServletInfo() {
    return "Short description";
} // </editor-fold>

}

```

Paso 06

Para iniciar una nueva venta limpiamos los datos de la sesión actual, para lo cual utilizamos el servlet **NuevaVenta.java**, a continuación tenemos su script:

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 *
 * @author Gustavo Coronel
 */
public class NuevaVenta extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        session.invalidate();
        response.sendRedirect("registrarVenta.html");
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    public String getServletInfo() {
        return "Short description";
    } // </editor-fold>

}
```

Paso 07

A continuación tenemos el script de la página de error **páginaError.html**:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>ERROR</h1>
    <p>Los datos ingresados no son correctos.</p>
    <p><a href="registrarVenta.html">Retornar</a></p>
  </body>
</html>
```

Esta página sólo muestra un mensaje de error predeterminado y tiene un enlace hacia la página **registrarVenta.html**.

Paso 08

A continuación tenemos el script de la página **registrarVenta.html**:

```
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>Registro de Ventas</h1>
    <table width="300">
      <tr>
        <td width="149"><a href="NuevaVenta">Nueva Venta</a></td>
        <td width="139"><a href="ListarCanasta">Ver Canasta</a></td>
      </tr>
    </table>
    <h3>Registrar Item</h3>
    <form name="form1" method="post" action="AregarItem">
      <table width="307">
        <tr>
          <td width="92">Artículo</td>
          <td width="203">
```

```
<input name="articulo" type="text" id="articulo" size="25" maxlength="25">
</td>
</tr>
<tr>
<td>Precio</td>
<td>
<input name="precio" type="text" id="precio" size="10" maxlength="10">
</td>
</tr>
<tr>
<td>Cantidad</td>
<td>
<input name="cantidad" type="text" id="cantidad" size="10" maxlength="10">
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="procesar" id="procesar" value="Procesar">
</td>
</tr>
</table>
</form>
</body>
</html>
```

Esta página muestra el formulario para el ingreso de los items de la canasta.

Paso 09

La página **index.jsp** que se crea por defecto cuando creamos el proyecto, debe hacer un forward hacia la página **registrarVenta.html**, el script es el siguiente:

```
<jsp:forward page="registrarVenta.html"/>
```

Paso 10

Finalmente, estamos en condiciones de ejecutar el proyecto, lo primero que tenemos es el formulario de ingreso de items, tal como se ilustra en la Figura 16.31.

Nueva Venta Ver Canasta

Registrar Item

Artículo	Camisa
Precio	68.0
Cantidad	3

Procesar

Terminado

Figura 16 . 31 Formulario de ingreso de items.

Después de ingresar datos y hacer clic en el botón procesar, se registra en item y se muestra la canasta, tal como se ilustra en la Figura 16.32.

Articulo	Precio	Cantidad	Subtotal
Camisa	68.0	3	204.0
Pantalon	120.0	2	240.0
Zapatos	240.0	1	240.0

Total: 684.0

Regresar Nueva Venta

Terminado

Figura 16 . 32 Visualización de la canasta por parte del servlet **ListarCanasta.java**.

Note que al mostrar el contenido de la canasta también se muestra el total de la venta, si queremos iniciar una nueva venta debemos hacer clic en el enlace **Nueva Venta** que los al mostrar la canasta ó en la página que muestra el formulario de ingreso de items, página **registrarVenta.html**.

Finalmente si al momento de ingresar los datos del item cometemos un error, por ejemplo ingresamos mal el precio o la cantidad, este error es detectado por el servlet **AgregarItem.java** y realiza un redireccionamiento a la página **paginaError.html** para mostrar el mensaje de error, tal como se ilustra 16.33.

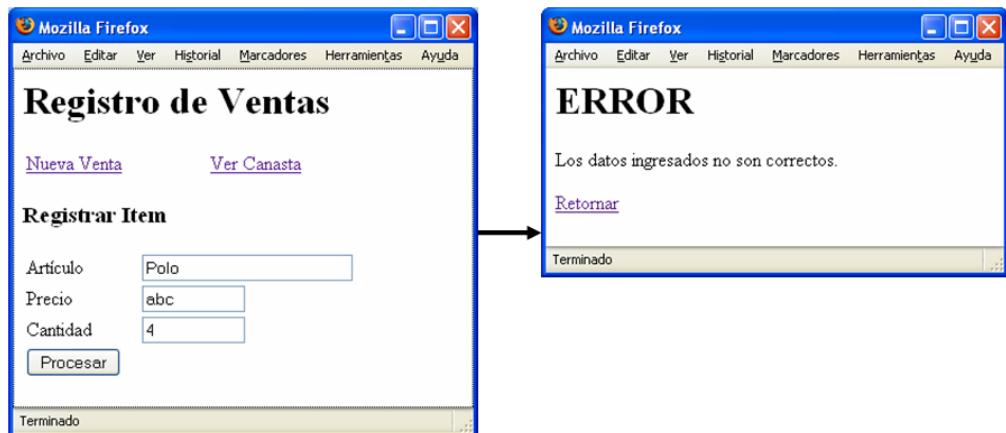
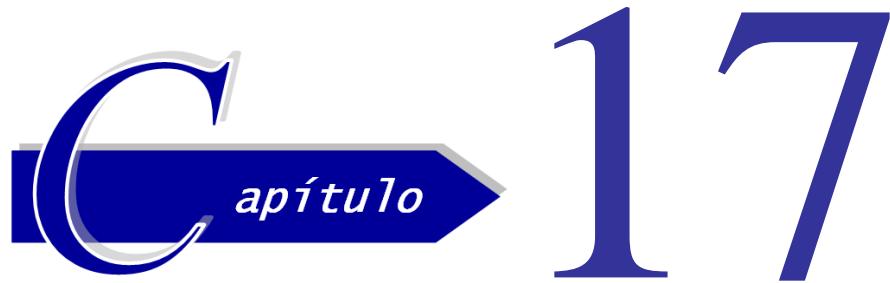


Figura 16 . 33 Mensaje de error cuando existen datos incorrectos.



17

Java Server Pages

En el desarrollo de aplicaciones web con Java, la base son los servlets, es por eso que debemos dedicarle el tiempo necesario para entender no solo su funcionamiento y como podemos aprovecharlos en la construcción de soluciones.

Los servlets son bastante utilizados en la implementación del controlador cuando aplicamos el patrón de diseño MVC en la construcción de aplicaciones web.

Temas a desarrollar:

- 17.1. Definición
- 17.2. Elementos Básicos
- 17.3. Directivas
- 17.4. Acciones
- 17.5. Objetos Implicitos

17.1. Definición

17.1.1. ¿Qué es JSP?

La tecnología Java Server Pages (JSP) aparece para aliviar lo difícil que supone generar páginas web mediante servlets. Además de esta forma es posible dividir el trabajo de construir una aplicación web entre programadores (servlets) y diseñadores gráficos (JSPs).

JSP es una tecnología que nos permite mezclar HTML estático con HTML generado dinámicamente. Muchas páginas Web que están construidas con programas CGI son casi estáticas, con la parte dinámica limitada a muy pocas localizaciones. Pero muchas variaciones CGI, incluyendo los servlets, hacen que generemos la página completa mediante nuestro programa, incluso aunque la mayoría de ella sea siempre lo mismo.

JSP es una especificación de Sun Microsystems. Sirve para crear y gestionar páginas Web dinámicas. Permite mezclar en una página código HTML para generar la parte estática, con contenido dinámico generado a partir de marcas especiales.

El contenido dinámico se obtiene, en esencia, gracias a la posibilidad de incrustar dentro de la página código Java de diferentes formas. Su objetivo final es separar la interfaz (presentación visual) de la implementación (lógica de ejecución).

Ejemplo 17 . 1

A continuación tenemos un ejemplo sencillo de una JSP que suma dos números, el nombre del archivo JSP es **demo01.jsp**.

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP - Demo 01</title>
  </head>
  <body>
    <%
      private int n1 = 15;
      private int n2 = 10;
      private int s;
    %>
    <h4>Suma de Dos N&uacute;meros</h4>
    <%
      s = n1 + n2;
      out.println("n1 = " + n1 + "<br>");
      out.println("n2 = " + n2 + "<br>");
      out.println("suma = " + s + "<br>");
    %>
  </body>
</html>
```

La Figura 17.1 nos muestra el resultado de la ejecución de este primer ejemplo.



Figura 17 . 1 Resultado de la ejecución de demo01.jsp.

17.1.2. Características

Las características de las páginas jsp son:

- La página JSP se convierte en un servlet.
- La conversión la realiza el servidor o contenedor JEE, la primera vez que se solicita la página JSP.
- Este servlet generado procesa cualquier requerimiento para esa página JSP.
- Si se modifica el código de la página JSP, entonces se regenera y recompila automáticamente el servlet y se recarga la próxima vez que sea solicitada.

NetBeans nos permite visualizar el servlet que genera el contenedor, tal como se ilustra en la Figura 17.2.

Por ejemplo, si estamos trabajando con el servidor **GlassFish V2**, el código del servlet para la página **demo01.jsp** es el siguiente:

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class demo01_jsp extends org.apache.jasper.runtime.HttpJspBase
implements org.apache.jasper.runtime.JspSourceDependent {

    private int n1 = 15;
    private int n2 = 10;
    private int s;

    private static final JspFactory _jspxFactory = JspFactory.getDefaultFactory();
    private static java.util.Vector _jspx_dependants;
    private org.apache.jasper.runtime.ResourceInjector _jspx_resourceInjector;

    public Object getDependants() {
        return _jspx_dependants;
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
throws java.io.IOException, ServletException {

        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;

        try {
            response.setContentType("text/html;charset=UTF-8");
            response.setHeader("X-Powered-By", "JSP/2.1");
            pageContext = _jspxFactory.getPageContext(this, request, response, null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            _jspx_out = out;
        }
    }
}
```

```
_jspx_resourceInjector = (org.apache.jasper.runtime.ResourceInjector)
    application.getAttribute("com.sun.appserv.jsp.resource.injector");

out.write("\n");
out.write("\n");
out.write("\n");
out.write("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"\n");
out.write("<http://www.w3.org/TR/html4/loose.dtd>\n");
out.write("\n");
out.write("<html>\n");
out.write("  <head>\n");
out.write("<meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\">\n");
out.write("<t<title>JSP - Demo 01</title>\n");
out.write("  </head>\n");
out.write("  <body>\n");
out.write("    \n");
out.write("    \n");
out.write("<t<h4>Suma de Dos N&uacute;meros</h4>\n");
out.write("    \n");
out.write("    \n");
s = n1 + n2;
out.println("n1 = " + n1 + "<br>");
out.println("n2 = " + n2 + "<br>");
out.println("suma = " + s + "<br>");

out.write("\n");
out.write("  </body>\n");
out.write("</html>");

} catch (Throwable t) {
    if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            out.clearBuffer();
        if (_jspx_page_context != null)
            _jspx_page_context.handlePageException(t);
    }
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
}
```

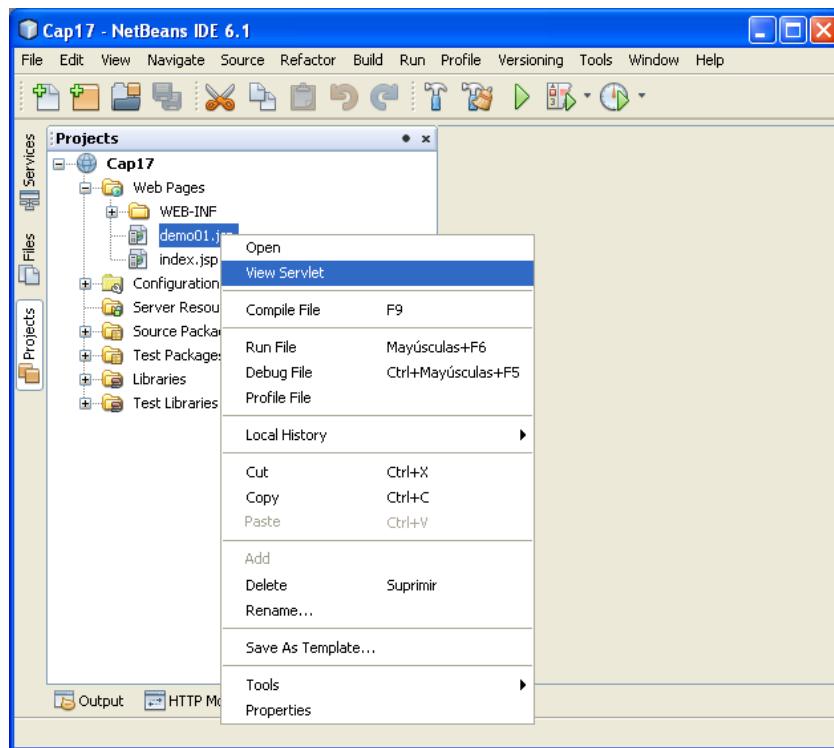


Figura 17 . 2 Opción para visualizar el servlet de una JSP.

17.1.3. Ciclo de Vida de un Documento JSP

17.1.3.1. A Nivel de Documento

1. Se crea el documento **jsp** en el editor.
2. Se publica.
3. El ejecutarse se crea un archivo **.java** y se compila generándose su **.class** para finalmente ejecutarse, esto se realiza únicamente la primera vez, y en los demás requerimientos solo se ejecuta su correspondiente **.class**.
4. Si existiese cambios en el documento **jsp** entonces se vuelve al punto 3.

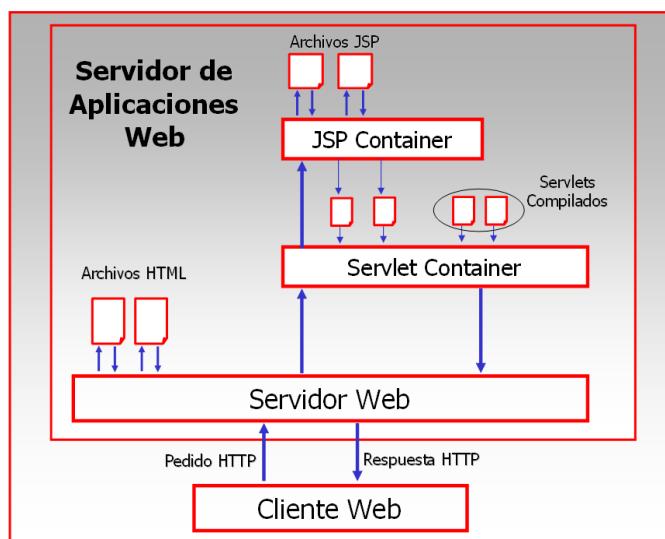


Figura 17 . 3 Arquitectura lógica de un cantenedor JEE.

La Figura 17.3 muestra de manera gráfica la arquitectura lógica del funcionamiento de una JSP.

17.1.3.2. A nivel de Ejecución de Eventos

El Figura 17.4 muestra el diagrama UML de las interfaces para crear documentos **jsp**.

Primero se ejecuta el método `jsplinit()`, que podría ser programado, este método se ejecuta una sola vez, la primera vez que el jsp es requerido, análogamente al método `init()` de un servlet.

La instancia creada para el servlet es también única, y es eliminada por el Garbage Collector del Servlet Engine, pero antes de eliminarse se ejecuta el método `jspDestroy()`, que podría ser también programado.

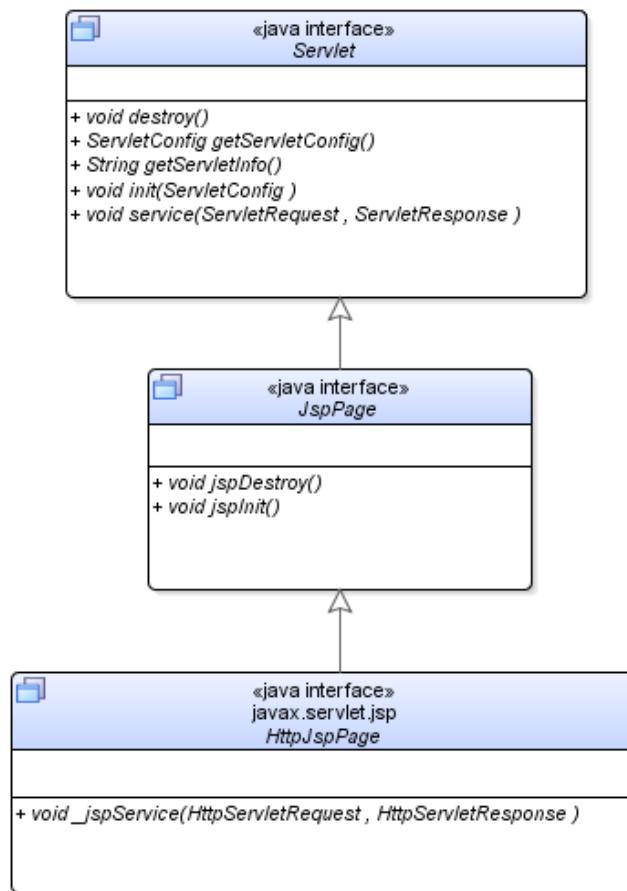


Figura 17 . 4 Interfaces que implementa un JSP.

Ejemplo 17 . 2

En el siguiente ejemplo se ilustra el funcionamiento de la sección de declaraciones de un JSP, que corresponde a la del servlet.

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Demo 02</title>
    </head>
    <body>
        <%!int cont = 0;%>
```

```

<h1>Demo 02</h1>
<%cont++;%>
<h2>n = <%=cont%></h2>
</body>
</html>

```

La primera vez que se ejecuta este jsp crea la variable **cont** y la inicializa en cero (0), y luego se incrementa en 1.

Las segundas veces que se ejecute el programa, el valor del contador se incrementa en 1, el resultado que se obtiene se muestra en la Figura 17.5.



Figura 17 . 5 Resultado de la ejecución del jsp.

17.2. Elementos Básicos

17.2.1. Declaraciones

Una **declaración** JSP nos permite definir métodos o campos que serán insertados dentro del cuerpo principal de la clase servlet (fuera del método service que procesa el requerimiento).

Sintaxis:

```

<%!
declaracion;
[ declaracion; ]
...
%>

```

A continuación tenemos un ejemplo ilustrativo:

```

<%!
private int cont = 0;
private int num1, num2, num3;
private Circulo objCirculo = new Circulo (2.0);
%>

```

Como las declaraciones no generan ninguna salida, normalmente se usan en conjunción con **expresiones** JSP o **scriptlets**. Por ejemplo, aquí tenemos un fragmento de JSP que imprime el número de veces que se ha solicitado la página actual desde que el servidor se inició (o la clase del servlet se modificó ó se recargó):

```

<%! private int accessCount = 0; %>
Acceso a esta página desde que se reinicio el servidor.

```

```
<%= ++accessCount %>
```

17.2.2. Expresiones JSP

Una **expresión JSP** se usa para insertar valores Java directamente en la salida.

Sintaxis:

```
<%= expresión_Java %>
```

La expresión Java es evaluada, convertida a un `String`, e insertada en la página. Esta evaluación se realiza durante la ejecución (cuando la página jsp es requerida) y así tiene total acceso a la información sobre la solicitud.

Por ejemplo, esto muestra la fecha y hora en que se solicitó la página:

```
Hora actual: <%= new java.util.Date() %>
```

Para simplificar estas expresiones, hay un gran número de variables predefinidas que podemos usar. Estos objetos implícitos se describen más adelante con más detalle, pero para el propósito de las expresiones, los más importantes son:

Objeto	Descripción
request	Objeto de tipo <code>HttpServletRequest</code>
response	Objeto de tipo <code>HttpServletResponse</code>
session	Objeto de tipo <code>HttpSession</code> asociado con el request.
out	Es de tipo <code>JspWriter</code> , se utiliza para enviar la salida al cliente.

Aquí tenemos un ejemplo:

```
Tú servidor es: <%= request.getRemoteHost() %>
```

Finalmente, también podemos utilizar etiquetas XML para las expresiones JSP, a continuación tenemos la sintaxis:

```
<jsp:expression>  
    Expresión Java  
</jsp:expression>
```

Demos recordar que los elementos XML, son sensibles a las mayúsculas; por eso debemos asegurarnos de usar minúsculas.

17.2.3. Scriptlets JSP

Si queremos hacer algo más complejo que insertar una simple expresión, los **scriptlets** JSP nos permiten insertar código dentro del método `service` que será construido al generar la página.

Sintaxis:

```
<%  
    // Código Java  
%>
```

Los Scriptlets tienen acceso a las mismas variables predefinidas que las expresiones. Por ejemplo, si queremos que la salida aparezca en la página resultante, tenemos que usar la variable **out**, tal como se ilustra a continuación:

```
<%
String queryData = request.getQueryString();
out.println("Datos enviados con GET: " + queryData);
%>
```

Observe que el código dentro de un scriptlet se insertará **exactamente** como está escrito, y cualquier HTML estático (plantilla de texto) anterior o posterior al scriptlet se convierte en sentencias **print**. Esto significa que los scriptlets no necesitan completar las sentencias Java, y los bloques abiertos pueden afectar al HTML estático fuera de los scriptlets. Por ejemplo, el siguiente fragmento JSP, contiene una mezcla de texto y scriptlets:

```
<% if (Math.random() < 0.5) {%
Este es un <B>agradable</B> dia!
%} else {%
Este es un <B>mal</B> dia!
%}>
```

Que se convertirá en algo como esto:

```
if (Math.random() < 0.5) {
    out.write("Este es un <B>agradable</B> dia!\n");
} else {
    out.write("Este es un <B>mal</B> dia!\n");
}
```

Si queremos usar los caracteres "%>" dentro de un scriptlet, debemos poner "%\>". Finalmente, si queremos usar el equivalente XML la sintaxis es la siguiente:

```
<jsp:scriptlet>
Código Java
</jsp:scriptlet>
```

Ejemplo 17 . 3

Desarrollar un proyecto que permita calcular el cociente y residuo de una división entera.

La interfaz de usuario es la que se muestra en la Figura 17.6.



Figura 17 . 6 Interfaz de usuario de la aplicación.

Los recursos para este proyecto son los siguientes:

Recurso	Descripción
matematica.html	Página web que muestra la interfaz de usuario.
matematica.jsp	Página jsp que recibe los datos del formulario, los procesa y muestra el resultado.

El script de la página **matematica.html** es el siguiente:

```
<html>
    <head>
        <title>Matematica</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>División</h1>
        <form name="form1" method="post" action="matematica.jsp">
            <table width="305">
                <tr>
                    <td width="84">Dividendo</td>
                    <td width="59">
                        <input name="dividendo" type="text" id="dividendo" size="5" maxlength="5">
                    </td>
                    <td width="146">&ampnbsp</td>
                </tr>
                <tr>
                    <td>Divisor</td>
                    <td><input name="divisor" type="text" id="divisor" size="5" maxlength="5"></td>
                    <td><input type="submit" name="calcular" id="calcular" value="Calcular"></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

El script de la página **matematica.jsp** es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%
    // Datos
    int dividendo = Integer.parseInt(request.getParameter("dividendo"));
    int divisor = Integer.parseInt(request.getParameter("divisor"));
    // Proceso
    int cociente = dividendo / divisor;
    int residuo = dividendo % divisor;
%>
<html>
```

```

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Matematica</title>
</head>
<body>
    <h1>División</h1>
    <table width="190">
        <tr>
            <td width="100">Dividendo</td>
            <td width="78"><%=dividendo%></td>
        </tr>
        <tr>
            <td>Divisor</td>
            <td><%=divisor%></td>
        </tr>
        <tr>
            <td>Cociente</td>
            <td><%=cociente%></td>
        </tr>
        <tr>
            <td>Residuo</td>
            <td><%=residuo%></td>
        </tr>
    </table>
    <p><a href="matematica.html">Retornar</a></p>
</body>
</html>

```

El resultado que se obtiene después de ingresar los datos y hacer clic en el botón calcular se puede apreciar en la Figura 17.7.



Figura 17 . 7 Resultado de la división.

Ejemplo 17 . 4

Hacer un proyecto que permita consultar el saldo de una cuenta de la base de datos EurekaBank.

Para este proyecto utilizaremos los siguientes recursos:

Recurso	Descripción
eurakaBank.css	Hoja de estilo que puede consultarla en el CD que viene junto con el libro.
consultarSaldo.jsp	Página jsp que muestra el formulario para ingresar el número de cuenta a consultar y también procesa el requerimiento. Esta página es recursiva.
paginaError.jsp	Página jsp que captura y muestra algún error que pueda producirse en la página consultarSaldo.jsp.

En la página de error **paginaError.jsp** debemos incluirle la siguiente directiva:

```
<%@ page isErrorPage="true" %>
```

El atributo `isErrorPage` en `true` especifica que la página jsp es una página de error y que podemos utilizar el objeto **exception**, que contiene una referencia a la excepción generada en el fichero JSP. En caso que `isErrorPage` tome valor `false` (valor por defecto), significa que no podemos usar el objeto **exception** y que la página jsp no es una página de error.

El script de la página **paginaError.jsp** es el siguiente:

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<%@ page isErrorPage="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link rel="stylesheet" type="text/css" href="eurekaBank.css">
<title>Consultar Saldo</title>
</head>
<body>
<h1>EurekaBank</h1>
<h2>Error</h2>
<p class="error"><%=exception.getMessage()%></p>
<p><a href="consultarSaldo.jsp">Retornar</a></p>
</body>
</html>
```

En la página **consultarSaldo.jsp** se manejan tres estados que se declaran en un tipo enumerado, la instrucción es la siguiente:

```
enum Estados {FORMULARIO, RESPUESTA, MENSAJE};
```

A continuación tenemos la descripción de cada estado:

Estado	Descripción
FORMULARIO	Muestra el formulario para ingresar el número de cuenta a consultar.
RESPUESTA	Muestra el saldo de la cuenta que se ha consultado.
MENSAJE	Muestra un mensaje en caso que la cuenta no exista.

El script de la página **consultarSaldo.jsp** es el siguiente:

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<%@ page import="java.sql.*" %>
<%@ page errorPage="paginaError.jsp" %>
<%!
    // Declaraciones de la página
    enum Estados {FORMULARIO, RESPUESTA, MENSAJE};

    Connection connection = null;

    String driver = "com.mysql.jdbc.Driver";
    String url = "jdbc:mysql://localhost/eurekabank";
    String user = "root";
    String pwd = "";
    String mensaje = "Número de cuenta no existe.";

%>
<%
    // declaración del requerimiento
    Estados estado = Estados.FORMULARIO;
    String cuenta = null;
    Double saldo = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    String query = "select * from cuenta where chr_cuencodigo=?";

    // Conexión con la base de datos
    if (connection == null) {
        Class.forName(driver).newInstance();
        connection = DriverManager.getConnection(url, user, pwd);
    }

    // Procesar rquerimiento
    if (request.getParameter("cuenta") != null) {
        // Datos
        cuenta = request.getParameter("cuenta");
        // Proceso
        ps = connection.prepareStatement(query);
        ps.setString(1, cuenta);
        rs = ps.executeQuery();
        if (rs.next()) {
            estado = Estados.RESPUESTA;
            saldo = rs.getDouble("dec_cuensaldo");
        } else {
    
```

```
        estado = Estados.MENSAJE;
    }
    rs.close();
    ps.close();
}
%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link rel="stylesheet" type="text/css" href="eurekaBank.css">
<title>Consultar Saldo</title>
</head>
<body>
<h1>EurekaBank</h1>
<h2>Consultar Saldo</h2>
<% if( estado == Estados.FORMULARIO ) {%
<form name="form1" method="post" action="consultarSaldo.jsp">
<table width="354">
<tr>
<td width="130">Número de Cuenta</td>
<td width="84">
<input name="cuenta" type="text" class="fieldEdit" id="cuenta"
size="8" maxlength="8">
</td>
<td width="124">
<input name="consultar" type="submit" class="button" id="consultar"
value="Consultar">
</td>
</tr>
</table>
</form>
<% } else if( estado == Estados.RESPUESTA ) { %
<table width="257">
<tr>
<td width="78">Cuenta</td>
<td width="167"><%=cuenta%></td>
</tr>
<tr>
<td>Saldo</td>
<td><%=saldo%></td>
</tr>
</table>
<p><a href="consultarSaldo.jsp">Retornar</a></p>
<% } else { %
<p>Mensaje: <%=mensaje%></p>
<p><a href="consultarSaldo.jsp">Retornar</a></p>
<% } %>
</body>
</html>
```

La directiva `<%@ page import="java.sql.*" %>` importa las librerías que se utilizarán para acceder a la base de datos.

La directiva `<%@ page errorPage="paginaError.jsp" %>` se utiliza para indicar cual es la página jsp que procesará los errores que pudieran ocurrir.

La Figura 17.8 muestra el formulario para el ingreso del número de cuenta.

The screenshot shows a Mozilla Firefox window with the title bar "Consultar Saldo - Mozilla Firefox". The menu bar includes "Archivo", "Editar", "Ver", "Historial", "Marcadores", "Herramientas", and "Ayuda". The main content area has a blue header "EurekaBank" and a sub-header "Consultar Saldo". Below this, there is a text input field labeled "Número de Cuenta" containing "00100001" and a blue "Consultar" button. At the bottom of the page, a status bar displays "Terminado".

Figura 17 . 8 Formulario para el ingreso del número de cuenta.

La Figura 17.9 muestra el resultado de la consulta.

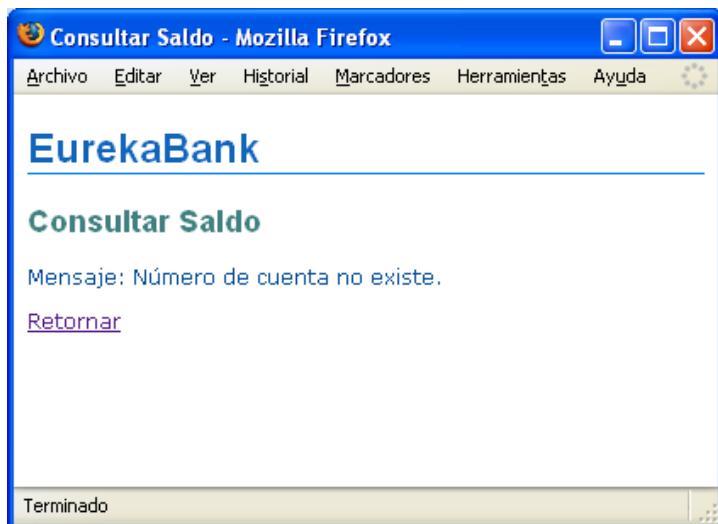
The screenshot shows a Mozilla Firefox window with the title bar "Consultar Saldo - Mozilla Firefox". The menu bar includes "Archivo", "Editar", "Ver", "Historial", "Marcadores", "Herramientas", and "Ayuda". The main content area has a blue header "EurekaBank" and a sub-header "Consultar Saldo". Below this, the results of the query are displayed in a table:

Cuenta	00100001
Saldo	7100.0

[Retornar](#) is shown below the table. At the bottom of the page, a status bar displays "Terminado".

Figura 17 . 9 Resultado de consulta.

Finalmente, la Figura 17.10 muestra el mensaje cuando la cuenta no exista.



17.3. Directivas

17.3.1. Introducción

Una directiva JSP afecta a la estructura general de la clase servlet. A continuación tenemos la sintaxis:

```
<%@ directiva atributo="valor" %>
```

Sin embargo, también podemos combinar múltiples atributos para una sola directiva, a continuación tenemos la sintaxis:

```
<%@ directiva atributo1="valor1"
   atributo2="valor2"
   ...
   atributoN="valorN" %>
```

Hay tres tipos principales de directivas:

Directiva	Descripción
include	Se usa para insertar un archivo dentro de la clase servlet, en el momento que la página JSP es traducida a un servlet
page	Se usa para definir atributos que se aplican toda la página jsp y cualquier archivo que se incluya con la directiva include .
taglib	Mediante esta directiva se puede ampliar el conjunto de etiquetas que el interprete de jsp es capaz de entender, de manera que la funcionalidad del mismo sea prácticamente ilimitada,

17.3.2. Directiva: include

Esta directiva nos permite incluir archivos en el momento en que la página JSP es traducida a un servlet.

Sintaxis:

```
<%@ include file="URL relativa" %>
```

Atributos

- file="URL relativa"

La URL que hay que introducir siempre es una URL relativa, es decir, sin protocolo, sin nombre de dominio (o dirección IP) y sin puerto. Como ejemplos tenemos las siguientes URLs:

- "error.jsp"
- "/m620/cabecera.html"

Si la URL relativa empieza con "/" la ruta es relativa al contexto de aplicación JSP.

El contenido del archivo incluido es analizado como texto normal JSP, pudiendo ser HTML estático, elementos de script, directivas y acciones.

Hay que tener cuidado de no insertar archivos que contengan las siguientes etiquetas <HTML>, </HTML>, <BODY>, o </BODY> debido a que todo el contenido del archivo es incluido dentro del archivo JSP. Estas etiquetas podrían entrar en conflicto con las etiquetas iguales dentro del archivo JSP que recibe la inserción del archivo, pudiendo producir un error.

Ejemplo 17 . 5

En este proyecto ilustraremos el uso de la directiva include, los recursos que utilizaremos son los siguientes:

Recurso	Descripción
header.html	Este archivo contiene el diseño de la cabecera que deben mostrar todas las páginas de la aplicación.
fecha.jsp	Esta página jsp muestra la fecha del servidor.
main.jsp	Es la página principal de la aplicación donde ilustraremos como usar la directiva include.

El script de la página **header.html** es el siguiente:

```
<table cellspacing="2" cellpadding="3" border="0" width="600">
<tr style="background-color:rgb(239,255,245); color:rgb(0,33,115);">
<td style="font-size:25.0pt; vertical-align:middle;" width="187">EurekaBank</td>
<td style="font-size:12.0pt; vertical-align:middle;" width="395">
<p>Un Banco Peruano a su Servicio<br />
Donde su Dinero CRECE con Seguridad.</p>
</td>
</tr>
</table>
```

Esta página podría tener lo que sería el encabezado de un sitio Web, en este caso es bastante sencillo pero podría ser la tarea de un especialista en diseño. Para este caso el resultado es el siguiente:

EurekaBank

Un Banco Peruano a su Servicio
Donde su Dinero CRECE con Seguridad.

El script de la página **fecha.jsp** es el siguiente:

```
<table cellspacing="3" cellpadding="2" border="0" width="600"
    style="background-color:rgb(255,251,240); color:rgb(0,0,128); font-size:15.0pt;
    border-color:rgb(0,49,148); border-style:solid; border-width:1.0pt;" bgcolor="Navy">
<tr>
<td>
    Hoy d&iacute;a es:<%= (new java.util.Date()).toLocaleString()%>
</td>
</tr>
</table>
```

Esta página muestra el fecha del sistema en un recuadro, tal como se ilustra a continuación:

Hoy día es:09/01/2009 08:38:51 PM

El script del programa **main.jsp** se muestra a continuación:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>JSP Page</title>
</head>
<body style="background-color:rgb(239,255,245);>
<%@ include file="/header.html"%>
<%@ include file="/fecha.jsp"%>
<p>Este es un ejemplo de como utilizar la directiva include.</p>
</body>
</html>
```

Este programa incluye los archivos **header.html** y **fecha.jsp**.

El resultado de ejecutar la página **main.jsp** se puede apreciar en la Figura 17.10.

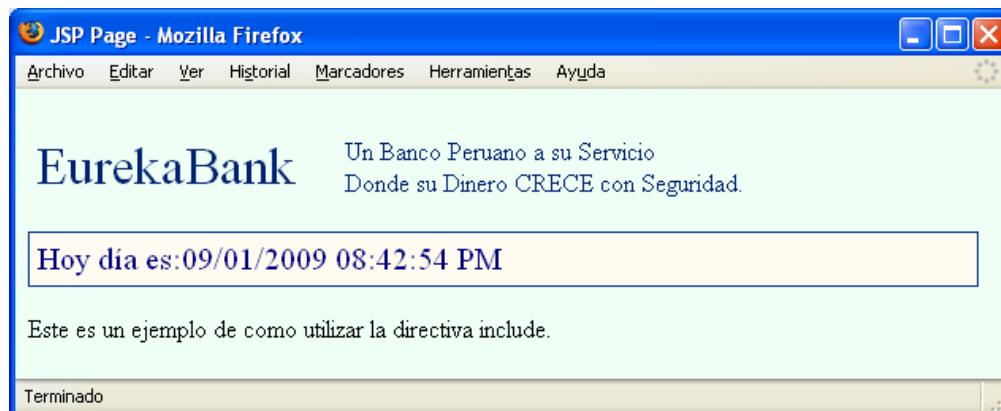


Figura 17 . 10 Ejecución del página **main.jsp**.

17.3.3.- Directiva: page

Define atributos que se aplican a una página JSP entera, y los archivos estáticos incluidos con la directiva **include**.

Sintaxis

En **negrita** tenemos los atributos por defecto.

```
<%@ page
[ language="java"]
[ extends="package.class"]
[ import="{package.class | package.*}, ..."]
[ session="true|false"]
[ buffer="none|8kb|sizekb"]
[ autoFlush="true|false"]
[ isThreadSafe="true|false"]
[ info="text"]
[ errorPage="relativeURL"]
[ contentType="{mimeType [,charset=characterSet]} | text/html ; charset=ISO-8859-1}"]
[ isErrorPage="true|false"]
%>
```

Descripción

Los atributos especificados por la directiva `<%@ page %>` (Incluidos los que tiene por defecto) son aplicados a toda la página JSP (una vez incluidos los ficheros estáticos que se incluyen mediante la directiva `include`).

Al conjunto formado por la página JSP y los ficheros estáticos que se incluyen se le denomina unidad de traducción (“**translation unit**”).

La directiva `<%@ page %>` no se aplica a los ficheros dinámicos que se incluyen mediante la acción `jsp:include`.

Se puede utilizar la directiva `page` más de una vez por unidad de traducción, pero sólo puede dársele valor una vez a cada atributo exceptuando el atributo `import` que puede utilizarse varias veces. Esto es debido a que el atributo `import` es similar a la sentencia `import` del lenguaje de programación Java. Se puede hacer uso de `import` más de una vez por página.

No importa donde se sitúe la directiva `<%@ page %>` en una página JSP o en los ficheros estáticos que se incluyen, pues se aplica a toda la unidad de traducción. Sin embargo habría que considerar una norma de buena práctica el situarlas todas al comienzo de la página JSP.

Atributos

- `language="java"`

El lenguaje con el cual añadimos funcionalidad o dinamismo a la parte estática. Aunque la especificación está abierta a otros lenguajes de script, de momento sólo hay versión para Java.

- `extends="package.class"`

Hace que la clase generada herede de esta clase. Una gran parte de la funcionalidad que proporciona el contenedor se pierde al utilizar este atributo ya que esta funcionalidad se consigue por herencia y en Java sólo puede heredarse de una clase.

Por ello este atributo es muy raramente utilizado, y, si se utiliza, hay que dotar al programa JSP con la funcionalidad que pierde.

- `import=" {package.class | package.*} , . . . "`

Una lista de paquetes o nombres cualificados de clases separados por el carácter “,” (coma), que van a ser importados en el programa JSP.

Estos paquetes y clases están disponibles para ser utilizados en las expresiones, scriptlets y métodos.

Los siguientes paquetes son importados implícitamente, por eso no es necesario importarlos:

- java.lang.*
- javax.servlet.*
- javax.servlet.jsp.*
- javax.servlet.http.*

A continuación tenemos algunos ejemplos de como podríamos utilizar el atributo **import**:

```
<%@ page import="java.util.*, java.lang.*" %>
<%@ page buffer="5kb" autoFlush="false" %>
<%@ page errorPage="error.jsp" %>
```

- **session="true|false"**

Indica si la página es añadida a la sesión del usuario o no. El valor por defecto es true.

- **buffer="none|8kb|sizekb"**

El tamaño del buffer usado por el objeto “out” (en kilobytes) para mandar el resultado de la ejecución del JSP al navegador (ej. en HTML).

El valor por defecto es 8kb.

- **autoFlush="true|false"**

El buffer de salida es automáticamente enviado al cliente cuando se llena. Cuando este atributo está a “**false**” y el buffer se llena se produce una excepción.

No puede establecer este atributo a “**false**” cuando el atributo “**buffer**” está puesto a “**none**”.

- **isThreadSafe="true|false"**

Este atributo especifica si la seguridad de threads está implementada en el archivo JSP. El valor por defecto, **true**, significa que el motor puede enviar múltiples requerimientos concurrentes a la página.

Si usamos el valor por defecto, varios threads pueden acceder a la página JSP. Por lo tanto, debemos sincronizar nuestros métodos para proporcionar seguridad de threads.

Con el valor **false**, el motor JSP no envía requerimientos concurrentes a la página JSP. Probablemente no sea recomendable forzar esta restricción en servidores de gran volumen de concurrencia por que puede perjudicar el rendimiento del servidor.

- **info="text"**

Cadena de texto que es incorporada literalmente a la página JSP. Posteriormente se puede obtener este valor mediante la llamada al método `Servlet.getServletInfo()`.

- **errorPage="relativeURL"**

Ruta relativa de la página jsp a la que se van a enviaran las excepciones (errores) que se produzcan en la página JSP. Si la ruta empieza con el carácter "/" (barra) entonces la ruta es relativa al directorio raíz de la aplicación. En caso contrario es relativa al directorio de la página JSP.

- **isErrorPage="true|false"**

Determina si una página JSP es una página de error o no. Si es una página de error se tendrá acceso al objeto implícito **exception**, no estando disponible en el caso que no sea una página de error.

- **contentType="mimeType [; charset=characterSet]" | "text/html ; charset=ISO-8859-1"**

El tipo MIME y el juego de caracteres que se utiliza dentro de una JSP determinan el tipo de salida que es enviada al cliente. Se puede utilizar cualquier tipo MIME o juego de caracteres que sea válido para el contenedor de JSP. Por defecto, el tipo MIME es **text/html** y el juego de **caracteres es ISO-8859-1**.

Ejemplo 17 . 6

En este ejemplo ilustraremos como funciona una página de error, para lo modificaremos el proyecto del Ejemplo 17.3, debe ejecutar cada una de los siguientes pasos.

1. Agregar la página jsp de nombre **paginaError.jsp**, el script de ésta página es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ page isErrorPage="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Error</h1>
    <p><%=exception.getMessage()%></p>
    <a href="matematica.html">Retornar</a>
  </body>
</html>
```

Note usted que mediante la directiva `<%@ page isErrorPage="true" %>` se le está indicando que se trata de una página de error.

2. En la página **matematica.jsp** debemos agregar la directiva `<%@ page isErrorPage="paginaError.jsp" %>` para indicarle cual será la página que tratará los errores.
3. Finalmente, cuando ejecutemos este proyecto e ingreamos como divisor el numero cero (0), obtendremos un mensaje de error por que no se puede dividir por cero, tal como se ilustra en la Figura 17.11.



Figura 17 . 11 Mensaje de error que muestra la página de error.

17.3.4. Directiva: taglib

Indica al contenedor JEE que la página jsp utilizará librerías de etiquetas. Estas librerías contienen etiquetas creadas por el propio programador con sus correspondientes atributos que encapsulan determinada funcionalidad. Lo habitual es utilizar librerías públicas que han diseñado otros programadores y han sido profusamente probadas. Se pueden obtener de <http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html>. Tiene dos atributos.

```
<%@ taglib uri="uriLibreriaEtiquetas" prefix="prefijoEtiqueta" %>
```

El contenido de cada atributo se describe a continuación:

Atributo	Descripción
uri	Permite localizar el archivo descriptor de la librería de etiquetas (TLD, Tag Library Descriptor).
prefix	Especifica el identificador que todas las etiquetas de la librería deben incorporar.

Las librerías de etiquetas es un tema muy interesante que facilita y complementa la programación habitual de las páginas jsp y que tocaremos en un capítulo posterior.

17.4. Acciones

Las acciones JSP usan sintaxis XML para controlar el comportamiento del motor de Servlets. Podemos insertar un fichero dinámicamente, reutilizar componentes JavaBeans, reenviar al usuario a otra página, o generar HTML.

Hay que tener cuidado con las mayúsculas y las minúsculas ya que el motor es sensible (case sensitive) con respecto a como se escriben las etiquetas.

17.4.1. Acción: <jsp:forward>

Esta acción permite redirigir la ejecución de la página JSP actual hacia otro recurso de forma permanente, si antes de utilizar esta etiqueta ya se ha enviado algún contenido del búfer del flujo de salida del cliente se producirá un error.

Cuando se localiza una etiqueta <jsp:forward> todo el contenido previo generado por la página JSP se descarta y se inicia la ejecución del recurso indicado en la acción <jsp:forward>.

Sintaxis:

```
<jsp:forward page="URLLocal"/>
```

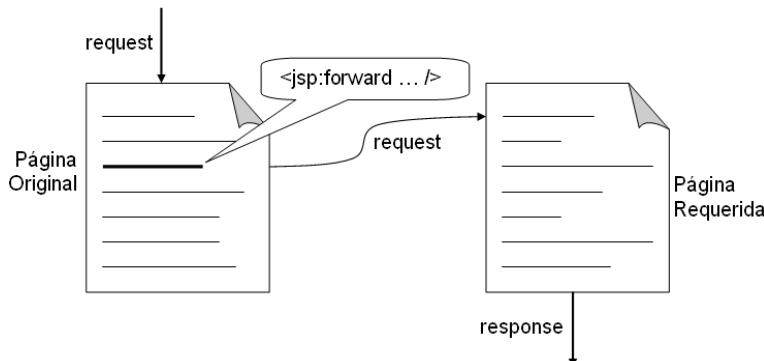


Figura 17 . 12 Esquema de funcionamiento de la acción <jsp:forward>.

El atributo `page` indica la localización del recurso al que se va a redirigir la ejecución de la página JSP actual. El valor de este atributo no tiene porque ser una constante, podemos utilizar expresiones y variables dentro de nuestra página JSP para indicar el valor correspondiente.

Cuando mediante la etiqueta `<jsp:forward>` se transfiere el control a la página JSP correspondiente, en el navegador no se muestra la URL de la nueva página, sino que se mantiene el anterior, esta nueva página tendrá un nuevo contexto, pero hay algunos atributos que se seguirán manteniendo en la nueva página dependiendo de su ámbito.

Así por ejemplo, se mantendrán los objetos (atributos) con ámbito de requerimiento (almacenados en el objeto `request`) y con ámbito de sesión (almacenados en el objeto `session`), los objetos con ámbito de aplicación (almacenados en el objeto `application`), se mantendrán en la nueva página si esta página pertenece a la misma aplicación Web que la página de origen. Los atributos que no se comparten son los que poseen ámbito de página (`page`).

A continuación tenemos un ejemplo ilustrativo:

```
<jsp:forward page="procesar.jsp"/>
```

En este ejemplo estamos redirigiendo el control a la página **procesar.jsp**.

Si antes de utilizar la acción `<jsp:forward>` enviamos algún contenido del búfer del flujo de salida, por ejemplo utilizando la sentencia `out.flush()`, se producirá un error cuando se pretenda redirigir la ejecución de la página actual hacia el recurso indicado.

Si deseamos indicar algún parámetro adicional a la página destino desde la página origen podemos utilizar otra acción estándar que tenemos disponible dentro de JSP, se trata de la acción `<jsp:param>`. Esta acción, que comentaremos en el siguiente apartado, permite especificar parámetros que se utilizarán en las acciones `<jsp:forward>`, `<jsp:include>` y `<jsp:plugin>` para indicar información adicional en forma de pares **nombre_de_parámetro/valor**.

17.4.2. Acción: `<jsp:param>`

Como ya hemos adelantado en el apartado anterior, esta acción se utiliza en colaboración con cualquiera de las siguientes acciones: `<jsp:forward>`, `<jsp:include>` o `<jsp:plugin>`.

Sintaxis:

```
<jsp:param name="nombreParametro" value="valorParametro"/>
```

La acción `<jsp:param>` permite ofrecer información adicional a otra acción.

Por ejemplo, asumamos que tenemos la página **demo03.jsp** que envía dos parámetros a la página **demo04.jsp**, tal como se ilustra en el siguiente script:

```
<jsp:forward page="demo04.jsp">
  <jsp:param name="nombre" value="Gustavo"/>
  <jsp:param name="apellido" value="Coronel"/>
</jsp:forward>
```

Si en **demo04.jsp** sólo queremos imprimir el valor de estos dos parámetros, el script podría ser el siguiente:

```
<p>Nombre: <%=request.getParameter("nombre")%></p>
<p>Apellido: <%=request.getParameter("apellido")%></p>
```

17.4.3. Acción: `<jsp:include>`

La acción `<jsp:include>` permite insertar en el contenido generado por la página actual, el contenido de otro recurso distinto, resultando la salida final que se envía al usuario una combinación de ambos contenidos. Al contrario de lo que ocurría en la acción `<jsp:forward>`, el control de la ejecución vuelve a la página original una vez que se ha terminado la ejecución de la página incluida.

Sintaxis:

```
<jsp:include page="URLLocal" flush="true|false"/>
```

Como se puede comprobar de la sintaxis anterior esta acción utiliza dos atributos. El atributo `page` donde especificamos la URL que identifica la página o recurso que queremos incluir en el contenido que se envía al cliente, y el atributo `flush` que indica si el búfer del flujo de salida de la página actual es enviado al cliente antes de realizar la inclusión de la página indicada.

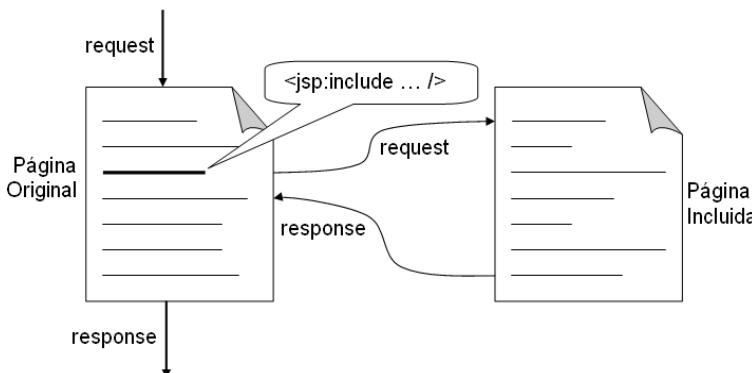


Figura 17 . 13 Esquema de funcionamiento de la acción `<jsp:include>`.

Si el atributo `flush` tiene el valor de **false**, primero se incluirá el contenido de la página indicada, y luego el de la página original.

Si el atributo `flush` tiene el valor **true** (forzar el vaciado de buffer) es problemático porque una vez que ha sucedido esto no se pueden hacer redirecciones ni ir a páginas de error, porque ya se han terminado de escribir las cabeceras.

Por ejemplo, supongamos que tenemos una página jsp de nombre **demo05.jsp** que debe incluir el resultado de la página **demo06.jsp**.

El script de la página **dem05.jsp** es el siguiente:

```
<h3>Se incluye el resultado de demo06.jsp</h3>
<jsp:include page="demo06.jsp" flush="true"/>
<h3>Se vuelve a la página inicial</h3>
```

El script de la página **demo06.jsp** es el siguiente:

```
<%for (int i = 1; i <= 7; i++) {%
<font size="<%i%>">Hola Perú</font><br>
<%}>
```

El resultado de la ejecución de la página demo05.jsp se puede apreciar en la Figura 17.14.

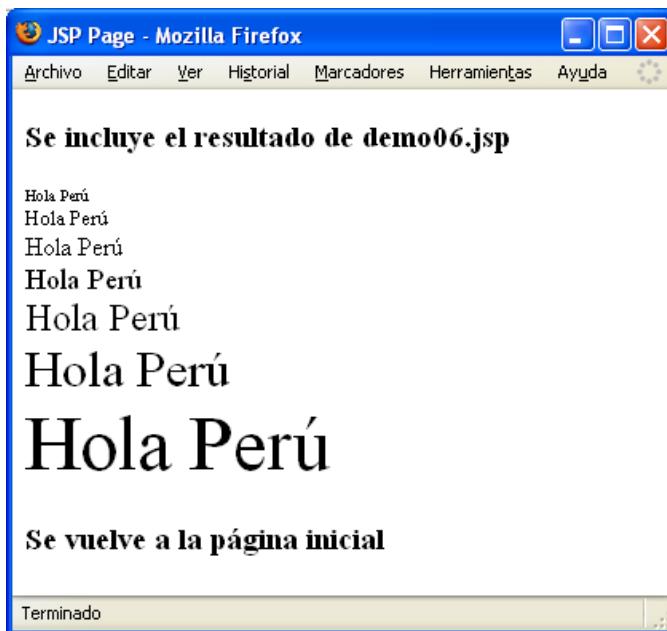


Figura 17 . 14 Resultado de la ejecución de demo05.jsp.

Con la acción `<jsp:include>` podemos utilizar la acción `<jsp:param>` para pasar información adicional a la página que vamos a incluir en forma de parámetros. La visibilidad en cuanto al ámbito de los atributos de la página original respecto a la incluida, es la misma que con la acción `<jsp:forward>`, es decir, son visibles los atributos con ámbito de requerimiento y con ámbito de sesión, y los que tienen ámbito de aplicación su visibilidad dependerá de si la página incluida pertenece a la misma aplicación Web que la página de origen o no.

Cuando se realiza una modificación en los recursos que se incluyen mediante la acción `<jsp:include>`, esta modificación se refleja de forma automática en el resultado de la ejecución de la página JSP que utiliza la acción. Esto es debido a que la incorporación del contenido de la página indicada en la acción `<jsp:include>` se realiza cada vez que se ejecuta la página original, es decir, el contenido de incluye en tiempo de ejecución.

Sin embargo la directiva `include` (que ya vimos en un apartado anterior), no reflejará los cambios en la página incluida de forma automática, ya que el contenido de la página que se desea incluir se incorpora no en tiempo de ejecución, sino cuando se traduce la página JSP al servlet correspondiente, es decir, en tiempo de traducción.

Las ventajas que ofrece la acción `<jsp:include>` sobre la directiva `include` son la recompilación automática, las clases de los servlets son más pequeñas, ya que la página incluida se incluye en tiempo de ejecución, además se permite la utilización de parámetros adicionales mediante la acción `<jsp:param>`. Por otro lado la directiva `include` ofrece la posibilidad de utilizar las variables con ámbito de la página y ofrece una mayor eficiencia en la ejecución, ya que a la

hora de ejecutarse en servlet que se corresponde con la página JSP ya tiene incluido el contenido de la página que se desea incluir.

17.4.4. Acción: <jsp:plugin>

Esta acción permite generar código **HTML** para asegurar que el navegador utiliza el plug-in de Java, se generarán por lo tanto las etiquetas **<OBJECT>** y **<EMBED>** correspondientes para indicar al navegador que para un applet determinado debe cargar el plug-in de Java para que el applet pueda ser ejecutado satisfactoriamente por el navegador.

17.4.5. Acción: <jsp:useBean>

Esta acción se utiliza para poder utilizar dentro de una página JSP un componente JavaBean en un ámbito determinado. El componente JavaBean podrá ser utilizado dentro de la página JSP haciendo referencia al nombre indicado dentro de la acción **<jsp:useBean>**, teniendo siempre en cuenta el ámbito al que pertenece el Bean, y que se indica también en la acción **<jsp:useBean>**.

La acción **<jsp:useBean>** indica a la página JSP que queremos tener un Bean determinado disponible, el contenedor de páginas JSP creará el Bean correspondiente o bien lo recuperará del ámbito correspondiente.

Sintaxis:

```
<jsp:useBean  
    id="beanInstanceName"  
    scope="page | request | session | application"  
    {  
        class="package.class" |  
        type="package.class" |  
        class="package.class" type="package.class" |  
        beanName="{package.class | <%= expression %>}" type="package.class"  
    }  
    {  
        /> |  
        > other elements </jsp:useBean>  
    }  
}
```

A continuación describimos los distintos atributos de la acción **<jsp:useBean>**.

Atributo	Descripción
id	Es el identificador que vamos a utilizar dentro de la página JSP, y durante el resto del ciclo de vida del Bean para hacer referencia al Bean que se crea o se utiliza. Se puede elegir cualquier nombre para hacer referencia a un Bean, aunque se deben seguir una serie de normas: este identificador debe ser único en la página, se distingue entre mayúsculas y minúsculas, el primer carácter debes ser una letra, sólo se permiten letras, números y carácter de subrayado (<u>_</u>), no se permiten por lo tanto espacios.
class	En este atributo se indica el nombre de la clase del Bean, a cada Bean le va a corresponder una clase, para organizar los componentes JavaBeans estas clases suelen encontrarse en paquetes, y si no los hemos importado en la directiva page , deberemos utilizar el nombre completo de la clase del Bean, indicando paquetes y subpaquetes.
type	Este atributo no suele ser utilizado, es opcional e indica el tipo de la clase del Bean, este tipo suele ser la clase padre, un interfaz o la

	propia clase, se utilizará para casos muy concretos cuando queremos realizar una casting de clases. Por defecto tiene el mismo valor que el indicado atributo class.
scope	Indica el ámbito que le va a corresponder al Bean, ya sabemos que existen cuatro ámbitos distintos, y por lo tanto este atributo podrá tener los valores page , request , session ó application , por defecto se utiliza el ámbito de página (page).
beanName	Es el nombre del Bean que se le pasa al método <code>instantiate(..)</code> de la clase <code>java.beans.Beans</code> , es posible indicar el atributo type y el atributo beanName y omitir el atributo class .

La acción `<jsp:useBean>` primero intenta localizar una instancia del Bean. Si el Bean no existe, `<jsp:useBean>` crea una instancia desde una clase o plantilla serializada.

Para localizar o instanciar el Bean, `<jsp:useBean>` sigue los siguientes pasos, en este orden:

1. El contenedor trata de localizar un objeto que posea el identificador y ámbito indicados en la acción `<jsp:useBean>`.
2. Si se encuentra el objeto, y se ha especificado un atributo **type**, el contenedor trata de utilizar la conversión de tipos del objeto encontrado con el tipo (**type**) especificado, si la conversión falla se lanzará una excepción `ClassCastException`. Si únicamente se ha indicado un atributo **class** se creará una nueva referencia al objeto en el ámbito indicado utilizando el identificador correspondiente.
3. Si el objeto no se encuentra en el ámbito especificado y no se ha indicado un atributo **class** o **beanName**, se lanzará una excepción `InstantiationException`.
4. Si el objeto no se ha encontrado en el ámbito indicado, y se ha especificado una clase concreta con un constructor público sin argumentos, se instanciará un objeto de esa clase. Un nuevo objeto se asociará con el identificador y ámbitos indicados. Si no se da alguna de las condiciones comentadas, se lanzará una excepción `InstantiationException`.
5. Si el objeto no se localiza en el ámbito indicado, y se ha especificado un atributo **beanName**, se invocará al método `instantiate(...)` de la clase `java.beans.Beans`, pasándole como parámetro el valor de la propiedad **beanName**. Si el método tiene éxito un nuevo objeto se asociará con el identificador y ámbitos indicados.
6. Si la acción `<jsp:useBean>` no posee un cuerpo vacío, se procesará el cuerpo de la etiqueta, únicamente si el objeto se crea como un objeto nuevo, es decir, no se había encontrado en el ámbito indicado.

El cuerpo de un `<jsp:useBean>` a menudo contiene elementos `<jsp:setProperty>` que establece valores de propiedades en el Bean. Como se describió en el paso 5, las etiquetas del cuerpo solo son procesadas si `<jsp:useBean>` instancia el Bean. Si el Bean ya existe y `<jsp:useBean>` lo localiza, las etiquetas del cuerpo no tienen efecto.

El siguiente script muestra un sencillo ejemplo de como crear un Bean con ámbito de página. Como se puede observar se ha utilizado la clase `java.util.Date` como clase del Bean, lo normal es utilizar un componente JavaBeans pero para este ejemplo la clase `Date` nos sirve perfectamente para mostrar como se puede acceder en la página al Bean que se ha creado utilizando su identificador.

```
<jsp:useBean id="fecha" class="java.util.Date" scope="page"/>
<%=fecha%>
```

En un ejemplo más convencional de como utilizar la acción `<jsp:useBean>`, podríamos crear el componente JavaBeans cuya clase es **Empleado**, y crearlo con ámbito de sesión, por lo tanto

este Bean será accesible por el resto de páginas JSP que compartan esta misma sesión. A continuación tenemos el script:

```
<jsp:useBean id="empleado" scope="session" class="dao.to.Empleado"/>
<%
    empleado.setCodigo("12345");
    empleado.setNombre("Gustavo Coronel");
    empleado.setUsuario("gcoronel");
%>
```

Para acceder desde otra página al Bean que hemos creado con ámbito de sesión únicamente se debe utilizar la misma acción `<jsp:useBean>` que hemos utilizado para crear el Bean. El siguiente script muestra como acceder a este Bean, se supone que ya lo hemos creado en el script anterior, si el Bean no estuviera creado se procedería a instanciarlo.

```
<jsp:useBean id="empleado" scope="session" class="dao.to.Empleado">
<p>Nombre: <%=empleado.getNombre()%></p>
```

Como se puede ver en los scripts anteriores, para acceder a una propiedad del Bean utilizamos los métodos de acceso correspondientes que ofrecen la clase del Bean, esta no es la forma usual de acceder a las propiedades de un Bean para obtener o establecer los valores de sus propiedades, sino que desde una página JSP utilizaremos un par de acciones para realizar estas tareas. Se trata de las acciones `<jsp:getProperty>` y `<jsp:setProperty>`, que permiten obtener y establecer el valor de una propiedad de un Bean, respectivamente. Estas dos acciones las veremos en los siguientes apartados.

17.4.6. Acción: `<jsp:getProperty>`

Esta acción forma parte de las acciones que nos permiten utilizar componentes Beans dentro de nuestras páginas JSP, en este caso la acción `<jsp:getProperty>` nos va a permitir obtener el valor de la propiedad de un Bean creado en la página con el ámbito correspondiente.

La sintaxis de esta acción es muy sencilla, no posee cuerpo y únicamente presenta dos atributos, como se puede observar en la sintaxis.

Sintaxis:

```
<jsp:getProperty name="nombreBean" property="nombrePropiedad"/>
```

El atributo **name** indica el identificador del Bean que hemos creado con la acción `<jsp:useBean>`, y cuyo valor de la propiedad queremos obtener. Se corresponderá con el valor del atributo **id** de la acción `<jsp:useBean>` correspondiente.

El atributo **property** indica el nombre de la propiedad del Bean cuyo valor se desea obtener. El valor de la propiedad se mostrará como código HTML, reemplazando en tiempo de ejecución a la acción `<jsp:getProperty>` correspondiente.

Esta acción accede al valor de la propiedad especificada del Bean correspondiente, la convierte en un objeto de la clase `String` y la imprime en el flujo de salida de la página JSP.

Utilizando esta nueva acción podríamos rescribir alguno de los ejemplos anteriores para reemplazar el uso de los métodos de la clase del Bean por la utilización de la acción `<jsp:getProperty>` para obtener el valor de la propiedad del Bean.

Por ejemplo, para imprimir el nombre del empleado en el script mostrado en el punto 17.4.5 podríamos modificarlo de la siguiente manera.

```
<jsp:useBean id="empleado" scope="session" class="dao.to.Empleado">
<p>Nombre: <jsp:getProperty name="empleado" property="nombre"/>%></p>
```

Si la propiedad que indicamos en esta acción no se corresponde con ninguna propiedad del Bean indicado se producirá un error.

17.4.7. Acción `<jsp:setProperty>`

Esta acción permite modificar las propiedades de los Beans a los que hacemos referencia en nuestras páginas JSP, es la acción complementaria a la acción `<jsp:getProperty>`.

Sintaxis:

```
<jsp:setProperty
    name=" nombreBean"
    {
        property= "*"  |
        property="nombrePropiedad" [ param="nombreParametro" ]  |
        property="nombrePropiedad" value="{cadena | <%= expresión %>}"
    }
/>
```

El atributo **name** indica el identificador del Bean que hemos creado con la acción `<jsp:useBean>`.

Los detalles del atributo **property** son una serie de atributos que combinados entre sí permiten asignar el valor a la propiedad del Bean de distinta forma. Así por ejemplo la forma de establecer el valor de la propiedad de un Bean puede ser cualquiera de las que aparecen a continuación:

- `property="*"`
- `property="nombrePropiedad"`
- `property="nombrePropiedad" param="nombreParámetro"`
- `property="nombrePropiedad" value="valorPropiedad"`

El valor de una propiedad de un Bean se puede establecer a partir de varios elementos:

- En el momento del requerimiento de la página a partir de los parámetros existentes en el objeto integrado **request**.
- En el momento de ejecución de la página a partir de la evaluación de una expresión válida de JSP.
- A partir de una cadena de caracteres indicada o como una constante en la propia página.

A continuación se comentan todos los atributos de la acción `<jsp:setProperty>`:

- **name:** nombre o identificador del Bean que se ha instanciado mediante la acción `<jsp:useBean>`.
- **property:** el nombre de la propiedad del Bean cuyo valor se desea establecer. Este atributo puede tener asignado el valor especial asterisco (*). Si indicamos el asterisco, de forma automática la etiqueta iterará sobre todos los parámetros del objeto request correspondiente estableciendo los nombres de las propiedades del Bean que se coincidan con el nombre de los parámetros del objeto request, asignándole el valor del parámetro cuando se de dicha coincidencia. Si un parámetro del objeto request posee el valor vacío ("") no se modificará el valor de la propiedad del Bean. Con el asterisco

podemos establecer el valor de varias propiedades del Bean de una sola vez, más adelante lo veremos mediante un ejemplo.

- **param**: este atributo permite indicar el nombre del parámetro del objeto request que se va a utilizar para establecer el valor de la propiedad del Bean indicada en el atributo **property**. Gracias a este atributo no es necesario que el Bean tenga el mismo nombre de propiedad que el parámetro del objeto request cuyo valor deseamos establecer para la propiedad. Si no se especifica el atributo **param** se asume que el nombre de la propiedad y el nombre del parámetro del objeto request es el mismo.
- **value**: contiene el valor que se va a asignar a la propiedad, puede ser una cadena o una expresión válida. Una acción `<jsp:setProperty>` no puede presentar los atributos **value** y **param** al mismo tiempo.

Volviendo al ejemplo presentado en el punto 17.4.5, podemos modificar el primer script de la siguiente manera:

```
<jsp:useBean id="empleado" scope="session" class="dao.to.Empleado"/>
<jsp:setProperty name="empleado" property="codigo" value="12345"/>
<jsp:setProperty name="empleado" property="nombre" value="Gustavo Coronel"/>
<jsp:setProperty name="empleado" property="usuario" value="gcoronel"/>
```

Como puede apreciar con estas modificaciones ya no es necesario utilizar los scripts.

Ejemplo 17 . 7

En este proyecto ilustraremos el uso de las acciones para el manejo de Beans. Se trata de hacer un componente que permita realizar la suma y el producto de dos números.

Los recursos a implementar son:

Recurso	Descripción
MyMath.java	Componente que implementa la lógica del problema.
operaciones.html	Página HTML que muestra el formulario de ingreso de datos.
operaciones.jsp	Página jsp que procesa los datos ingresados en la página operaciones.html .
paginaError.jsp	Página de error asociada a la página operaciones.html .

A continuación tenemos el script de la clase **MyMath.java**:

```
package logic;

/**
 *
 * @author Gustavo Coronel
 */
public class MyMath {

    private int num1 = 0;
    private int num2 = 0;
    private int suma = 0;
    private int producto = 0;

    private void proceso() {
```

```

        suma = num1 + num2;
        producto = num1 * num2;
    }

    public int getNum1() {
        return num1;
    }

    public void setNum1(int num1) {
        this.num1 = num1;
        proceso();
    }

    public int getNum2() {
        return num2;
    }

    public void setNum2(int num2) {
        this.num2 = num2;
        proceso();
    }

    public int getProducto() {
        return producto;
    }

    public int getSuma() {
        return suma;
    }
}

```

El script de la página **operaciones.html** es el siguiente:

```

<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>Operaciones</title>
    </head>
    <body>
        <h1>Operaciones</h1>
        <form id="form1" name="form1" method="post" action="operaciones.jsp">
            <table width="600">
                <tr>
                    <td width="85">Número 1</td>
                    <td width="503">
                        <input name="num1" type="text" id="num1" size="5" maxlength="5" />
                    </td>
                </tr>
                <tr>
                    <td>Número 2</td>

```

```
<td><input name="num2" type="text" id="num2" size="5" maxlength="5" /></td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="procesar" id="procesar" value="Procesar" />
</td>
</tr>
</table>
</form>
</body>
</html>
```

Algo muy importante en este formulario es el nombre de los campos, note usted que coincide con el nombre de los campos en la clase, esto es fundamental para hacer una asignación automática de los valores cuando se defina en Bean en la página **operaciones.jsp**.

A continuación tenemos el script de **operaciones.jsp**:

```
<%@ page contentType="text/html; charset=utf-8" language="java" %>
<%@ page errorPage="paginaError.jsp" %>
<jsp:useBean id="myMath" class="logic.MyMath" scope="page"/>
<jsp:setProperty name="myMath" property="*"/>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Operaciones</title>
</head>
<body>
<h1>Operaciones</h1>
<table width="268">
<tr>
<td width="96">Número 1</td>
<td width="160"><jsp:getProperty name="myMath" property="num1"/></td>
</tr>
<tr>
<td>Número 2</td>
<td><jsp:getProperty name="myMath" property="num2"/></td>
</tr>
<tr>
<td>Suma</td>
<td><jsp:getProperty name="myMath" property="suma"/></td>
</tr>
<tr>
<td>Producto</td>
<td><jsp:getProperty name="myMath" property="producto"/></td>
</tr>
</table>
<p><a href="operaciones.html">Retornar</a></p>
</body>
```

```
</html>
```

Al inicio de esta página se está definiendo lo siguiente:

- Asociando la página de error, la instrucción es la siguiente:

```
<%@ page errorPage="paginaError.jsp" %>
```

- Creación de Bean, la instrucción es la siguiente:

```
<jsp:useBean id="myMath" class="logic.MyMath" scope="page"/>
```

- Asignando los valores provenientes del formulario, la instrucción es la siguiente:

```
<jsp:setProperty name="myMath" property="*"/>
```

A continuación tenemos el script de la página de error:

```
<%@ page contentType="text/html; charset=utf-8" language="java" %>
<%@ page isErrorPage="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Operaciones</title>
</head>
<body>
    <h1>Operaciones</h1>
    <h2>Error</h2>
    <p>Mensaje: <%= exception.getMessage()%></p>
    <p><a href="operaciones.html">Retornar</a></p>
</body>
</html>
```

La ejecución del proyecto nos muestra en primer lugar el formulario de ingreso de datos, se puede apreciar en el Figura 17.15.

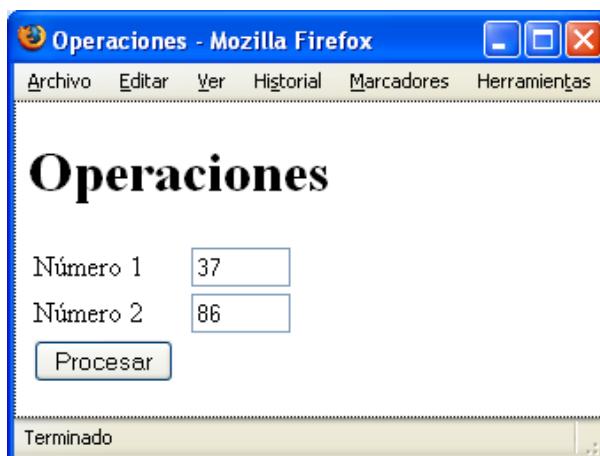


Figura 17 . 15 Formulario de ingreso de datos.

Después de ingresar dos valores y hacer clic en el botón **Procesar** obtenemos los resultados, tal como se ilustra en la Figura 17.16.

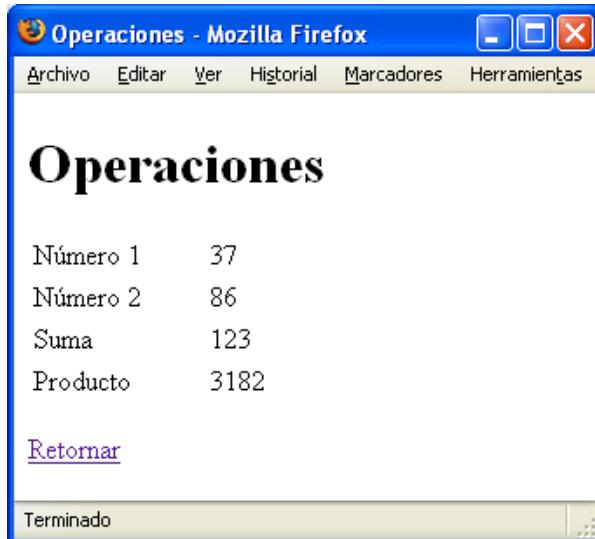


Figura 17 . 16 Resultado después de procesar los datos del formulario.

En caso de ingresar datos errados el resultado se muestra en la Figura 17.17.



Figura 17 . 17 Página de error mostrando un mensaje de error.

17.5. Objetos Implicitos

Para simplificar el código en expresiones y scriplets, tenemos ocho variables definidas automáticamente, a estas variables también se les denomina "**Objetos Implícitos**".

Estos objetos corresponden con objetos útiles del API de servlets (request, response, session, ...) y que en realidad son variables instanciadas de manera automática en el servlet generado a partir del JSP.

17.5.1. Objeto: request

Este es el `HttpServletRequest` asociado con el requerimiento del cliente, y nos permite tener acceso a los parámetros asociados, para lo cual debemos utilizar el método `getParameter(..)`.

el tipo de petición (GET, POST, HEAD, etc.), y las cabeceras HTTP entrantes (cookies, Referer, etc.).

A continuación tenemos un ejemplo ilustrativo de como utilizar el objeto **request** para obtener el valor del parámetro **nombre**:

```
<%
String nombre = request.getParameter("nombre");
%>
```

17.5.2. Objeto: response

Este es el **HttpServletResponse** asociado con la respuesta al cliente.

17.5.3. Objeto: out

Este es el **PrintWriter** usado para enviar la salida al cliente. Sin embargo, para poder hacer útil el objeto **response**, esta es una versión con buffer de **PrintWriter** llamada **JspWriter**.

Podemos ajustar el tamaño del buffer, o incluso desactivar el buffer, usando el atributo **buffer** de la directiva **page**. Este objeto se usa casi exclusivamente en scriptlets ya que las expresiones JSP obtienen un lugar en el stream de salida, y por eso raramente se refieren explícitamente a **out**.

17.5.4 Objeto: session

Este es el objeto **HttpSession** asociado con el requerimiento. Recuerde que las sesiones se crean automáticamente, por lo tanto esta variable es creada incluso si no hubiera una sesión de referencia entrante.

La única excepción es cuando usamos el atributo **session** de la directiva **page** para desactivar las sesiones, en cuyo caso los intentos de referenciar la variable **session** causarán un error en el momento de traducir la página JSP a un servlet.

El objeto **session** también permite almacenar objetos en forma de atributos dentro del ámbito de la sesión, y por lo tanto presenta los mismos métodos que el objeto **request** para este propósito.

A continuación hacemos una descripción de estos métodos:

- void setAttribute(String nombre, Object valor)

Almacena un objeto en la sesión actual. Como se puede comprobar el objeto puede ser cualquier tipo.

- Object getAttribute(String nombre)

Devuelve un objeto almacenado en la sesión actual.

- Enumeration getAttributeNames()

Devuelve dentro de un objeto `java.util.Enumeration` los nombres de todos los atributos disponibles en la sesión.

- void removeAttribute(String nombre)

Elimina un objeto de la sesión actual.

A continuación tenemos un ejemplo de como se puede utilizar el objeto **session** para guardar un objeto de tipo **Empleado**:

```
<%  
    Empleado empleado = new egcc.dao.to.Empleado();  
    empleado.setCodigo("12345");  
    empleado.setNombre("Gustavo Coronel");  
    session.setAttribute( "empleado", empleado );  
%>
```

En este ejemplo el nombre del atributo y el nombre de la variable estan coicidiendo, pero no necesariamente debe ser así.

17.5.5. Objeto: application

Este objeto integrado corresponde con el **ServletContext** obtenido mediante **getServletConfig().getContext()** y representa la aplicación Web a la que pertenece la página JSP actual. Debemos recordar que las aplicaciones Web se encuentran definidas en el fichero de configuración del contenedor JEE, y una página JSP pertenecerá a una aplicación u otra.

El objeto **application** se encuentra disponible en cualquier página JSP y como instancia del interfaz **ServletContext** ofrece todos los métodos de esta interfaz, además de los métodos ya conocidos que se utilizan para almacenar y recuperar los atributos con ámbito de aplicación, no debemos olvidar que el objeto **application** define otro ámbito para los objetos que se almacenan como atributos, y en este caso es el ámbito más general posible.

Todos los objetos que se almacenan en este contexto en forma de atributo están disponibles para todos los servlets y páginas jsp de la aplicación. Supongamos que queremos que todos los servlets y páginas jsp comparten el dato u objeto, entonces, este dato u objeto deberíamos guardarlo a nivel de aplicación utilizando el objeto **application**.

A continuación ilustramos como guardar un dato a nivel de aplicación utilizando el objeto **application**:

```
<%  
    ...  
    application.setAttribute( "igv", "0.19" );  
    ...  
%>
```

Para recuperar este valor debemos utilizar un script como el que se ilustra a continuación:

```
<%  
    ...  
    double igv = Double.parseDouble( application.getAttribute( "igv" ) );  
    ...  
%>
```

17.5.6. Objeto: config

Este objeto integrado es una instancia del interfaz **javax.servlet.ServletConfig**, y su función es la de almacenar información de configuración del servlet generado a partir de la página JSP correspondiente.

Normalmente las páginas JSP no suelen interactuar con parámetros de inicialización del servlet generado, por lo tanto el objeto **config** en la práctica no se suele utilizar.

17.5.7. Objeto: pageContext

JSP presenta una nueva clase llamada **PageContext** para encapsular características de uso específicas del servidor.

Permite acceder al espacio de nombres de la página JSP actual, ofrece también acceso a varios atributos de la página así como una capa sobre los detalles de implementación.

Además el objeto **pageContext** ofrece una serie de métodos que permiten obtener el resto de los objetos integrados de JSP, también nos va a permitir acceder a los atributos pertenecientes a distintos ámbitos.

Podemos distinguir varios grupos de métodos en la clase PageContext, el primero de estos grupos es el que nos permite acceder y obtener los distintos objetos integrados de JSP. Estos métodos no los utilizaremos nunca directamente en nuestras páginas JSP veremos que serán muy útiles cuando tratemos la implementación del mecanismo de etiquetas personalizadas, además estos métodos sí que son utilizados por el servlet equivalente a una página JSP determinada para obtener los distintos objetos integrados y así poder realizar la traducción.

A continuación se muestra un fragmento de un servlet equivalente a una página JSP donde podemos apreciar como se utiliza el objeto **pageContext**:

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class operaciones_jsp extends org.apache.jasper.runtime.HttpJspBase
implements org.apache.jasper.runtime.JspSourceDependent {

    private static final JspFactory _jspxFactory = JspFactory.getDefaultFactory();
    private static java.util.Vector _jspx_dependants;
    private org.apache.jasper.runtime.ResourceInjector _jspx_resourceInjector;

    public Object getDependants() {
        return _jspx_dependants;
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {

        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;

        try {

```

```
response.setContentType("text/html; charset=utf-8");
response.setHeader("X-Powered-By", "JSP/2.1");
pageContext = _jspxFactory.getPageContext(this, request, response,
    "paginaError.jsp", true, 8192, true);
_jspx_page_context = pageContext;
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
...
...
...
```

Los métodos que permiten obtener los objetos integrados son los siguientes:

- **Exception getException()**

Devuelve la excepción que se le ha pasado a la página de error, es decir, devuelve una referencia al objeto integrado exception.

- **JspWriter getOut()**

Devuelve el flujo de salida actual, es decir, el objeto out.

- **Object getPage()**

Devuelve la instancia del servlet para la página actual, es decir, el objeto integrado page.

- **ServletRequest getRequest()**

Devuelve el requerimiento que inició el procesamiento de la página JSP actual, es decir, el objeto integrado request.

- **ServletResponse getResponse()**

Devuelve la respuesta que la página envía al cliente que realizó el requerimiento, es decir, el objeto integrado response.

- **ServletConfig getServletConfig()**

Devuelve un objeto que va a representar la configuración del servlet con el que se corresponde la página JSP actual, es decir, se obtiene una referencia al objeto integrado config.

- **ServletContext getServletContext()**

Devuelve el contexto en el que se ejecuta el servlet que corresponde con la página actual, es decir, se obtiene una referencia al objeto integrado application.

- **HttpSession getSession()**

Devuelve la sesión asociada con el requerimiento de la página actual, si existe se obtendrá una referencia al objeto integrado session.

El siguiente conjunto de métodos es utilizado para controlar el procesamiento de páginas, indicando si se desea incluir un contenido en la página actual o redirigir la ejecución de la página hacia otro recurso.

- void forward(String ruta)

Este método redirige la ejecución de la página actual hacia el recurso indicado mediante su ruta relativa, sin embargo una vez que se haya procesado el recurso al que se ha enviado la ejecución, se vuelve a procesar la página origen. Si el contenido del búfer del objeto out ya ha sido enviado (aunque sea parcialmente) al cliente se producirá un error.

- void include(String ruta)

Incluye el resultado de la ejecución del recurso indicado a través de su ruta relativa. La diferencia con el método anterior es que en la página de origen es posible enviar previamente el contenido del búfer.

Como ya hemos dicho el objeto **pageContext** representa el ámbito de la página actual, por lo tanto poseerá los métodos ya conocidos para acceder y almacenar atributos, pero además de estos métodos el objeto **pageContext** posee otros métodos que nos permiten acceder a atributos de diverso ámbito (aplicación, sesión, requerimiento y la propia página). Estos métodos son los que se muestran a continuación:

- void setAttribute(String nombre, Object valor, int ámbito)

Almacena el atributo correspondiente con el nombre y ámbito indicado. Los ámbitos se encuentran definidos como constantes de tipo entero de la clase **pageContext**, más adelante comentaremos la correspondencia de la constante con el ámbito. Esta clase también presenta el método **setAttribute()** sin el último parámetro, es decir, sin el parámetro del ámbito, en ese caso se almacenará el atributo con ámbito de página.

- Enumeration getAttributeNamesInScope(int ámbito)

Devuelve en un objeto del interfaz **java.util.Enumeration** todos los nombres de los atributos que pertenecen al ámbito indicado.

- Object getAttribute(String nombre, int ámbito)

Devuelve el valor del atributo indicado que pertenece al ámbito que le pasamos por parámetro, también disponemos del método **getAttribute()** sin utilizar el parámetro del ámbito, en este caso nos devolverá el valor del atributo que pertenece al ámbito de la página.

- Object findAttribute(String nombre)

Devuelve el valor del atributo indicado buscándolo en los distintos ámbitos, empezando del más particular al más general. Se devolverá el valor del primer atributo que se encuentre, el orden de consulta de los ámbitos es página, requerimiento, sesión y aplicación.

- int getAttributesScope(String nombre)

Devuelve el ámbito al que pertenece el atributo cuyo nombre le pasamos por parámetro. El orden de búsqueda en los ámbitos es: es página, requerimiento, sesión y aplicación, se devuelve el primer ámbito en el que se encuentre el atributo.

- void removeAttribute(String nombre, int ámbito)

Elimina el atributo indicado en un ámbito específico. También podemos utilizar el método **removeAttribute()** sin el parámetro del ámbito, de esta forma se eliminará el atributo en el ámbito de página.

Las distintas constantes definidas en la clase **PageContext** que se corresponden con los cuatro distintos ámbitos de un atributo son las siguientes (de más particular a más general):

- PAGE_SCOPE

Ámbito de los atributos almacenados en el objeto integrado **pageContext**, representa el ámbito de página.

- REQUEST_SCOPE

Ámbito de los atributos almacenados en el objeto **request**, representa el ámbito de requerimiento.

- SESSION_SCOPE

Ámbito de los atributos almacenados en el objeto integrado **session**, representa el ámbito de sesión.

- APPLICATION_SCOPE

Ámbito de los atributos almacenados en el objeto integrado **application**, representa el ámbito de aplicación.

Por ejemplo, supongamos que tenemos una página jsp donde encontramos el siguiente script:

```
<%  
    request.setAttribute("nombre", "Gustavo");  
    request.setAttribute("pais", "Perú");  
%>  
<jsp:forward page="demo09.jsp" />
```

En el siguiente script ilustramos una forma de leer estos dos atributos en la página **demo09.jsp**:

```
<p>Hola: <%= pageContext.getAttribute("nombre", pageContext.REQUEST_SCOPE) %></p>  
<p>De: <%= pageContext.getAttribute("pais", pageContext.REQUEST_SCOPE) %></p>
```

Como puede observar estamos haciendo uso del objeto **pageContext** y el método **getAttribute(...)**.

17.5.8. Objeto: page

El objeto **page** es una instancia de la clase **java.lang.Object**, y representa la página JSP actual, o para ser más exactos, una instancia de la clase del servlet generada a partir de la página JSP actual. Se puede utilizar para realizar una llamada a cualquiera de los métodos definidos por la clase del servlet.

Utilizar el objeto **page**, es similar a utilizar la referencia a la clase actual, es decir, la referencia **this**. Este objeto pertenece a la categoría de objetos integrados relacionados con los servlets, en este caso representa una referencia al propio servlet.

En nuestras páginas JSP no es muy común utilizar el objeto **page**.

Ejemplo 17 . 8

En este ejemplo desarrollaremos un proyecto para registrar un nuevo cliente en la base de datos EurekaBank.

Para el desarrollo de este requerimiento trabajaremos con dos proyectos que describo a continuación:

Proyecto	Descripción
EurekaBankDAO	<p>Este proyecto implementa la capa de acceso a base de datos.</p> <p>La razón de que sea un proyecto aparte es que facilita su desarrollo, ya que puede ser responsabilidad de una persona o equipo dentro del área de sistema que no necesariamente hace el desarrollo web.</p> <p>Otra razón importante es que los componentes DAO pueden ser compartidos por varias aplicaciones de la empresa.</p>
Cap17Ejemplo08	Este proyecto es el que implementa el requerimiento planteado.

Parte 1

En esta primera parte explicare lo concerniente al proyecto **EurekaBankDAO**, recuerde que el patrón DAO se explica en el Ejemplo 16.4 del capítulo anterior, por consiguiente, lo que haré es implementar las clases necesarias para este requerimiento.

En primer lugar necesitamos la clase **Transfer Object** (TO) al que llamaremos **ClienteTO**, el script de esta clase es el siguiente:

```
package dao.to;

/**
 *
 * @author Gustavo Coronel
 */
public class ClienteTO {

    // Campos
    private String codigo = null;
    private String paterno = null;
    private String materno = null;
    private String nombre = null;
    private String dni = null;
    private String ciudad = null;
    private String direccion = null;
    private String telefono = null;
    private String email = null;

    // Métodos
    public String getCiudad() {
        return ciudad;
    }

    public void setCiudad(String ciudad) {
        this.ciudad = ciudad;
    }
}
```

```
public String getCodigo() {
    return codigo;
}

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

public String getDireccion() {
    return direccion;
}

public void setDireccion(String direccion) {
    this.direccion = direccion;
}

public String getDni() {
    return dni;
}

public void setDni(String dni) {
    this.dni = dni;
}

public String getEmail() {
    return email;
}

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

public String getMaterno() {
    return materno;
}

public void setMaterno(String materno) {
    this.materno = materno;
}

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}
```

```

public String getPaterno() {
    return paterno;
}

public void setPaterno(String paterno) {
    this.paterno = paterno;
}

public String getTelefono() {
    return telefono;
}

public void setTelefono(String telefono) {
    this.telefono = telefono;
}

```

Debido a que se necesita generar el código para el nuevo cliente es necesario interactuar con la tabla **contador**, donde se guardan los correlativos de las diversas tablas. También necesitamos la clase TO para esta tabla, el script es el siguiente:

```

package dao.to;

/**
 *
 * @author Gustavo Coronel
 */
public class ContadorTO {

    // Campos

    private String tabla = null;
    private Integer item = null;
    private Integer longitud = null;

    // Métodos

    public String getTabla() {
        return tabla;
    }

    public void setTabla(String tabla) {
        this.tabla = tabla;
    }

    public Integer getItem() {
        return item;
    }

    public void setItem(Integer item) {

```

```
    this.item = item;
}

public Integer getLongitud() {
    return longitud;
}

public void setLongitud(Integer longitud) {
    this.longitud = longitud;
}

} // ContadorTO
```

Es necesario implementar las interfaces DAO para las tablas **cliente** y **contador**, a las que llamaremos **IClienteDAO** y **IContadorDAO** respectivamente.

El script de la interfaz **IClienteDAO** es el siguiente:

```
package dao.design;

import dao.to.ClienteTO;

/**
 *
 * @author Gustavo Coronel
 */
public interface IClienteDAO {

    public void insertar( ClienteTO clienteTO ) throws Exception;
}
```

El script de la interfaz **IContadorDAO** es el siguiente:

```
package dao.design;

import dao.to.ContadorTO;

public interface IContadorDAO {

    public abstract String generaCodigo(String tabla) throws Exception;
    public abstract ContadorTO consultar(String tabla) throws Exception;
}
```

El diseño de la implementación del componente DAO para la tabla **contador** se muestra en el diagrama de clases de la Figura 17.18.

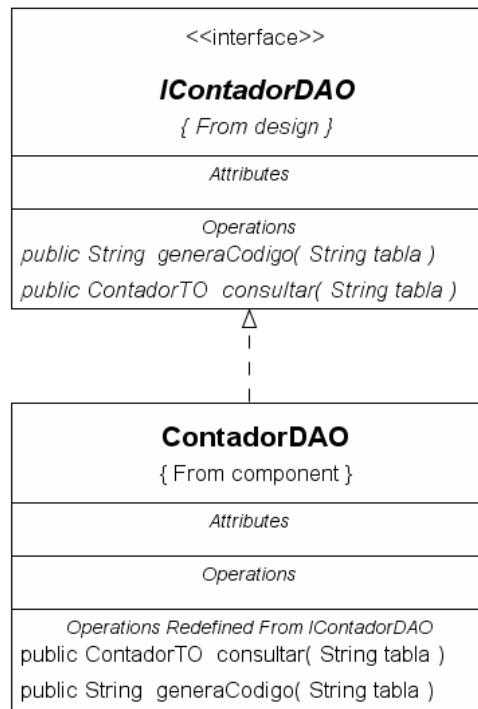


Figura 17 . 18 Diseño del componente DAO de la tabla **contador**.

A continuación tenemos el script que implementación del componente **ContadorDAO**:

```

package dao.component;

import dao.design.IContadorDAO;
import dao.ds.AccesoDB;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import dao.to.ContadorTO;

public class ContadorDAO implements IContadorDAO {

    public ContadorTO consultar(String tabla) throws Exception {
        Connection cn = AccesoDB.getConnection();
        String sql = "select * from contador where vch_conttabla = ?";
        PreparedStatement ps = cn.prepareStatement(sql);
        ps.setString(1, tabla);
        ResultSet rs = ps.executeQuery();
        ContadorTO contadorTO = null;
        if(rs.next()){
            contadorTO = new ContadorTO();
            contadorTO.setTabla(rs.getString("vch_conttabla"));
            contadorTO.setItem(rs.getInt("int_contitem"));
            contadorTO.setLongitud(rs.getInt("int_contlongitud"));
        }
        return contadorTO;
    }
}

```

```
public String generaCodigo(String tabla) throws Exception {
    ContadorTO contadorTO = this.consultar(tabla);
    if( contadorTO == null ){
        throw new Exception("Nombre de tabla no existe.");
    }
    String codigo = null;
    String sql = "update contador set int_contitem = int_contitem + 1 " +
    "where vch_conttabla = ?";
    Connection cn = AccesoDB.getConnection();
    PreparedStatement ps = cn.prepareStatement(sql);
    ps.setString(1, tabla);
    ps.executeUpdate();
    for(int k = 1; k <= contadorTO.getLongitud(); k++){
        codigo = codigo + "0";
    }
    codigo = codigo + String.valueOf(contadorTO.getItem());
    codigo = codigo.substring(codigo.length() - contadorTO.getLongitud());
    return codigo;
}
} // ContadorDAO
```

El método **generaCodigo(String tabla)** es el encargado de generar un nuevo código para una tabla específica que le pasamos como parámetro y actualiza el contador. Note que no maneja una transacción, por el contrario la operación que realiza participa de una transacción que ha sido inicializada por otro método.

El diseño de la implementación del componente DAO para la tabla **cliente** se muestra en el diagrama de clases de la Figura 17.18.

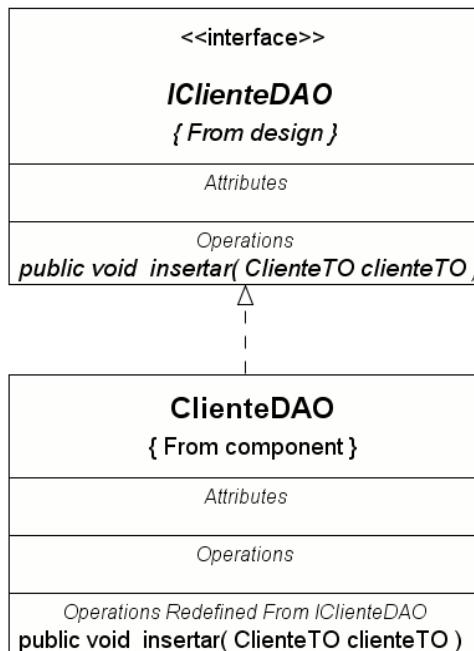


Figura 17 . 19 Diseño del componente DAO para la tabla **cliente**.

A continuación tenemos el script que implementa el componente **ClienteDAO**.

```

package dao.component;

import dao.DAOFactory;
import dao.design.IClienteDAO;
import dao.design.IContadorDAO;
import dao.ds.AccesoDB;
import dao.to.ClienteTO;
import java.sql.Connection;
import java.sql.PreparedStatement;

/**
 *
 * @author Gustavo Coronel
 */
public class ClienteDAO implements IClienteDAO {

    public void insertar(ClienteTO clienteTO) throws Exception {
        Connection cn = null;
        try {
            cn = AccesoDB.getConnection();
            cn.setAutoCommit(false); // Inicio de transacción
            IContadorDAO contadorDAO = DAOFactory.getInstance().getContadorDAO();
            String codigo = contadorDAO.generaCodigo("cliente");
            String sql = "insert into cliente(chr_cliecodigo,vch_cliepaterno,vch_ciematerno," +
                    "vch_cienombre,chr_ciedni,vch_clieciudad,vch_ciedireccion,vch_clitelefono," +
                    "vch_clieemail) values(?,?,?,?,?,?,?,?,?)";
            PreparedStatement ps = cn.prepareStatement(sql);
            ps.setString(1, codigo);
            ps.setString(2, clienteTO.getPaterno());
            ps.setString(3, clienteTO.getMaterno());
            ps.setString(4, clienteTO.getNombre());
            ps.setString(5, clienteTO.getDni());
            ps.setString(6, clienteTO.getCiudad());
            ps.setString(7, clienteTO.getDireccion());
            ps.setString(8, clienteTO.getTelefono());
            ps.setString(9, clienteTO.getEmail());
            ps.executeUpdate();
            clienteTO.setCodigo(codigo);
            cn.commit(); // Confirmar transacción
        } catch (Exception e) {
            try {
                cn.rollback(); // Cancelar transacción
            } catch (Exception e1) {
            }
            throw e;
        }
    }
}

```

}

Para tener acceso al componente **ClienteDAO** debemos implementar el método correspondiente en la clase DAOFactory, tal como se ilustra en el siguiente script:

```
package dao;

import dao.component.ClienteDAO;
import dao.component.ContadorDAO;
import dao.component.EmpleadoDAO;
import dao.design.IClienteDAO;
import dao.design.IContadorDAO;
import dao.design.IEmpleadoDAO;

/**
 *
 * @author Gustavo Coronel
 */
public class DAOFactory {

    private static DAOFactory daoFac;

    static {
        daoFac = new DAOFactory();
    }

    public static DAOFactory getInstance() {
        return daoFac;
    }

    public IEmpleadoDAO getEmpleadoDAO() {
        return new EmpleadoDAO();
    }

    public IContadorDAO getContadorDAO() {
        return new ContadorDAO();
    }

    public IClienteDAO getClienteDAO() {
        return new ClienteDAO();
    }

}
```

Parte 2

En esta segunda parte explicare lo concerniente al proyecto web, al que llamaremos **Cap17Ejemplo08**, los recursos que conforman este proyecto se explican en la siguiente tabla:

Recurso	Descripción
registrarCliente.jsp	Página jsp que muestra el formulario de ingreso de datos y muestra el mensaje al usuario sobre el estado del proceso ejecutado.
RegistraCliente.java	Servlet que recibe los datos del formulario y se comunica con el componente ClienteDAO para ejecutar el proceso de inserción de un nuevo cliente, comunica el la página JSP sobre el resultado del proceso.

Adicionalmente, el proyecto de hacer referencia al proyecto **EurekaBankDAO** y la librería JDBC para MySQL, tal como lo puede verificar en la Figura 17.20.

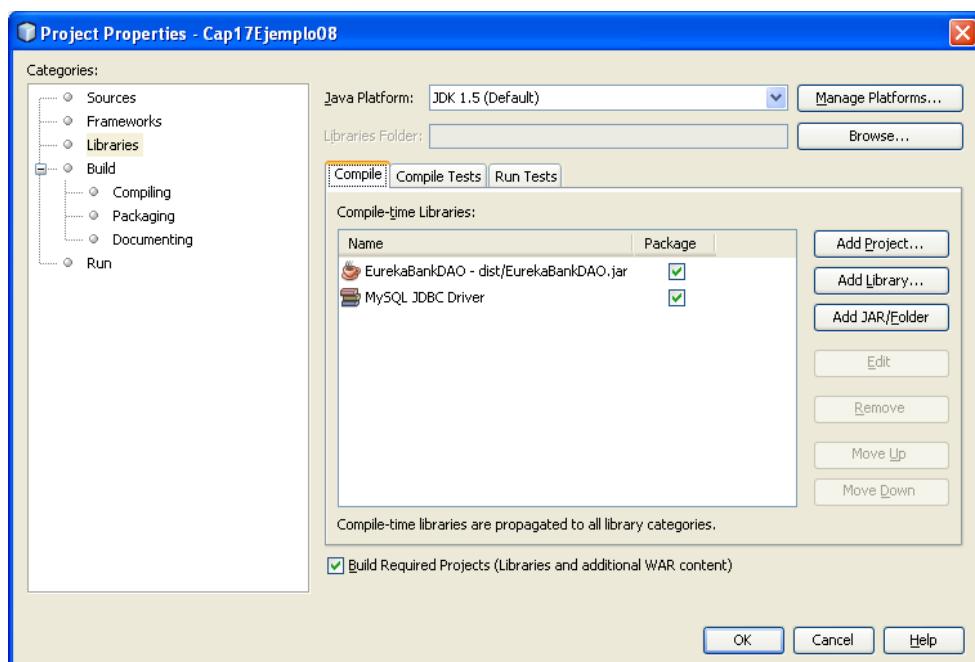


Figura 17 . 20 Configuración del proyecto **Cap17Ejemplo08**.

La Figura 17.21 muestra el diagrama de clases de la aplicación.

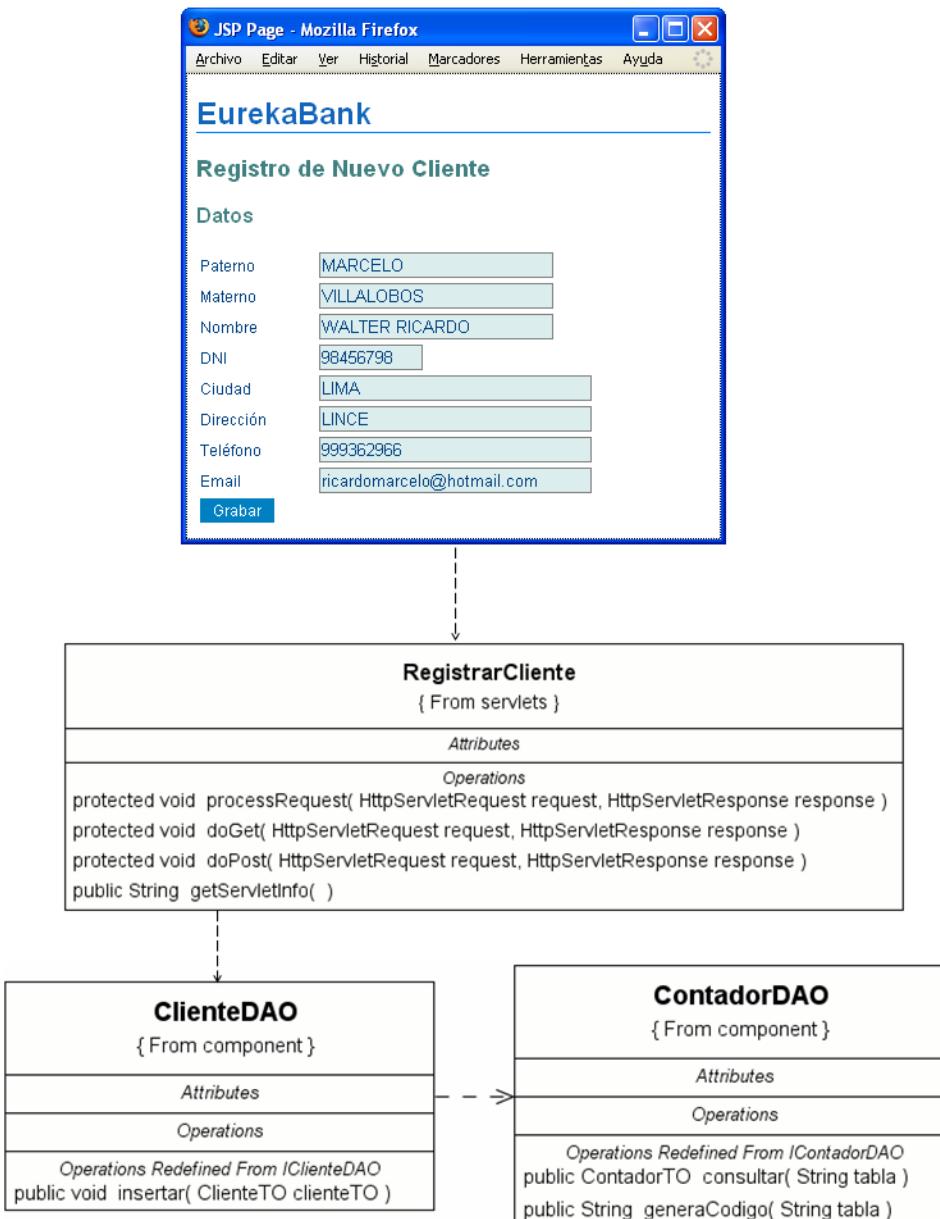


Figura 17 . 21 Diagrama de clases de la aplicación.

El servlet **RegistrarCliente.java** es el que recibe los parámetros del formulario HTML y se comunica con el componente **ClienteDAO** para poder registrar el nuevo cliente en la base de datos.

Para obtener una instancia del componente **ClienteDAO** de utilizar la clase **DAOFactory**, tal como se ilustra a continuación:

```
IClienteDAO clienteDAO = DAOFactory.getInstance().getClienteDAO();
```

A continuación tenemos el script del servlet **RegistrarCliente.java**:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package servlets;
```

```

import dao.DAOFactory;
import dao.design.IClienteDAO;
import dao.to.ClienteTO;
import java.io.IOException;
import java.text.MessageFormat;
import java.util.ArrayList;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author Gustavo Coronel
 */
public class RegistrarCliente extends HttpServlet {

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        ArrayList<String> mensajes = new ArrayList<String>();
        try {
            // Datos
            String paterno = request.getParameter("paterno");
            String materno = request.getParameter("materno");
            String nombre = request.getParameter("nombre");
            String dni = request.getParameter("dni");
            String ciudad = request.getParameter("ciudad");
            String direccion = request.getParameter("direccion");
            String telefono = request.getParameter("telefono");
            String email = request.getParameter("email");
            // Proceso
            ClienteTO clienteTO = new ClienteTO();
            clienteTO.setPaterno(paterno);
            clienteTO.setMaterno(materno);
            clienteTO.setNombre(nombre);
            clienteTO.setDni(dni);
            clienteTO.setCiudad(ciudad);
            clienteTO.setDireccion(direccion);
            clienteTO.setTelefono(telefono);
            clienteTO.setEmail(email);
            IClienteDAO clienteDAO = DAOFactory.getInstance().getClienteDAO();
            clienteDAO.insertar(clienteTO);
        }
    }
}

```

```
        mensajes.add("Proceso ejecutado correctamente.");
        mensajes.add(MessageFormat.format("Código asignado: {0}.", clienteTO.getCodigo()));
    } catch (Exception e) { // Error
        mensajes.add("Error en el proceso.");
        mensajes.add("Error: " + e.getMessage());
    }
    // Salida
    request.setAttribute("mensajes", mensajes);
    RequestDispatcher rd = request.getRequestDispatcher("registrarCliente.jsp");
    rd.forward(request, response);
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

public String getServletInfo() {
    return "Short description";
}// </editor-fold>
}
```

Después que el servlet termina su proceso realiza un **forward** a la página **registrarCliente.jsp**, previamente en el objeto **request** se registra el atributo **mensajes** con el resultado del proceso.

En el formulario de ingreso de datos debemos asegurar que los caracteres que se ingresen se conviertan en mayúsculas, para conseguirlo en los cuadros de texto utilizamos el siguiente código JavaScript:

```
onChange="javascript:this.value=this.value.toUpperCase()"
```

A continuación tenemos el script de la página **registrarCliente.jsp**:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ page import="java.util.ArrayList" %>
<%
    ArrayList<String> mensajes = null;
    if (request.getAttribute("mensajes") != null) {
        mensajes = (ArrayList<String>) request.getAttribute("mensajes");
    }
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link rel="stylesheet" type="text/css" href="css/EurekaBank.css">
<title>JSP Page</title>
</head>
<body>
<h1>EurekaBank</h1>
<h2>Registro de Nuevo Cliente</h2>
<% if (mensajes == null) {%
<div id="datos">
<h3>Datos</h3>
<form name="form1" method="post" action="RegistrarCliente">
<table width="338">
<tr>
<td width="88">Paterno</td>
<td width="238">
<input name="paterno" type="text" class="fieldEdit" id="paterno" size="25"
onchange="javascript:this.value=this.value.toUpperCase()" maxlength="25">
</td>
</tr>
<tr>
<td>Materno</td>
<td>
<input name="materno" type="text" class="fieldEdit" id="materno" size="25"
onchange="javascript:this.value=this.value.toUpperCase()" maxlength="25">
</td>
</tr>
<tr>
<td>Nombre</td>
<td>
<input name="nombre" type="text" class="fieldEdit" id="nombre" size="25"
onchange="javascript:this.value=this.value.toUpperCase()" maxlength="30">
</td>
</tr>
<tr>
<td>DNI</td>
<td>
<input name="dni" type="text" class="fieldEdit" id="dni" size="8" maxlength="8">
</td>
</tr>
<tr>
<td>Ciudad</td>
<td>
<input name="ciudad" type="text" class="fieldEdit" id="ciudad" size="30"
onchange="javascript:this.value=this.value.toUpperCase()" maxlength="30">
</td>
</tr>
<tr>
<td>Dirección</td>
<td>
```

```
<input name="direccion" type="text" class="fieldEdit" id="direccion" size="30"
       maxlength="50" onChange="javascript:this.value=this.value.toUpperCase()">
</td>
</tr>
<tr>
    <td>Teléfono</td>
    <td>
        <input name="telefono" type="text" class="fieldEdit" id="telefono"
               size="30" maxlength="20">
    </td>
</tr>
<tr>
    <td>Email</td>
    <td>
        <input name="email" type="text" class="fieldEdit" id="email" size="30"
               onChange="javascript:this.value=this.value.toLowerCase()" maxlength="50">
    </td>
</tr>
<tr>
    <td colspan="2">
        <input name="grabar" type="submit" class="button" id="grabar" value="Grabar">
    </td>
</tr>
</table>
</form>
</div>
<% }%>
<% if (mensajes != null) {%


<h3>Mensaje</h3>
    <% for (String mensaje : mensajes) {%
        <p><%= mensaje%></p>
    <% }%>
    <p><a href="registrarCliente.jsp">Retornar</a></p>


<% }%>
</body>
</html>
```

Note usted que la página **registrarCliente.jsp** recibe el resultado del proceso a través del atributo **mensajes** del objeto **request**.

Finalmente, cuando ejecutamos el proyecto tenemos inicialmente el formulario de ingreso de datos, tal como lo puede observar en la Figura 17.22.

JSP Page - Mozilla Firefox

EurekaBank

Registro de Nuevo Cliente

Datos

Paterno	MARCELO
Materno	VILLALOBOS
Nombre	WALTER RICARDO
DNI	98456798
Ciudad	LIMA
Dirección	LINCE
Teléfono	999362966
Email	ricardomarcelo@hotmail.com

Grabar

Figura 17 . 22 Formulario de ingreso de datos.

Si el proceso se ejecuta correctamente obtiene un mensaje y el código que se le ha asignado al cliente, tal como lo puede observar en la Figura 17.23.

JSP Page - Mozilla Firefox

EurekaBank

Registro de Nuevo Cliente

Mensaje

Proceso ejecutado correctamente.

Código asignado: 00026.

[Retornar](#)

Figura 17 . 23 Mensaje cuando el proceso se ejecuta correctamente.

Página en Blanco



18

JavaServer Pages Standard Tag Library

JavaServer Pages Standard Tag Library (JSTL) es un conjunto de librerías de etiquetas simples y estándares que encapsulan la funcionalidad principal que es usada comúnmente para escribir páginas JSP.

JSTL responde a la demanda de los desarrolladores de un conjunto de acciones JSP personalizadas para manejar las tareas que necesitan casi todas las páginas JSP, incluyendo procesamiento condicional, bucles, internacionalización, acceso a bases de datos y procesamiento XML. Esto acelarára el desarrollo de las aplicaciones web y elimina la necesidad de los scripting.

Temas a desarrollar:

- 18.1. Introducción
- 18.2. Lenguaje de Expresiones
- 18.3. Funciones EL
- 18.4. Etiquetas del Core

18.1. Introducción

18.1.1. ¿Qué es JSTL?

JavaServer Pages Standard Tag Library (JSTL) es un conjunto de librerías de etiquetas simples y estándares que encapsulan la funcionalidad principal que es usada comúnmente para escribir páginas JSP.

JSTL responde a la demanda de los desarrolladores de un conjunto de acciones JSP personalizadas para manejar las tareas que necesitan casi todas las páginas JSP, incluyendo procesamiento condicional, bucles, internacionalización, acceso a bases de datos y procesamiento XML. Esto acelarára el desarrollo de las aplicaciones web y elimina la necesidad de los scriptlets.

18.1.2. ¿Cuál es el problema con los scriptlets JSP?

La especificación JSP ahora se ha convertido en una tecnología estándar para la creación de sitios Web dinámicos en Java, y el problema es que han aparecido algunas debilidades:

- El código Java embebido en scriptlets es desordenado.

- Un programador que no conoce Java no puede modificar el código Java embebido, anulando uno de los mayores beneficios de las JSP: permitir a los diseñadores y personas que escriben la lógica de presentación que actualicen el contenido de la página.
- El código Java dentro de scriptlets JSP no pueden ser reutilizados por otros JSP, por lo tanto la lógica común termina siendo re-implementada en múltiples páginas.
- La recuperación de objetos fuera del HTTP **request** y **session** es complicada. Es necesario hacer el Casting de objetos y esto ocasiona que tengamos que importar más Clases en los JSP.

18.1.3. ¿Como mejoran esta situación la librería JSTL?

- Debido a que las etiquetas JSTL son XML, estas etiquetas se integran limpia y uniformemente a las etiquetas HTML.
- Las 5 librerías de etiquetas JSTL incluyen la mayoría de funcionalidad que será necesaria en una página JSP. Las etiquetas JSTL son muy sencillas de usarlas para personas que no conocen de programación, a lo mucho necesitarán conocimientos de etiquetas del estilo HTML.
- Las etiquetas JSTL encapsulan la lógica como el formato de fechas y números. Usando los scriptlets JSP, esta misma lógica necesitaría ser repetida en todos los sitios donde es usada, o necesitaría ser movida a Clases de ayuda.
- Las etiquetas JSTL pueden referenciar objetos que se encuentren en el alcance **request** y **session** sin conocer el tipo del objeto y sin necesidad de hacer el Casting.
- Los JSP Expression Language (EL) facilitan las llamadas a los métodos **Get** y **Set** en los objetos Java. EL es usado extensamente en la librería JSTL.

18.1.4. ¿Cuales son las desventajas de los JSTL?

- Los JSTL pueden agregar mayor sobrecarga en el servidor. Los scriptlets y las librerías de etiquetas son compilados a servlets, los cuales luego son ejecutados por el contenedor. El código Java embebido en los scriptlets es básicamente copiado en el servlet resultante. En cambio, las etiquetas JSTL, causan un poco más de código en el servlet. En la mayoría de casos esta cantidad no es mensurable pero debe ser considerado.
- Los scriptlets son más potentes que las etiquetas JSTL. Si desea hacer todo en un script JSP pues es muy probable que insertará todo el código en Java en él. A pesar que las etiquetas JSTL proporciona un potente conjunto de librerías reutilizables, no puede hacer todo lo que el código Java puede hacer. La librería JSTL está diseñada para facilitar la codificación en el lado de presentación que es típicamente encontrado en la capa de Vista si hablamos de la arquitectura Modelo-Vista-Controlador.

18.1.5. Librerías JSTL

Son cinco las librerías de etiquetas independientes, cada una contiene acciones personalizadas dirigidas a un área funcional específica. La siguiente tabla lista cada librería con su prefijo de etiqueta recomendado y la URI por defecto:

Descripción	Prefijo	URI por Defecto
Core	c	http://java.sun.com/jsp/jstl/core
XML Processing	x	http://java.sun.com/jsp/jstl/xml
I18N	fmt	http://java.sun.com/jsp/jstl/fmt

Database Access	sql	http://java.sun.com/jsp/jstl/sql
Function	fn	http://java.sun.com/jsp/jstl/functions

La librería **Core** contiene acciones para las tareas rutinarias, como incluir o excluir una parte de una página dependiendo de una condición en tiempo de ejecución, hacer un bucle sobre una colección de ítems, manipular URLs para seguimiento de sesión, y la correcta interpretación del recurso objetivo, así como acciones para importar contenido de otros recursos y redireccionar la respuesta a una URL diferente.

La librería **XML** contiene acciones para procesamiento XML, incluído validar un documento XML y transformarlo usando XSLT. También proporciona acciones para extraer parte de un documento XML validado, hacer bucles sobre un conjunto de nodos, y procesamiento condicional basado en valores de nodos.

La librería **I18N** contiene etiquetas que son empleados para dar formato específico a elementos dentro de JSP's, estos formatos incluyen aquellos para Fechas, Monedas y Números, así como apoyo para formateo basado en el "Locale/i18n" (Lenguaje del usuario).

La librería **SQL** contiene etiquetas para hacer una rápida y fácil integración con bases de datos. Generalmente se le utiliza para hacer prototipos.

Basado en los principios del patrón Model-View-Controller (MVC), el uso de estos Tags en JSP's resulta en una pésima decisión de diseño, ya que se estaría mezclando la capa de acceso a datos (DAO) con la Vista de presentación (JSP)

La librería **Function** define un conjunto de funciones estándar y un mecanismo para añadir nuevas funciones al Lenguaje de Expresiones.

18.1.6. Instalación de JSTL

Java EE incluye la implementación de las librerías JSTL, pero si el contenedor JEE que estamos utilizando no lo incluye, entonces debemos descargar del proyecto Apache Taglibs la última versión de JSTL e instalarlo en el contenedor JEE ó copiar los archivos JAR en el directorio **WEB-INF/lib** de nuestra aplicación.

Si estamos utilizando NetBeans con GlassFish solo debemos importar las librerías, por ejemplo, si la librería que queremos utilizar es el **core** la instrucción para importarla es:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Pero si vamos a utilizar un contenedor JEE que no implementa por defecto JSTL tendríamos 3 alternativas, deberíamos optar por una de ellas:

1. Habilitar en el IDE las librerías JSTL.
2. Implementar manualmente JSTL en nuestro proyecto web.
3. Instalar en el contenedor JEE las librerías JSTL.

Por ejemplo, a continuación describimos que deberíamos hacer para cada uno de los casos si estaríamos utilizando Tomcat variación 6.

1. Habilitar en el IDE las librerías JSTL.

Para habilitar la librería JSTL en NetBeans debe seguir los siguientes pasos:

- Cargar la ventana de propiedades (Project Properties) del proyecto.
- En la ficha **Compile** hacer clic en el botón **Add Library...** para cargar la ventana **Add Library**, ver Figura 18.1.
- En la ventana **Add Library** de la lista de librerías seleccionar **JSTL 1.1** y hacer clic en el botón **Add Library**, ver Figura 18.2.

- Finalmente en la ventana de propiedades del proyecto hacer clic en el botón **Ok**.

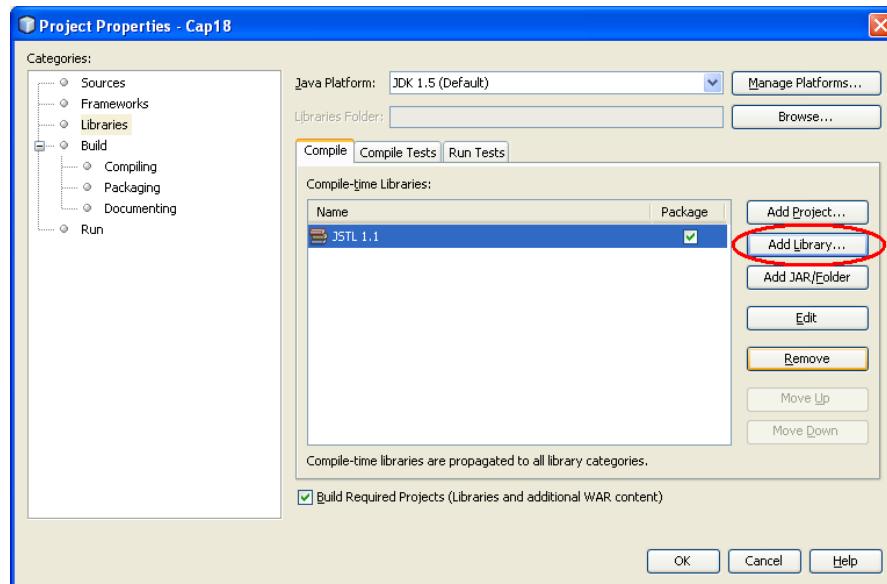


Figura 18 . 1 Ventana Project Properties.

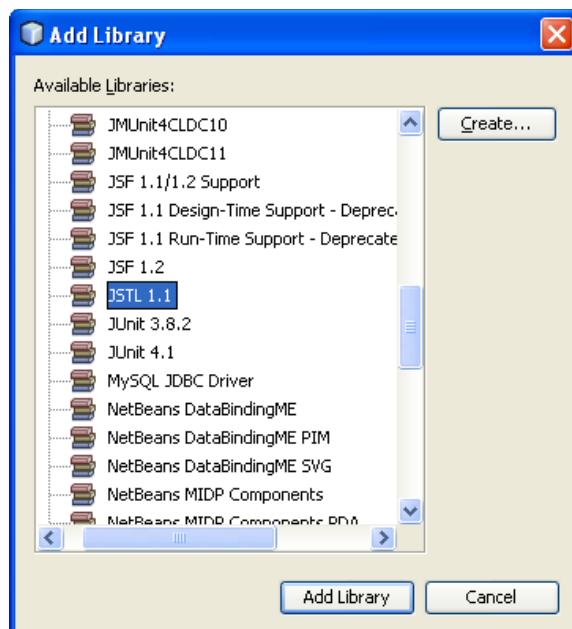


Figura 18 . 2 Ventana Add Library.

2. Implementar manualmente JSTL en nuestro proyecto web.

Tendríamos que descargar las librerías de la página del proyecto **Apache Taglibs** (<http://jakarta.apache.org/taglibs/index.html>), desempaquetar el archivo ZIP en algún directorio fuera del proyecto y copiar los archivos **jstl.jar** y **standard.jar** en el directorio **WEB-INF/lib** de nuestro proyecto.

Si no existe el directorio **lib** en nuestro proyecto debemos crearlo, tal como se ilustra en la Figura 18.3.

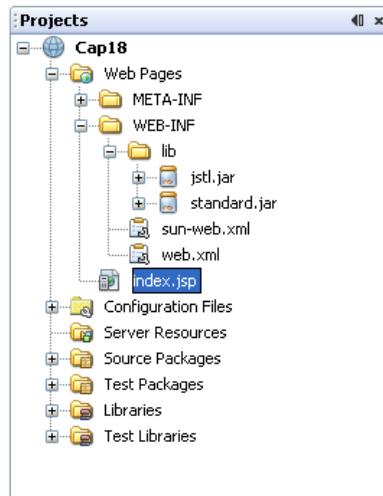


Figura 18 . 3 Carpeta lib con las librerías JSTL.

3. Instalar en el contenedor JEE las librerías JSTL.

La tercera opción que tenemos es descargar las librerías de la página del proyecto Apache Taglibs (<http://jakarta.apache.org/taglibs/index.html>), desempaquetar el archivo ZIP en algún directorio fuera del proyecto y copiamos los archivos **jstl.jar** y **standard.jar** en el directorio **lib** de la carpeta donde se ha instalado tomcat, tal como ilustra en la Figura 18.4.

Una de las tres alternativas es la que debe seleccionar, en nuestro caso elegiremos la tercera, luego cada vez que necesitemos utilizar JSTL alguna librería JSTL debemos importarla utilizando la instrucción correspondiente, tal como se ilustra a continuación:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>
```

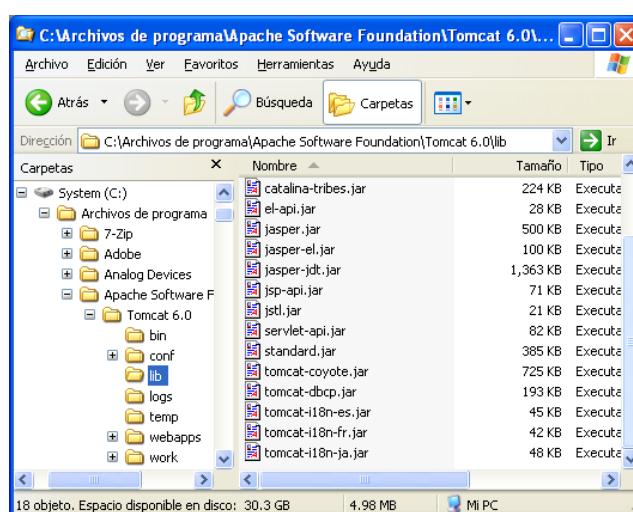


Figura 18 . 4 Librerías de JSTL en el directorio lib de Tomcat.

18.2. Lenguaje de Expresiones

18.2.1. Introducción

Además de las librerías de etiquetas, JSTL define un lenguaje de expresiones, que facilita enormemente el tratamiento de información. JSP usa Java para referenciar atributos dinámicos. Con JSTL ya no es necesario.

Comparemos por ejemplo la lectura de un parámetro:

- Con JSP

```
<%= request.getParameter("nombre") %>
```

- Con JSTL

```
${param.nombre}
```

En JSTL se evita el uso de Java proporcionando un lenguaje de expresiones llamado **Expression Language** (EL). EL no es un lenguaje de programación en sí mismo, su único propósito es:

- Referenciar objetos y sus propiedades.
- Escribir expresiones simples.

EL se basa en las siguientes normas:

- Las expresiones JSTL comienzan con \${ y terminan con }. Lo que hay en medio es tratado como una expresión.
- Las expresiones se componen de:
 - **Identificadores**. Hay once identificadores reservados que corresponden a once objetos implícitos (los veremos luego). El resto de identificadores sirven para crear variables.
 - **Literales**. Son números, cadenas delimitadas por comillas simples o dobles, y los valores true, false y null.
 - **Operadores**. Permiten comparar y operar con identificadores y literales.
 - **Operadores de acceso**. Se usan para referenciar propiedades de los objetos.
- Podemos usar expresiones en cualquier parte del documento, o como valores de los atributos de etiquetas JSTL, exceptuando los atributos **var** y **scope**, que no aceptan expresiones.
- En cualquier sitio donde sea valido colocar una expresión, también será válido colocar más de una. Por ejemplo: value="Hola \${nombre} \${apellidos}".
- Las expresiones pueden contener operaciones aritméticas, comparaciones, operaciones booleanas, y agruparse mediante paréntesis. También pueden decidir cuando una variable existe o no (usando empty).
- Para acceder al campo de un bean, o a un elemento de una colección (array, o Map), se usa el operador punto, o el operador corchete:

```
${bean.propiedad}  
${map.elemento}
```

```
 ${header['User-Agent']}
```

Si un campo de un bean es otro bean, podemos encadenar puntos para acceder a una propiedad del segundo bean:

```
 ${bean1.bean2.propiedad}
```

- Las expresiones pueden aparecer como parte del valor de un atributo de una etiqueta:

```
<c:out value="${15*4}" />
<c:out value="${nombre}" />
<c:if test="${tabla.indice % 2 == 0}">es par</c:if>
```

O independientemente junto a texto estático como el HTML:

```
<input type="text" name="usuario" value="${requestScope.empleado.nombre}" />
```

18.2.2. Operadores

Los operadores y sus significados son los mismos que en otros lenguajes de programación.

Además de los operadores punto (.) y [], EL provee los siguientes operadores:

- Aritméticos: +, - (binario), *, / y div, % y mod, - (unario)
- Lógicos: and, &&, or, ||, not, !
- Relacionales: ==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Las comparaciones se pueden hacer contra otros valores, o contra literales de tipo boolean, string, integer, o de punto flotante
- Empty: El operador **empty** puede ser usado para determinar si un valor es nulo o vacío.
- Condicional: **A ? B : C**. Evalúa **B** o **C**, dependiendo del resultado de la evaluación de **A**.

La tabla 18.1 muestra un resumen d los operadores que podemos utilizar en EL.

Tabla 18 . 1 Resumen de operadores

Operador	Description
.	Accede a una propiedad
[]	Accede a un elemento de un array/lista
()	Agrupa una subexpression
+	Suma
-	Resta o negación de un número
/ o div	División
% o mod	Módulo (resto)
== o eq	Comprueba Igualdad
!= o ne	Comprueba desigualdad
< o lt	Comprueba menor que
> o gt	Comprueba mayor que
<= o le	Comprueba menor o igual que

Operador	Description
>= o gt	Comprueba mayor o igual que
&& o and	Comprueba AND lógico
 o or	Comprueba OR lógico
! o not	Complemento binario booleano
empty	Comprueba un valor vacío (null, string vacío, o una colección vacía)

A continuación tenemos el orden de precedencia de los operadores de mayor a menor, de izquierda a derecha:

- [], .
- () - Usado para cambiar la precedencia de un operador.
- - (unario), not, !, empty
- *, /, div, %, mod
- +, - binario
- <, >, <=, >=, lt, gt, le, ge
- ==, !=, eq, ne
- && (o and)
- || (o or)
- ? :

El operador **empty** comprueba si una colección o cadena es vacía o nula. Ejemplo:

```
 ${empty param.login}
```

Otra manera de hacerlo es usando la palabra clave **null**:

```
 ${param.login == null}
```

En expresiones EL podemos usar números, caracteres, booleanos (palabras clave true, false), y nulls (palabra clave null).

18.2.3. Acceso a datos

El operador punto (.) permite recuperar la propiedad de un objeto por su nombre.

Ejemplo:

```
 ${empleado.nombre}
```

Las propiedades se recuperan usando las convenciones de los JavaBeans. Por ejemplo, en el código anterior, invocábamos el método `empleado.getNombre()` del objeto empleado.

El operador corchete ([]) permite recuperar una propiedad con nombre o indexada por número.

Ejemplo:

```
 ${empleado["nombre"]}  
 ${empleado[0]}  
 ${header['User-Agent']}
```

Dentro del corchete puede haber una cadena literal, una variable, o una expresión.

18.2.4. Palabras Reservadas

Las siguientes palabras están reservadas por EL y no pueden ser usadas como identificadores:

and	eq	gt	true	instanceof
or	ne	le	false	empty
not	lt	ge	null	div
mod				

18.2.5. Objetos implícitos

Ciertos objetos son automáticamente accesibles a cualquier etiqueta JSP.

A través de ellos es posible acceder a cualquier variable de los ámbitos **page**, **request**, **session**, **application**, a parámetros HTTP, cookies, valores de cabeceras, contexto de la página, y parámetros de inicialización del contexto. Todos, excepto **pageContext**, están implementados usando la clase `java.util.Map`.

Tabla 18 . 2 Objetos Implícitos

Objetos implícitos	contiene
pageScope	Variables de ámbito página.
requestScope	Variables de ámbito request .
sessionScope	Variables de ámbito session .
applicationScope	Variables de ámbito application .
param	Parámetros del request como cadenas.
paramValues	Parámetros del request como arreglo de cadenas.
header	Cabeceras del request HTTP como cadenas.
headerValues	Cabeceras del request HTTP como arreglo de cadenas.
cookie	Valores de las cookies recibidas en el request.
initParam	Parámetros de inicialización de la aplicación Web.
pageContext	El objeto PageContext de la página actual.

A continuación tenemos una serie de ejemplos de acceso a los datos utilizando EL.

1. Acceso a Todos los Elementos

```
<c:forEach items="${header}" var="item">
<c:out value="${item}" /><br/>
</c:forEach>
```

2. Acceso por Nombre Utilizando Punto: `objeto.propiedad`

```
Host: <c:out value="${header.Host}" /> <br/>
```

3. Acceso por Nombre con Corchetes: `objeto['propiedad']`

Usar este formato evita que se interprete el nombre como una expresión.
Por ejemplo: queremos la propiedad "**Accept-Language**" no una resta entre las variables **Accept** y **Language**.

```
Host: <c:out value="${header['Host']}" /> <br/>
Accept-Language: <c:out value="${header['Accept-Language']}" /> <br/>
```

18.2.6. Objeto: pageContext

En la Tabla 18.3 se listan algunos de los valores disponibles en el contexto de la página. Existen otros que podemos consultar en la documentación oficial (<http://java.sun.com/javaee/reference/index.jsp>).

Por ejemplo, `pageContext.request.authType` esta ejecutando el método `HttpServletRequest.getAuthType()`. De igual modo podemos acceder al resto de métodos.

Tabla 18 . 3 pageContext

Expresión	Descripción	Ejemplo
<code> \${pageContext.exception.message}</code>	Devuelve una descripción del error cuando la página en curso es una página de error JSP.	"Algo ha ido mal"
<code> \${pageContext.errorData}</code>	Información de un error JSP.	
<code> \${pageContext.request.authType}</code>	Tipo de autenticación usado en la página.	BASIC
<code> \${pageContext.request.remoteUser}</code>	Identificador del usuario (cuando está en uso la autenticación del contenedor).	gustavo
<code> \${pageContext.request.contextPath}</code>	Nombre (también llamado contexto) de la aplicación.	/cap18
<code> \${pageContext.request.cookies}</code>	Array de cookies.	
<code> \${pageContext.request.method}</code>	Método HTTP usado para acceder a la página.	GET
<code> \${pageContext.request.queryString}</code>	Query de la página (el texto de la URL que viene después del PATH)	p1=valor&p2=valor
<code> \${pageContext.request.requestURL}</code>	URL usada para acceder a la página.	http://localhost/app/pagecontext.jsp
<code> \${pageContext.session.new}</code>	Contiene true si la sesión es nueva, false si no lo es.	true
<code> \${pageContext.servletContext.serverInfo}</code>	Información sobre el contenedor JSP.	Sun Java System Application Server 9.1_02

Por ejemplo, construyamos la página `pageContext.jsp`, luego incluyamos en dentro de las etiquetas **BODY** el siguiente código:

```

authType: ${pageContext.request.authType} <br/>
remoteUser: ${pageContext.request.remoteUser} <br/>
contextPath: ${pageContext.request.contextPath} <br/>
cookies: ${pageContext.request.cookies} <br/>
method: ${pageContext.request.method} <br/>
queryString: ${pageContext.request.queryString} <br/>
requestURL: ${pageContext.request.requestURL} <br/>
session.new: ${pageContext.session.new} <br/>
serverInfo: ${pageContext.servletContext.serverInfo} <br/>
message: ${pageContext.exception.message}

```

Cuando ejecutemos esta página tendremos un resultado similar al siguiente:

```

authType:
remoteUser:
contextPath: /Cap18
cookies: [Ljava.util.List;@19d75ee
method: GET
queryString:
requestURL: http://localhost:8084/Cap18/pageContext.jsp
session.new: false
serverInfo: Apache Tomcat/6.0.18
message:

```

Vemos que varios valores aparecen en blanco, y las razones son las siguientes:

- **authType** y **remoteUser** porque no estamos usando autenticación.
- **queryString** porque no hay parámetros, prueba con `pageContext.jsp?var1=Gustavo&var2=Coronel` (`http://localhost:8084/Cap18/pageContext.jsp?var1=Gustavo&var2=Coronel`).
- **message** porque no es una página de error, y por tanto no existe una excepción en el contexto de página.

18.2.7. Objeto: pageContext.errorData

Cuando se produce un error la excepción ocurrida está disponible en el ámbito **request** con el nombre `javax.servlet.error.exception`. Además, se adjunta al contexto de la página, información adicional en forma de bean con nombre **errorData**. Este bean tiene las siguientes propiedades:

Tabla 18 . 4 Propiedades del objeto **errorData**.

Propiedad	Tipo Java	Descripción
requestURI	String	URI de la petición fallida.
servletName	String	Nombre de la página o servlet que lanzó la excepción.
statusCode	int	Código HTTP del fallo.
throwable	Throwable	La excepción que produjo el fallo.

Para declarar una página de error JSP que responda a un código o excepción, podemos usar el descriptor de despliegue, archivo web.xml, a continuación tenemos un ejemplo:

```

<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/errorPage.jsp</location>
</error-page>

```

O una declaración para una página concreta, tal como se ilustra a continuación:

```
<%@page errorPage="errorPage.jsp" %>
```

En la página que mostrará el error (Página de Error) debemos incluir:

```
<%@page isErrorPage="true" %>
```

A continuación tenemos un ejemplo de una página de error denominada **errorPage.jsp**:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page isErrorPage="true" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Página de Error - errorPage</title>
    </head>
    <body>
        <p>URI del requerimiento fallido: ${pageContext.errorData.requestURI}</p>
        <p>Quien lanzo el error: ${pageContext.errorData.servletName}</p>
        <p>Código de error: ${pageContext.errorData.statusCode}</p>
        <p>Excepción: ${pageContext.errorData.throwable.class}</p>
        <p>Mensaje: ${pageContext.errorData.throwable.message}</p>
    </body>
</html>
```

Para probar la página de error podemos utilizar esta otra página que lanza intencionadamente una excepción, como la que se muestra a continuación:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page errorPage="errorPage.jsp" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>demo02.jsp</title>
    </head>
    <body>
        <%
            if (true) {
                throw new Exception("Prueba de error.");
            }
        %>
    </body>
</html>
```

A continuación tenemos el resultado:

```
URI del requerimiento fallido: /Cap18/demo02.jsp
Quien lanzo el error: jsp
Código de error: 500
Excepción: class java.lang.Exception
Mensaje: Prueba de error.
```

18.3. Funciones EL

Las funciones representan un modo de extender la funcionalidad del lenguaje de expresiones. Se usan dentro de cualquier expresión del Lenguaje de Expresiones, por ejemplo:

```
<c:set var="nombre" value="Gustavo"/>
Mi nombre tiene ${fn:length(nombre)} letras.
```

Existen 16 funciones de manipulación de cadenas:

- Alternar una cadena entre mayúsculas/minúsculas: toLowerCase, toUpperCase.
- Obtener una subcadena: substring, substringAfter, substringBefore.
- Eliminar el espacio en blanco en ambos extremos de una cadena: trim.
- Reemplazar caracteres en una cadena: replace.
- Comprobar si una cadena contiene a otra: indexOf, startsWith, endsWith, contains, containsIgnoreCase.
- Convertir una cadena en array: split
- Convertir un array en cadena: join.
- Codificar los caracteres especiales del XML: escapeXml.
- Obtener la longitud de una cadena: length.

La función **length** también puede aplicarse a instancias de java.util.Collection, caso en el cual, devolverá el número de elementos de una colección.

Todas las funciones EL interpretan las variables no definidas o definidas con valor igual a null, como cadenas vacías.

18.4. Etiquetas del Core

18.4.1. Etiqueta: out

Muestra el resultado de una expresión. Su funcionalidad es equivalente a la de <%= %>.

Sintaxis:

- Sin Cuerpo

```
<c:out value="valor" [escapeXml="{true|false}"]  
[default="valorPorDefecto"] />
```

- Con Cuerpo

```
<c:out value="valor" [escapeXml="{true|false}"]>  
    Valor por Defecto  
</c:out>
```

A continuación se describen sus atributos:

Atributo	Descripción	Requerido	Por defecto
value	Información a mostrar	sí	ninguno
default	Información a mostrar por defecto	no	cuerpo
escapeXml	true si debe convertir caracteres especiales a sus correspondientes entidades (por ejemplo, > para <).	no	true

Aquí tenemos algunos ejemplos:

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:out value="Bienvenidos al Mundo de Java"/>
<c:out value="${variableInexistente}" default="No existe el dato requerido."/>
<c:out value="${variableInexistente}">
    No existe la información solicitada.

```

```
</c:out>
```

Para evitar que un nombre sea confundido con una expresión lo ponemos entre comillas:

```
<c:out value="${header['User-Agent']}' default="navegador no especificado"/>
```

Hemos usado comillas simples en **User-Agent** para que no se confundan con las comillas dobles que delimitan el valor del atributo, también podríamos haber usado \"**User-Agent**".

18.4.2. Etiqueta: set

Se utiliza para establecer el valor de una variable o la propiedad de un objeto.

Sintaxis:

- Establece el valor de una variable usando el atributo **value**.

```
<c:set value="valor"  
var="nombreVariable" [scope="{page|request|session|application}"]/>
```

- Establece el valor de una variable usando el contenido del cuerpo.

```
<c:set var="nombreVariable" [scope="{page|request|session|application}"]>  
Contenido del Cuerpo.  
</c:set>
```

- Establece el valor de una propiedad de un objeto usando el atributo **value**.

```
<c:set value="valor"  
target="objeto" property="nombrePropiedad"/>
```

- Establece el valor de una propiedad de un objeto usando el contenido del cuerpo.

```
<c:set target="objeto" property="nombrePropiedad">  
Contenido del Cuerpo.  
</c:set>
```

- Establece un valor diferido.

```
<c:set var="nombreVariable" value="valor-diferido"/>
```

A continuación describimos los atributos:

Atributo	Descripción	Requerido	Por defecto
value	Información a grabar.	no	cuerpo
target	Nombre de un bean cuya propiedad será modificada	no	ninguno
property	Propiedad a modificar	no	ninguna
var	Nombre de la variable en la que guardar.	no	ninguno
scope	Ámbito de la variable en la que grabar la información (page, request, session, o application)	no	page

A continuación tenemos algunos ejemplos:

- Graba el valor 10 en la variable de nombre **suma**, la variable tiene alcance de página.

```
<c:set var="suma" value="${4 + 6}" />
```

- Similar al ejemplo anterior, pero la variable tiene alcance de sesión.

```
<c:set var="suma" scope="session">
    10
</c:set>
```

- En el siguiente ejemplo estamos modificando la propiedad **nombre** del bean **empleadoTO**.

```
<c:set var="dato" value="Claudia" />
<c:set target="${empleadoTO}" property="nombre" value="${dato}" />
```

18.4.3. Etiqueta: remove

Se utiliza para eliminar variables de un alcance en particular.

Sintaxis:

```
<c:remove var="nombreVariable"
    [scope="{page|request|session|application}"]/>
```

A continuación describimos sus atributos:

Atributo	Descripción	Requerido	Por defecto
var	Nombre de la variable a quitar	sí	--
scope	Alcance de la variable a quitar.	no	todos los ámbitos

Cuando no se especifica alcance, la etiqueta busca en todos los alcances, yendo del más específico al más general (**page**, **request**, **session**, **application**), hasta encontrar una variable con ese nombre. Si la variable no se encuentra, la etiqueta finaliza sin error.

18.4.4. Etiqueta: if

Evalúa la condición especificada en el atributo **test**, si es **true** ejecuta el cuerpo de la etiqueta.

Sintaxis

- Sin contenido en el cuerpo.

```
<c:if test="condición"
    var="nombreVariable" [scope="{page|request|session|application}"]/>
```

- Con contenido en el cuerpo.

```
<c:if test="condición"
    [var="nombreVariable"] [scope="{page|request|session|application}"]>
        Contenido del Cuerpo
</c:if>
```

A continuación describimos cada uno de sus atributos:

Atributo	Descripción	Requerido	Por defecto
test	Condición a evaluar. Solo procesa el cuerpo si es true.	sí	--
var	Nombre de la variable para la valor resultante de la evaluación de la condición. El tipo de la variable es booleano.	no	ninguno
scope	Ámbito de la variable.	no	page

En el cuerpo es posible colocar otras etiquetas, incluyendo otras etiquetas **if**. Es útil guardar el resultado de evaluar la condición para evitar repetir los cálculos.

A continuación tenemos un ejemplo donde leemos del alcance **request** el objeto **empleadoTO**, y luego verificando si éste objeto es nulo ó no se decide imprimir un mensaje o sus propiedades.

```
<c:set var="empleadoTO" value="${requestScope.empleadoTO}" />
<c:if test="${empleadoTO != null}">
    <p>Nombre: ${empleadoTO.nombre}</p>
    <p>Paterno: ${empleadoTO.paterno}</p>
    <p>Materno: ${empleadoTO.materno}</p>
</c:if>
<c:if test="${empleadoTO == null}">
    <p>No existe el usuario.</p>
</c:if>
```

18.4.5. Etiquetas: choose, when, otherwise

La etiqueta **choose** no tiene atributos, acepta como hijos uno o más **when** y una etiqueta **otherwise**.

Sintaxis:

```
<c:choose>
    Contenido del Cuerpo (etiquetas <c:when> y <c:otherwise>)
</c:choose>
```

Representa una alternativa dentro de una etiqueta **choose**.

Sintaxis:

```
<c:when test="Condición">
    Contenido del Cuerpo
</c:when>
```

La etiqueta **when** sólo tiene un atributo, tal como se explica a continuación:

Atributo	Descripción	Requerido	Por defecto
Test	Condición a evaluar.	sí	--

La etiqueta **otherwise** representa la última alternativa dentro de la etiqueta **choose** y no tiene atributos.

Sintaxis:

```
<c:otherwise>
    Bloque Condicional.
</c:otherwise>
```

Ejecuta el cuerpo de la primera etiqueta **when** cuya condición se evalúe como verdadera (true), o el cuerpo de la etiqueta **otherwise** (si existe) cuando ninguna de las condiciones de la etiqueta **when** resulte verdadera.

A continuación tenemos un ejemplo ilustrativo de como utilizar estas etiquetas:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Demo 05</title>
    </head>
    <body>
        <c:set var="nota" value="${param.nota}" />
        Nota:<c:out value="${nota}" /><br>
        Condición:
        <c:choose>
            <c:when test="${nota >= 14}">Aprobado</c:when>
            <c:when test="${nota >= 11}">Asistente</c:when>
            <c:when test="${nota >= 0}">Desaprobado</c:when>
            <c:otherwise>Nota Fuera de Rango</c:otherwise>
        </c:choose>
    </body>
</html>
```

En este ejemplo recogemos el parámetro **nota** en la variable del mismo nombre y lo evaluamos para obtener la condición de **Aprobado**, **Asistente** o **Desaprobado**.

18.4.6. Etiqueta: forEach

Repite el contenido de su cuerpo a través de una colección de objetos, o un número de veces.

Sintaxis 1:

```
<c:forEach [var="nombreVariable"] items="colección"
    [varStatus="variableEstado"]
    [begin="inicio"] [end="fin"] [step="paso"]>
```

Contenido del Cuerpo

```
</c:forEach>
```

Sintaxis 2:

```
<c:forEach [var="nombreVariable"]
    [varStatus="variableEstado"]
    begin="inicio" end="fin" [step="paso"]>
```

Contenido del Cuerpo

</c:forEach>

A continuación describimos cada uno de sus atributos:

Atributo	Descripción	Requerido	Por defecto
items	Colección de items sobre los que se itera.	no	ninguno
begin	Elemento con el que se inicia la iteración (0=primero).	no	0
end	Elemento con el que finaliza la iteración.	no	último
step	Procesa solo cada step elementos.	no	1
var	Nombre de la variable con la que se expone el elemento actual.	no	ninguno
varStatus	Nombre de la variable con la que se expone el estado de la iteración actual.	no	ninguno

La variable varStatus tiene propiedades que describen el estado de la iteración, tal como se describe a continuación:

Atributo	Tipo	Requerido
begin	número	El valor del atributo begin.
current	número	El elemento actual.
end	número	El valor del atributo end.
index	número	Índice del elemento actual dentro de la colección.
count	número	Número de iteración (empezando en 1).
first	boolean	Indica si estamos en la primera iteración.
last	boolean	Indica si estamos en la última iteración.
step	número	El valor del atributo step.

A continuación tenemos algunos ejemplos:

- Muestra el mensaje **Perú Campeón** 10 veces.

```
<c:forEach var="n" begin="1" end="10">
    <p>${n} Perú Campeon</p>
</c:forEach>
```

- Imprime el nombre y apellido paterno de una lista de empleados.

```
<c:set var="mensaje" value="${requestScope.mensaje}" />
<c:set var="lista" value="${requestScope.lista}" />
<c:out value="${mensaje}" /><br>
<c:if test="${fn:length(lista) == 0}">
    <c:out value="No hay elementos." /><br>
</c:if>
```

```
<c:forEach items="${lista}" var="item">
    ${item.nombre} ${item.paterno}<br>
</c:forEach>
```

18.4.7. Etiqueta: forTokens

Permite descomponer una cadena en tokens. Analizar una cadena se llama parsing. Las partes en las que se descompone una cadena se llaman tokens.

Sintaxis:

```
<c:forTokens items="cadenaDeTokens" delims="delimitadores"
    [var="nombreVariable"]
    [varStatus="variableEstado"]
    [begin="inicio"] [end="fin"] [step="paso"]>
    Contenido del Cuerpo.
</c:forTokens>
```

A continuación describimos cada uno de sus atributos:

Atributo	Descripción	Requerido	Por defecto
var	Nombre del atributo con el que se expone el token actual	no	ninguno
items	Cadena de tokens sobre la cual iterar.	sí	ninguno
delims	Conjunto de delimitadores (caracteres que separan los tokens en la cadena).	sí	ninguno
varStatus	Nombre de la variable con la que se expone el estado de la iteración actual.	no	ninguno
begin	Elemento con el que se inicia la iteración (0=primero).	no	0
end	Elemento con el que finaliza la iteración.	no	último
step	Procesa solo cada step elementos.	no	1

Los delimitadores que aparecen uno tras otro son tratados como si solo existiera uno.

Por ejemplo, si tenemos el siguiente script en una página JSP:

```
<c:set var="paises" value="Perú,España,,Argentina,Brasil,Colombia"/>
<ul>
    <c:forTokens items="${paises}" delims="," var="pais">
        <li><c:out value="${pais}" /></li>
    </c:forTokens>
</ul>
```

La salida que genera es la siguiente:

```
<ul>
    <li>Perú</li>
    <li>España</li>
    <li>Argentina</li>
    <li>Brasil</li>
    <li>Colombia</li>
```


18.4.8. Etiqueta: import

Importa el contenido de un recurso basado en una URL.

Sintaxis:

- Contenido del recurso "en línea" o exportado como un objeto **String**.

```
<c:import url="url" [context="contexto"]  
[var="nombreVariable"] [scope="{page|request|session|application}"]  
[charEncoding="charEncoding"]>
```

Contenido del cuerpo es opcional para etiquetas <c:param>.

```
</c:import>
```

- Contenido del recurso es exportado como un objeto **Reader**.

```
<c:import url="url" [context="contexto"]  
varReader="nombreVariableReader"  
[charEncoding="charEncoding"]>
```

Contenido del cuerpo donde nombreVariableReader es utilizada por otra acción.

```
</c:import>
```

A continuación describimos cada uno de sus atributos:

Atributo	Descripción	Requerido	Por defecto
url	URL para importar el recurso. Es válido cualquier protocolo soportado por <code>java.net.URL</code> . Es decir, al menos HTTP, HTTPS, FTP, file, y JAR.	sí	--
context	Nombre del contexto cuando se accede a un recurso URL relativo que pertenece a otro contexto.	no	contexto actual
var	Nombre de la variable con la que se expone el contenido del recurso.	no	muestra en página
scope	Ámbito de la variable.	no	page
charEncoding	Juego de caracteres con el que importar los datos.	no	ISO-8859-1
varReader	Nombre de una variable con la que exponer el <code>java.io.Reader</code> con que se lee la URL.	no	ninguno

En HTML, la URL relativa `/images/smiley.jpg` lee un archivo de la raíz del servidor web, pero en JSTL, esa misma URL lee el fichero de la raíz de nuestra aplicación web. Si quisieramos que la URL fuera relativa a otra aplicación Web usaríamos:

```
<c:import context="/otraAplicacion" url="/directorio/pagina.jsp"/>
```

Cuando dos páginas están en la misma aplicación web, comparten el mismo ámbito request, session, y application. Si las páginas están en diferente aplicación y son accedidas mediante **c:import**, ambas comparten el ámbito request.

18.4.9. Etiqueta: param

Añade parámetros a requerimientos de una URL. Anidadas a acciones **<c:import>**, **<c:url>**, **<c:redirect>**.

Sintaxis:

- El valor del parámetro se especifica en el atributo “**value**”.

```
<c:param name="nombreParámetro" value="valorParámetro"/>
```

- El valor del parámetro se especifica en el contenido del cuerpo.

```
<c:param name="nombreParámetro">
    valorParámetro
</c:param>
```

A continuación describimos cada uno de sus atributos:

Atributo	Descripción	Requerido	Por defecto
name	nombre del parámetro	sí	--
value	valor del parámetro	no	valor vacío

Si quieres que un bean sea accesible en otra página, sería engorroso convertirlo en una cadena de texto, es mejor usar **c:set** para guardar esta información en un contexto como **session** y proceder a recuperarlo desde allí.

A continuación tenemos algunos ejemplos:

- Cuando queremos reutilizar un texto lo grabamos en una variable.

```
<c:import url="http://www.minedu.gob.pe/noticias/index.php?id=7539"
           var="sitio"/>
<c:out value="${sitio}" />
```

- Podemos hacer que los contenidos leídos sirvan de entrada a otra etiqueta colocando **c:import** en el cuerpo de dicha etiqueta (suponiendo que esta última lo admite).

```
<c:import url="http://www.minedu.gob.pe/noticias/index.php">
    <c:param name="id" value="7539"/>
</c:import>
```

18.4.10 Etiqueta: redirect

Envía una dirección HTTP al cliente como respuesta, abortando el proceso de la página actual.

Sintaxis:

- Sin contenido en el cuerpo.

```
<c:redirect url="recurso" [context="contexto"]/>
```

- Con contenido en el cuerpo para especificar el valor de los parámetros.

```
<c:redirect url="recurso" [context="contexto"]>  
    Etiquetas c:param  
</c:redirect>
```

A continuación describimos cada uno de sus atributos.

Atributo	Descripción	Requerido	Por defecto
url	URL a la que redirigir.	sí	--
context	Nombre del contexto cuando se accede a un recurso URL relativo que pertenece a otro contexto.	no	contexto actual

A continuación tenemos un ejemplo ilustrativo:

```
<c:redirect url="http://www.minedu.gob.pe/noticias/index.php">  
    <c:param name="id" value="7539"/>  
</c:redirect>
```

18.4.11. Etiqueta: url

Esta etiqueta procesa y reescribe una dirección URL si es necesario. Sólo direcciones URL relativas son re-escritas. Las URL absolutos no son re-escritas para evitar situaciones donde las URL externas puede ser re-escritas y exponer la identificación de la sesión.

Sintaxis:

- Sin contenido en el cuerpo.

```
<c:url value="valor" [context="contexto"]  
    [var="nombreVariable"] [scope="{page|request|session|application}"]/>
```

- Con contenido en el cuerpo para especificar los parámetros.

```
<c:url value="valor" [context="contexto"]  
    [var="nombreVariable"] [scope="{page|request|session|application}"]>
```

Etiquetas <c:param>

```
</c:url>
```

A continuación describimos cada uno de sus parámetros.

Atributo	Descripción	Requerido	Por defecto
value	URL a ser procesada.	sí	--
context	Nombre del contexto cuando se accede a un recurso URL relativo que pertenece a otro contexto.	no	contexto actual
var	Nombre de la variable con la que se expone el contenido del recurso.	no	ningún

scope	Ámbito de la variable.	no	page
-------	------------------------	----	------

Esta etiqueta es útil por dos motivos:

- Preserva la información de sesión codificada en la URL.
- Permite usar URLs relativas a la aplicación web, no a la raíz del servidor.

A continuación tenemos un ejemplo ilustrativo:

```
<c:url value="demo05.jsp" var="destino">
    <c:param name="nota" value="15"/>
</c:url>
<a href="#">Enlace

```

18.4.12. Etiqueta: catch

La acción **catch** permite a las páginas jsp manejar errores desde generados desde cualquier otra acción, y permite el manejo de errores para múltiples acciones a la vez.

Las acciones de representan el proceso central de la página no deberían ser encapsuladas en una etiqueta **catch**, por lo que excepción debe propagarse a una página de error, mientras que las acciones de importancia secundaria si deberían ser encapsuladas en una etiqueta **catch**.

La excepción generada es almacenada en la variable especificada en el atributo **var**, que siempre tiene alcance de página. Si no se produce ninguna excepción, la variable especifica en el atributo **var** se removida en caso de que exista.

Sintaxis:

```
<c:catch [var="nombreVariable"]>
```

Acciones Anidadas.

```
</c:catch>
```

A continuación describimos sus atributos.

Atributo	Descripción	Requerido	Por defecto
var	Variable para exponer información sobre el error	no	ninguno

A continuación tenemos un ejemplo ilustrativo.

```
<c:catch var="error1">
    <fmt:parseNumber var="total" value="ABCD"/>
    Importe: ${dato.total}<br>
</c:catch>
<c:if test="#">not empty error1">
    Lo sentimos, no existe el dato solicitado.<br>
    Error: ${error1}
</c:if>
```

El bloque **if** solo se ejecutará si alguna acción que se encuentran dentro del bloque **catch** genera algún error.

Ejemplo 18 . 1

Se tiene el requerimiento de una aplicación que permita a los empleados registrar los depósitos y retiros que los clientes del banco necesitan hacer por ventanilla.

Para atender este requerimiento es importante definir primero que patrones de diseño utilizaremos para el desarrollo de la aplicación.

Patrón de Diseño	Descripción
Data Access Object DAO	Este patrón de diseño permite crear una capa abstracta para acceder a una fuente de datos.
Model View Controller MVC	Este patrón de diseño divide una aplicación web en tres capas. Para el desarrollo de este ejemplo se utilizará MVC Model 2.

Luego debemos definir que tecnologías se utilizaran en cada una de las capas.

Capa	Descripción
Acceso a Datos	Se utilizará el patrón de diseño DAO con JDBC.
Model	Para el desarrollo de esta capa se utilizará clases simples (POJO).
Controller	Para el desarrollo de esta capa se utilizará Servlets.
View	Para el desarrollo de esta capa se utilizará páginas JSP y JSTL.

La Figura 18.5 muestra el esquema de aplicación de los patrones de diseño MVC (Modelo 2) y DAO.

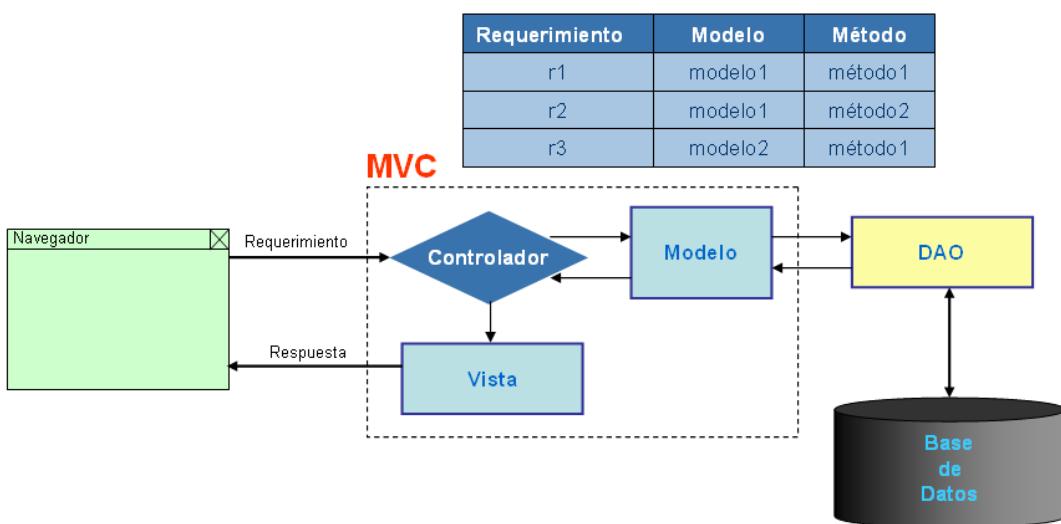


Figura 18 . 5 Esquema de aplicación del los patrones de diseño MVC y DAO.

La 18.6 se muestra la tecnología utilizada en cada una de las capas del patrón de diseño MVC.

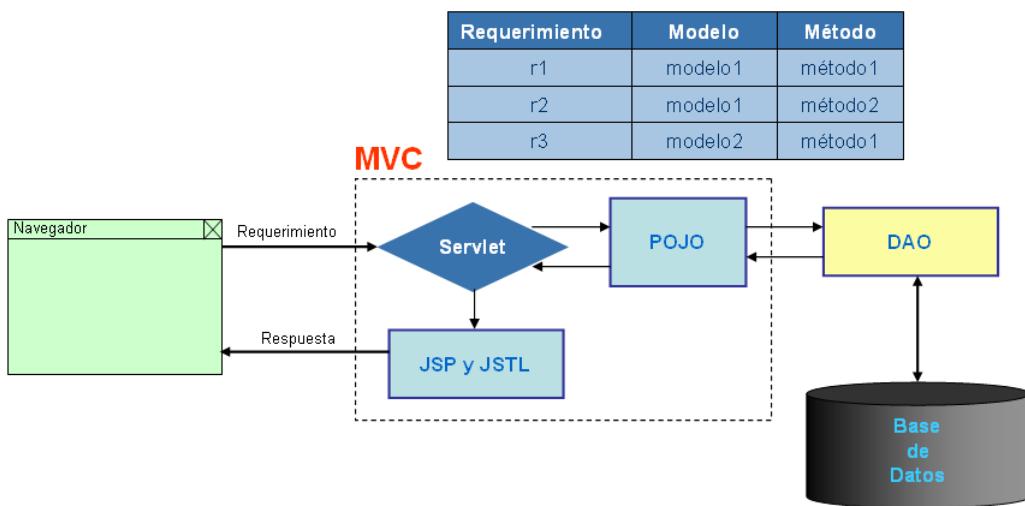


Figura 18 . 6 Tecnología utilizada en la implementación del patrón de diseño MVC.

- La Vista (View) es responsable de generar el código HTML que se envía al navegador, su implementación se realiza con páginas JSP y JSTL.
- El Controlador (Controller) es responsable de recibir los requerimientos de del usuario, desde el navegador, determina cual es el modelo a utilizar y el métodos a ejecutar, el resultado se lo comunica a la vista para que genere el código HTML y se lo envíe al navegador.
- El Modelo (Model) es responsable de implementar las reglas de negocio, si es necesario acceder a la fuente de datos debe hacerlo a través de la capa DAO.
- La capa DAO permite el acceso a la fuente de datos, cualquier operación que se requiera con respecto a la fuente de datos debe desarrollarse en esta capa. El acceso a la fuente de datos es totalmente transparente, la capa Model no necesita y no tiene por que enterarse cual es el proveedor de datos.

Tabajaremos con dos proyectos que describo a continuación:

Proyecto	Descripción
EurekaBankDAO	Este proyecto es la continuación del proyecto trabajado en el Ejemplo 17.8 del capítulo anterior.
Cap18Ejemplo01	Este proyecto es el que implementa el requerimiento planteado.

Parte 1

Diseño e Implementación de la Capa de Acceso a la Fuente de Datos

La Figura 18.7 muestra el Diagrama de Clases del diseño de la capa de acceso a la fuente de datos.

- La clase **AccesoDB** provee el objeto **Connection** para acceder a la fuente de datos.
- En la interfaz **ICuentaDAO** se definen los métodos que debe implementar el componente **CuentaDAO**.
- El componente **CuentaDAO** implementa la interfaz **ICuentaDAO**, para acceder a la fuente debe obtener el objeto **Connection** desde la clase **AccesoDB**.
- Para acceder a una instancia del componente **CuentaDAO** debe hacer a traves de la clase **DAOFactory**.

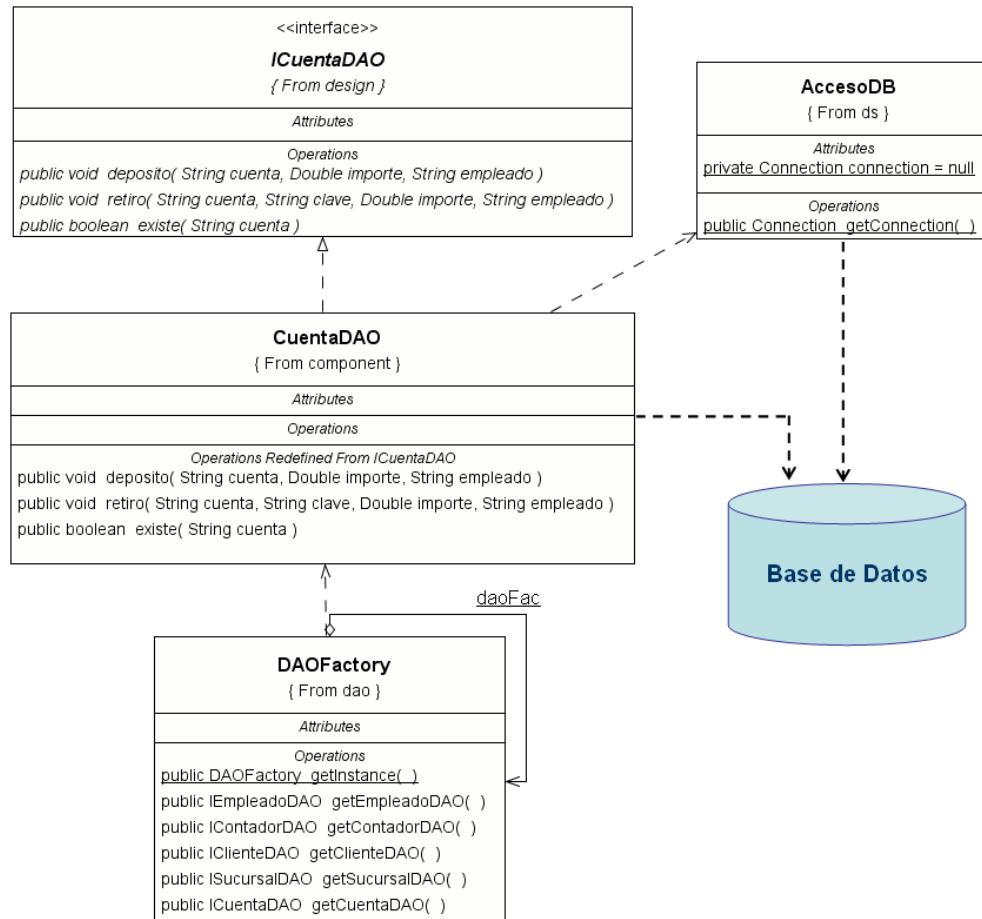


Figura 18 . 7 Diseño de la capa de acceso a la fuente de datos – DAO.

A continuación tenemos el script de la clase **AccesoDB**:

```

package dao.ds;

import java.sql.Connection;
import java.sql.DriverManager;

/**
 * 
 * @author Gustavo Coronel
 */
public class AccesoDB {

    private static Connection connection = null;

    public static Connection getConnection() throws Exception {
        String driver = "com.mysql.jdbc.Driver";
        String url = "jdbc:mysql://localhost/eurekabank";
        String user = "root";
        String pwd = "";
        if (connection == null) {
            Class.forName(driver).newInstance();
        }
        return connection;
    }
}
  
```

```

        connection = DriverManager.getConnection(url, user, pwd);
    }
    return connection;
} // getConnection

}

```

A continuación tenemos el script de la interfaz **ICuentaDAO**:

```

package dao.design;

/**
 *
 * @author Gustavo Coronel
 */
public interface ICuentaDAO {

    public abstract void deposito(String cuenta, Double importe, String empleado)
        throws Exception;

    public abstract void retiro(String cuenta, String clave, Double importe, String empleado)
        throws Exception;

    public abstract boolean existe(String cuenta) throws Exception;

}

```

A continuación tenemos el script de la clase **CuentaDAO**:

```

package dao.component;

import dao.design.ICuentaDAO;
import dao.ds.AccesoDB;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 *
 * @author Gustavo Coronel
 */
public class CuentaDAO implements ICuentaDAO {

    public void deposito(String cuenta, Double importe, String empleado) throws Exception {
        String query = "{call usp_deposito(?, ?, ?, ?)}";
        Connection cn = AccesoDB.getConnection();
        cn.setAutoCommit(true);
        CallableStatement cs = cn.prepareCall(query);
        cs.registerOutParameter(1, java.sql.Types.VARCHAR);

```

```
        cs.setString(2, cuenta);
        cs.setDouble(3, importe);
        cs.setString(4, empleado);
        cs.execute();
        String estado = cs.getString(1);
        cs.close();
        cs = null;
        if (!estado.equals("ok")) {
            throw new Exception(estado);
        }
    }

    public void retiro(String cuenta, String clave, Double importe, String empleado) throws Exception {
        String query = "{call usp_retiro(?, ?, ?, ?, ?)}";
        Connection cn = AccesoDB.getConnection();
        cn.setAutoCommit(true);
        CallableStatement cs = cn.prepareCall(query);
        cs.registerOutParameter(1, java.sql.Types.VARCHAR);
        cs.setString(2, cuenta);
        cs.setString(3, clave);
        cs.setDouble(4, importe);
        cs.setString(5, empleado);
        cs.execute();
        String estado = cs.getString(1);
        cs.close();
        cs = null;
        if (!estado.equals("ok")) {
            throw new Exception(estado);
        }
    }

    public boolean existe(String cuenta) throws Exception {
        String sql = "select count(*) as dato from cuenta where chr_cuencodigo = ?";
        PreparedStatement ps = AccesoDB.getConnection().prepareStatement(sql);
        ps.setString(1, cuenta);
        ResultSet rs = ps.executeQuery();
        rs.next();
        int dato = rs.getInt("dato");
        return (dato == 1);
    }
}
```

A continuación tenemos el script de la clase **DAOFactory**:

```
package dao;

import dao.component.ClienteDAO;
import dao.component.ContadorDAO;
import dao.component.CuentaDAO;
```

```
import dao.component.EmpleadoDAO;
import dao.component.SucursalDAO;
import dao.design.IClienteDAO;
import dao.design.IContadorDAO;
import dao.design.ICuentaDAO;
import dao.design.EmpleadoDAO;
import dao.design.SucursalDAO;
import dao.to.SucursalTO;

/**
 *
 * @author Gustavo Coronel
 */
public class DAOFactory {

    private static DAOFactory daoFac;

    static {
        daoFac = new DAOFactory();
    }

    public static DAOFactory getInstance() {
        return daoFac;
    }

    public IEmpleadoDAO getEmpleadoDAO() {
        return new EmpleadoDAO();
    }

    public IContadorDAO getContadorDAO() {
        return new ContadorDAO();
    }

    public IClienteDAO getClienteDAO() {
        return new ClienteDAO();
    }

    public ISucursalDAO getSucursalDAO() {
        return new SucursalDAO();
    }

    public ICuentaDAO getCuentaDAO(){
        return new CuentaDAO();
    }

}
```

Otros componentes de la capa DAO son:

- La clase **EmpleadoTO** representa al objeto TO de la entidad Empleado, a continuación tenemos su script:

```
package dao.to;

/**
 *
 * @author Gustavo Coronel
 */
public class EmpleadoTO {

    private String codigo;
    private String paterno;
    private String materno;
    private String nombre;
    private String ciudad;
    private String direccion;
    private String usuario;
    private String clave;

    public String getCiudad() {
        return ciudad;
    }

    public void setCiudad(String ciudad) {
        this.ciudad = ciudad;
    }

    public String getClave() {
        return clave;
    }

    public void setClave(String clave) {
        this.clave = clave;
    }

    public String getCodigo() {
        return codigo;
    }

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

    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
}
```

```

    }

    public String getMaterno() {
        return materno;
    }

    public void setMaterno(String materno) {
        this.materno = materno;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getPaterno() {
        return paterno;
    }

    public void setPaterno(String paterno) {
        this.paterno = paterno;
    }

    public String getUsuario() {
        return usuario;
    }

    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
}

```

- La clase **SucursalTO** representa el objeto TO de la entidad Sucursal, a continuación tenemos su script:

```

package dao.to;

/**
 *
 * @author Gustavo Coronel
 */
public class SucursalTO {

    private String codigo = null;
    private String nombre = null;
    private String ciudad = null;
}

```

```
private String direccion = null;
private String contcuenta = null;

public String getCodigo() {
    return codigo;
}

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

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getCiudad() {
    return ciudad;
}

public void setCiudad(String ciudad) {
    this.ciudad = ciudad;
}

public String getDireccion() {
    return direccion;
}

public void setDireccion(String direccion) {
    this.direccion = direccion;
}

public String getContcuenta() {
    return contcuenta;
}

public void setContcuenta(String contcuenta) {
    this.contcuenta = contcuenta;
}

}
```

- La interfaz **IEmpleadoDAO** define los métodos que debe implementar la clase EmpleadoDAO, su script es el siguiente:

```
package dao.design;
```

```

import dao.to.EmpleadoTO;
import java.util.ArrayList;

/**
 *
 * @author Gustavo Coronel
 */
public interface IEmpleadoDAO {

    public abstract ArrayList<EmpleadoTO> consultarPorSucursal(String sucursal) throws Exception;

    public abstract EmpleadoTO consultarPorUsuario(String usuario) throws Exception;

    public abstract ArrayList<EmpleadoTO> consultarPorPaterno(String paterno) throws Exception;

}

```

- La clase **EmpleadoDAO** implementa la interfaz **IEmpleadoDAO**, su script es el siguiente:

```

package dao.component;

import dao.design.IEmpleadoDAO;
import dao.ds.AccesoDB;
import dao.to.EmpleadoTO;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;

/**
 *
 * @author Gustavo Coronel
 */
public class EmpleadoDAO implements IEmpleadoDAO {

    public ArrayList<EmpleadoTO> consultarPorSucursal(String sucursal) throws Exception {
        String query = "select * from empleado where chr_emplcodigo " +
                      "in (select chr_emplcodigo from asignado where chr_sucodigo = ?)";
        PreparedStatement ps = AccesoDB.getConnection().prepareStatement(query);
        ps.setString(1, sucursal);
        ResultSet rs = ps.executeQuery();
        ArrayList<EmpleadoTO> lista = new ArrayList<EmpleadoTO>();
        EmpleadoTO empl = null;
        while (rs.next()) {
            empl = new EmpleadoTO();
            empl.setCodigo(rs.getString("chr_emplcodigo"));
            empl.setPaterno(rs.getString("vch_emplpaterno"));
            empl.setMaterno(rs.getString("vch_emplmaterno"));
            empl.setNombre(rs.getString("vch_emplnombre"));
            empl.setCiudad(rs.getString("vch_emplciudad"));
            empl.setDireccion(rs.getString("vch_empldireccion"));
        }
        return lista;
    }
}

```

```
        empl.setUsuario(rs.getString("vch_emplusuario"));
        empl.setClave(rs.getString("vch_emplclave"));
        lista.add(empl);
    }
    rs.close();
    ps.close();
    return lista;
}

public EmpleadoTO consultarPorUsuario(String usuario) throws Exception {
    String sql = "select * from empleado where vch_emplusuario = ?";
    PreparedStatement ps = AccesoDB.getConnection().prepareStatement(sql);
    ps.setString(1, usuario);
    ResultSet rs = ps.executeQuery();
    EmpleadoTO empleadoTO = null;
    if (rs.next()) {
        empleadoTO = new EmpleadoTO();
        empleadoTO.setCodigo(rs.getString("chr.emplicodigo"));
        empleadoTO.setPaterno(rs.getString("vch.emplpaterno"));
        empleadoTO.setMaterno(rs.getString("vch.emplmaterno"));
        empleadoTO.setNombre(rs.getString("vch.emplnombre"));
        empleadoTO.setCiudad(rs.getString("vch.emplciudad"));
        empleadoTO.setDireccion(rs.getString("vch.empldireccion"));
        empleadoTO.setUsuario(rs.getString("vch.emplusuario"));
        empleadoTO.setClave(rs.getString("vch.emplclave"));
    }
    return empleadoTO;
}

public ArrayList<EmpleadoTO> consultarPorPaterno(String paterno) throws Exception {
    ArrayList<EmpleadoTO> lista = new ArrayList<EmpleadoTO>();
    String sql = "select * from empleado where vch_emplpaterno like ?";
    PreparedStatement ps = AccesoDB.getConnection().prepareStatement(sql);
    paterno = "%" + paterno + "%";
    ps.setString(1, paterno);
    ResultSet rs = ps.executeQuery();
    EmpleadoTO empleadoTO = null;
    while (rs.next()) {
        empleadoTO = new EmpleadoTO();
        empleadoTO.setCodigo(rs.getString("chr.emplicodigo"));
        empleadoTO.setPaterno(rs.getString("vch.emplpaterno"));
        empleadoTO.setMaterno(rs.getString("vch.emplmaterno"));
        empleadoTO.setNombre(rs.getString("vch.emplnombre"));
        empleadoTO.setCiudad(rs.getString("vch.emplciudad"));
        empleadoTO.setDireccion(rs.getString("vch.empldireccion"));
        empleadoTO.setUsuario(rs.getString("vch.emplusuario"));
        empleadoTO.setClave(rs.getString("vch.emplclave"));
        lista.add(empleadoTO);
    }
}
```

```

        return lista;
    }

}

```

- La interfaz **ISucursalDAO**, define los métodos que debe implementar la clase SucursalDAO, su script es el siguiente:

```

package dao.design;

import dao.to.SucursalTO;

/**
 *
 * @author Gustavo Coronel
 */
public interface ISucursalDAO {

    public abstract SucursalTO consultaPorEmpleado( String codigo ) throws Exception;

}

```

- La clase **SucursalDAO** implementa la interfaz ISucursalDAO, su script es el siguiente:

```

package dao.component;

import dao.design.ISucursalDAO;
import dao.ds.AccesoDB;
import dao.to.SucursalTO;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

/**
 *
 * @author Gustavo Coronel
 */
public class SucursalDAO implements ISucursalDAO {

    public SucursalTO consultaPorEmpleado(String codigo) throws Exception {
        String query = "select * from sucursal " +
                "where chr_sucodigo in " +
                "(select chr_sucodigo from asignado " +
                "where chr_emplcodigo = ? " +
                "and dtt_asigfechabaja is null)";

        PreparedStatement ps = AccesoDB.getConnection().prepareStatement(query);
        ps.setString(1, codigo);
        ResultSet rs = ps.executeQuery();
        SucursalTO sucursalTO = null;
        if (rs.next()) {
            sucursalTO = new SucursalTO();

```

```

        sucursalTO.setCodigo(rs.getString("chr_sucucodigo"));
        sucursalTO.setNombre(rs.getString("vch_sucunombre"));
        sucursalTO.setCiudad(rs.getString("vch_sucuciudad"));
        sucursalTO.setDireccion(rs.getString("vch_sucudireccion"));
        sucursalTO.setContcuenta(rs.getString("int_sucucontcuenta"));
    }

    return sucursalTO;
}

}

```

Parte 2 Implementación del Requerimiento

Diseño y Programación de Acceso al Sistema

La Figura 18.8 muestra el diagrama de clases de acceso al sistema.

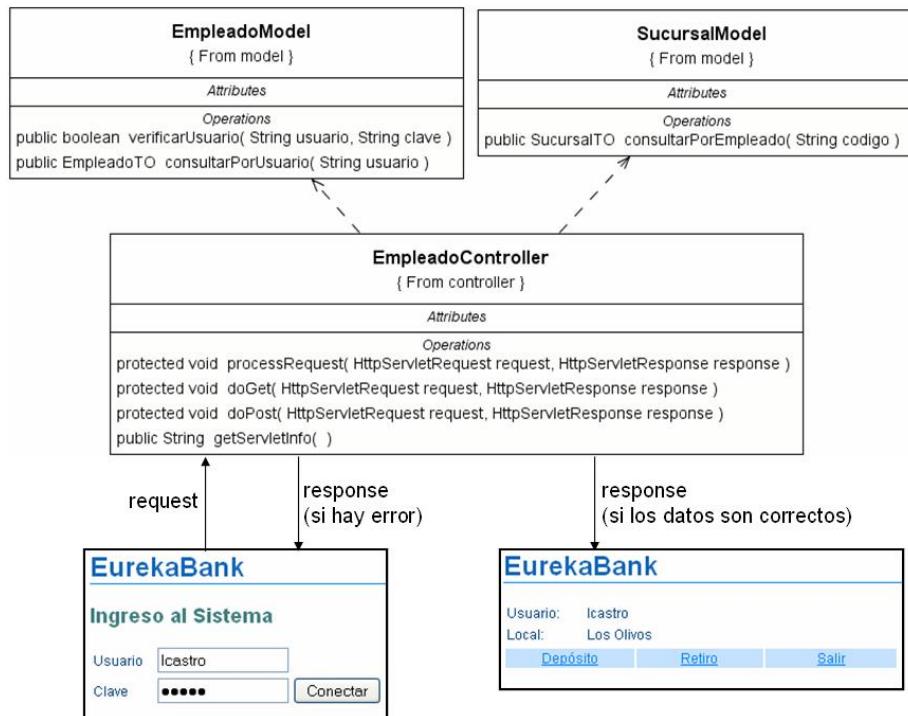


Figura 18 . 8 Diagrama de clases de proceso de acceso al sistema.

- La clase **EmpleadoModel** tiene dos métodos que describo a continuación:
 - **consultarPorUsuario()**, este método permite consultar los datos de un empleado por su nombre de usuario, retorna una instancia del objeto **EmpleadoTO**.
 - **verificarUsuario()**, este método se encarga de validar el nombre de usuario y su respectiva clase de un empleado.
- La clase **SucursalModel** tiene el método **consultarPorEmpleado()** que permite consultar los datos de la sucursal donde el empleado está laborando, retorna una instancia de la clase **SucursalTO**.

- El servlet **EmpleadoController** recibe el requerimiento de validación de la página **login.jsp**, si los datos no son correctos envía un mensaje de error a la misma página, y de ser correctos hace un forward a la página **main.jsp**.
- La página **login.jsp** muestra el formulario que permite el ingreso al sistema, los empleados deben ingresar su nombre de usuario y su clave, luego deben hacer clic en el botón **Conectar**.
- La página **main.jsp** es la página principal del sistema, presenta el menú de la aplicación el cual cuenta con las siguientes opciones:
 - **Depósito**, es un enlace que nos lleva a la página **deposito.jsp**.
 - **Retiro**, es un enlace que nos lleva a la página **retiro.jsp**.
 - **Salir**, es un enlace que finaliza la sesión del empleado y luego hace un forward a la página **login.jsp**.

A continuación tenemos el script de la clase **EmpleadoModel**:

```
package model;

import dao.DAOFactory;
import dao.design.IEmpleadoDAO;
import dao.to.EmpleadoTO;

/**
 *
 * @author gustavo
 */
public class EmpleadoModel {

    public boolean verificarUsuario(String usuario, String clave) throws Exception {
        IEmpleadoDAO empleadoDAO = DAOFactory.getInstance().getEmpleadoDAO();
        EmpleadoTO empleadoTO = empleadoDAO.consultarPorUsuario(usuario);
        if (empleadoTO == null) {
            return false;
        }
        boolean rpta = false;
        if (empleadoTO.getClave().equals(clave)) {
            rpta = true;
        }
        return rpta;
    }

    public EmpleadoTO consultarPorUsuario(String usuario) throws Exception {
        IEmpleadoDAO empleadoDAO = DAOFactory.getInstance().getEmpleadoDAO();
        EmpleadoTO empleadoTO = empleadoDAO.consultarPorUsuario(usuario);
        return empleadoTO;
    }
}
```

A continuación tenemos el script de la clase **SucursalModel**:

```
package model;

import dao.DAOFactory;
import dao.design.ISucursalDAO;
import dao.to.SucursalTO;

/**
 *
 * @author Gustavo Coronel
 */
public class SucursalModel {

    public SucursalTO consultarPorEmpleado(String codigo) throws Exception {
        ISucursalDAO sucursalDAO = DAOFactory.getInstance().getSucursalDAO();
        SucursalTO sucursalTO = sucursalDAO.consultaPorEmpleado(codigo);
        return sucursalTO;
    }
}
```

A continuación tenemos la codificación del controlador, el servlet **EmpleadoController**:

```
package controller;

import dao.to.EmpleadoTO;
import dao.to.SucursalTO;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import model.EmpleadoModel;
import model.SucursalModel;

/**
 *
 * @author Gustavo Coronel
 */
public class EmpleadoController extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        String destino = "/view/main.jsp";
        try {
            String usuario = request.getParameter("usuario");

```

```

String clave = request.getParameter("clave");
EmpleadoModel empleadoModel = new EmpleadoModel();
boolean rpta = empleadoModel.verificarUsuario(usuario, clave);
if (rpta) {
    EmpleadoTO empleadoTO = empleadoModel.consultarPorUsuario(usuario);
    SucursalModel sucursalModel = new SucursalModel();
    SucursalTO sucursalTO = sucursalModel.consultarPorEmpleado(empleadoTO.getCodigo());
    session.setAttribute("empleadoTO", empleadoTO);
    session.setAttribute("sucursalTO", sucursalTO);
} else{
    request.setAttribute("mensaje", "Datos no son correctos.");
    destino = "/view/login.jsp";
}
} catch (Exception e) {
    request.setAttribute("error", e.getMessage());
    destino = "/view/login.jsp";
}
RequestDispatcher rd = request.getRequestDispatcher(destino);
rd.forward(request, response);
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}

public String getServletInfo() {
return "Short description";
}// </editor-fold>

}

```

A continuación tenemos el script de la página **login.jsp**:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<base href="<%=" request.getRequestURL().toString()%>" />
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link rel="stylesheet" type="text/css" href="../css/EurekaBank.css"/>
<title>JSP Page</title>

```

```

</head>
<body>
    <h1>EurekaBank</h1>
    <h2>Ingreso al Sistema</h2>
    <form name="form1" method="post" action="

```

A continuación tenemos el script de la página **main.jsp**:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <base href="<%= request.getRequestURL().toString()%>" />
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <link rel="stylesheet" type="text/css" href="../css/EurekaBank.css"/>
        <title>JSP Page</title>
    </head>
    <body>
        <h1>EurekaBank</h1>
        <table width="500">
            <tr>

```

```

<td width="62">Usuario:</td>
<td width="178">${sessionScope.empleadoTO.usuario}</td>
<td width="47">Local:</td>
<td width="193">${sessionScope.sucursalTO.nombre}</td>
</tr>
</table>
<table width="521">
<tr class="menu01">
<td width="143"><a href="

```

Diseño y Programación de los Procesos del Sistema

La Figura 18.9 muestra el diagrama de clases de los procesos de depósito y retiro, para que el controlador pueda identificar el tipo de proceso se utiliza un control oculto de nombre **op**, este control tomará el valor **01** para depósito y el valor **02** para retiro.

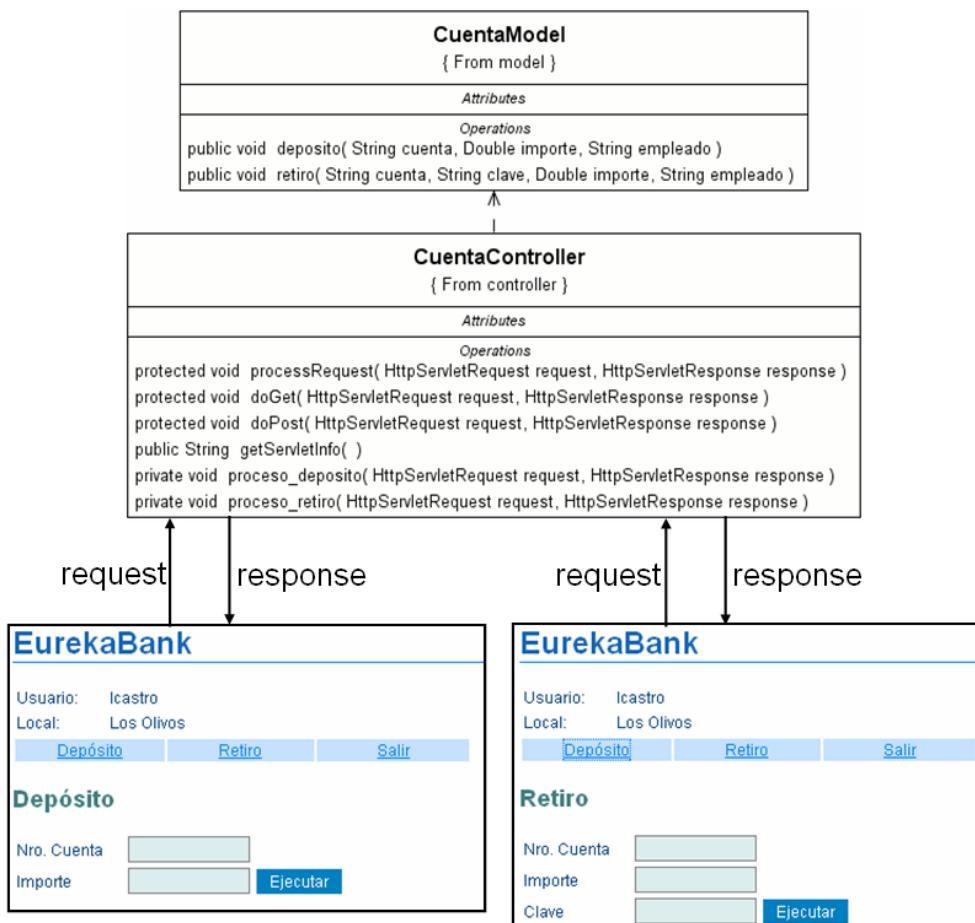


Figura 18 . 9 Diagrama de clases de los procesos de depósito y retiro.

El script del controlador, en este caso es el servlet **CuentaController** se implementa con el siguiente script:

```
package controller;

import dao.to.EmpleadoTO;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import model.CuentaModel;

/**
 *
 * @author Gustavo Coronel
 */
public class CuentaController extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String op = request.getParameter("op");
        if (op.equals("01")) {
            proceso_deposito(request, response);
        } else if (op.equals("02")) {
            proceso_retiro(request, response);
        } else {
            RequestDispatcher rd = request.getRequestDispatcher("/view/main.jsp");
            rd.forward(request, response);
        }
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>

    private void proceso_deposito(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        try {

```

```

// Datos
String cuenta = request.getParameter("cuenta");
Double importe = Double.parseDouble(request.getParameter("importe"));
HttpSession session = request.getSession(true);
EmpleadoTO empleadoTO = (EmpleadoTO) session.getAttribute("empleadoTO");
String empleado = empleadoTO.getCodigo();
// Proceso
CuentaModel cuentaModel = new CuentaModel();
cuentaModel.deposito(cuenta, importe, empleado);
request.setAttribute("mensaje", "Proceso Ok.");
} catch (Exception e) {
    request.setAttribute("error", e.getMessage());
}
RequestDispatcher rd = request.getRequestDispatcher("./view/deposito.jsp");
rd.forward(request, response);
}

private void proceso_retiro(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
try {
    // Datos
    String cuenta = request.getParameter("cuenta");
    Double importe = Double.parseDouble(request.getParameter("importe"));
    String clave = request.getParameter("clave");
    HttpSession session = request.getSession(true);
    EmpleadoTO empleadoTO = (EmpleadoTO) session.getAttribute("empleadoTO");
    String empleado = empleadoTO.getCodigo();
    // Proceso
    CuentaModel cuentaModel = new CuentaModel();
    cuentaModel.retiro(cuenta, clave, importe, empleado);
    request.setAttribute("mensaje", "Proceso Ok.");
} catch (Exception e) {
    request.setAttribute("error", e.getMessage());
}
RequestDispatcher rd = request.getRequestDispatcher("./view/retiro.jsp");
rd.forward(request, response);
}
}

```

La implementación de la página **deposito.jsp** se realiza mediante el siguiente script:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

```

```

<html>
<head>
<base href="<%=" request.getRequestURL().toString()%>" />

```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<link rel="stylesheet" type="text/css" href="../css/EurekaBank.css"/>
<title>JSP Page</title>
</head>
<body>
<h1>EurekaBank</h1>
<table width="318">
<tr>
<td width="67">Usuario:</td>
<td width="239">${sessionScope.empleadoTO.usuario}</td>
</tr>
<tr>
<td>Local:</td>
<td>${sessionScope.sucursalTO.nombre}</td>
</tr>
</table>
<table width="350">
<tr class="menu01">
<td width="111">
<a href="
```

```

        size="10" maxlength="10">
    </td>
    <td width="117">
        <input name="ejecutar" type="submit" class="button" id="ejecutar" value="Ejecutar">
    </td>
</tr>
</table>
</form>
<c:if test="#{requestScope['mensaje'] != null}">
    <div id="mensaje">
        <h2>Mensaje</h2>
        <p>${requestScope["mensaje"]}</p>
    </div>
</c:if>
<c:if test="#{requestScope['error'] != null}">
    <div id="Error">
        <h2>Error</h2>
        <p class="error">${requestScope["error"]}</p>
    </div>
</c:if>
</body>
</html>

```

La implementación de la página retiro.jsp se realiza mediante el siguiente script:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <base href="<%=" request.getRequestURL().toString()%>" />
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <link rel="stylesheet" type="text/css" href="../css/EurekaBank.css"/>
        <title>JSP Page</title>
    </head>
    <body>
        <h1>EurekaBank</h1>
        <table width="318">
            <tr>
                <td width="67">Usuario:</td>
                <td width="239">${sessionScope.empleadoTO.usuario}</td>
            </tr>
            <tr>
                <td>Local:</td>
                <td>${sessionScope.sucursalTO.nombre}</td>
            </tr>
        </table>
        <table width="350">

```

```
<tr class="menu01">
    <td width="111">
        <a href="
```

```

        </div>
    </c:if>
    <c:if test="#{requestScope['error'] != null}">
        <div id="Error">
            <h2>Error</h2>
            <p class="error">${requestScope["error"]}</p>
        </div>
    </c:if>
</body>
</html>

```

Finalmente, para cerrar la sesión se implementa el servlet **FinalizarSession**, su script es el siguiente:

```

package controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 *
 * @author Gustavo Coronel
 */
public class FinalizarSession extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        session.invalidate();
        RequestDispatcher rd = request.getRequestDispatcher("/index.jsp");
        rd.forward(request, response);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    public String getServletInfo() {

```

```
    return "Short description";  
} // </editor-fold>  
  
}
```

Difinitivamente, la programación en capas y utilizando patrones de diseño hacen mas ordenado el desarrollo, los tiempos para entregar los productos se acortan y mejor aún si se cuenta con un buen equipo de desarrolladores, y lo mas importante, el mantenimiento se hace mas fácil.

Por ejemplo si queremos cambiar la cara de la aplicación solo tenemos que centrarnos en la vista, osea, en los JSP, por que toda la lógica ya esta funcionando.

Si queremos hacer cambios en los procesos de fondo, tendríamos que centrarnos en el DAO, por ejemplo, si revisamos el Capítulo 8, podemos ver las reglas de negocio, tomelo como un reto implementarlas correctamente en este proyecto.

Enlaces

- **<http://java.sun.com/>**
Sitio oficial de Java.
- **<http://books.coreservlets.com/>**
Sitio dedicado a Java, aquí encontraras libros en formato PDF.
- **<http://www.kickjava.com/freeBooks.html>**
Interesante sitio donde encontrará gran cantidad de manuales referentes a Java.
- **<http://www.javahispano.org/>**
Sitio dedicado a Java, sobre todo en español.
- **<http://www.theserverside.com/tss>**
Sitio dedicado a Java.
- **<http://javahispano.net/>**
Interesante comunidad desde donde podrá bajar código libremente.
- **<http://www.elistas.net/lista/todojava>**
Lista de interés para todo lo relacionado a Java, muy interesante.
- **<http://java.sun.com/products/jsp/docs.html>**
Documentación Oficial sobre JSP.
- **<http://java.sun.com/products/jsp/tags/11/tags11.html>**
Referencia sobre la sintaxis de JSP.
- **<http://java.sun.com/javaee/reference/index.jsp>**
Documentación Oficial sobre Java EE
- **<http://www.jugperu.com/portal/>**
Comunidad peruana de Usuarios de Java.
- **<http://javabat.com/>**
Sitio con ejercicios de Java para practicar online.
- **<http://javaweb.osmosislatina.com/>**
Interesando Sitio dedicado a Java.
- **<http://gcoronelc.blogspot.com>**
Blog de Gustavo Coronel

Bibliografía

- **The Java EE 5Tutorial**
Autor: SunMicrosystems, Inc.
Editorial: SunMicrosystems, Inc.
- **Core Servlets and JavaServer Pages**
Autor: Marty Hall
Editorial: Microsystem Press
- **La Biblia de Java 2 v5.0**
Autor: Herbert Schildt
Editorial: ANAYA
- **Piensa en Java**
Autor: Bruce Eckel
Editorial: PEARSON EDUCACIÓN
- **Desarrollo Web con JSP**
Autores: Jayson Falkner, Ben Galbraith, et al.
Editorial: ANAYA
- **Beginning JavaServer Pages**
Autores: Vivek Chopra, Sing Li, Rupert Jones, Jon Eaves y John T. Bell
Editorial: Wiley Publishing, Inc.
- **Advanced JavaServer Pages**
Autor: David M. Geary
Editorial: Prentice Hall PTR
- **Professional JSP**
Autor: Karl Avedal, Danny Ayers, Timothy Briggs, et al.
Editorial: Wrox
- **Java Servlet Programming**
Autor: Jason Hunter, William Crawford
Editorial: O'Reilly & Associates, Inc.
- **Aprenda Servlets de Java como si estubiera en Primero**
Autores: Javier García de Jalón, José Ignacio Rodríguez y Aitor Imaz
Editorial: TECNUN
- **Tecnologías de Servidor con Java**
Autor: Ángel Esteban
Editorial: Grupo Eidos