

# Java y MySQL



CD-ROM MULTIMEDIA

## **Dedicatoria**

Esta obra está dedicada especialmente a mi familia.

# Acerca del Autor

---

Eric Gustavo Coronel Castillo es Ingeniero Electrónico de la Facultad de Electrónica de la Universidad Nacional de Ingeniería, desde fines de los 80's se dedica a la Informática & Sistemas, especialmente en los temas de Desarrollo y Base de Datos, en la actualidad es uno de los profesionales mas reconocido de nuestro medio debido a su amplia experiencia.

Como Desarrollador y Consultor de Sistemas ha brindado sus servicios a instituciones públicas y privadas como la Universidad Nacional de Ingeniería, Policía Nacional del Perú, Prodeco Asociados S.A., Constructora Racarrumy SA, Infecom EIRL, Casas & Cosas S.A., GrapPeru SAC, AlfaVia, Instituto Peruano del Deporte y a muchas pequeñas y microempresas.

Como Docente cuenta con más de 15 años de experiencia en el dictado de cursos de desarrollo orientados al procesamiento de datos, en la elaboración de manuales técnicos, y en la difusión de tecnologías emergentes. Actualmente es docente en la Facultad de Ingeniería de Sistemas de la Universidad Nacional de Ingeniería en su unidad de capacitación – SistemasUNI y en el Instituto Superior San Ignacio de Loyola. También ha prestado sus servicios como docente a la Universidad “Alas Peruanas”, Universidad Particular “Antenor Orrego”, Universidad Nacional de Trujillo, SENCICO, GrapPerú, programa PECl del Instituto San Ignacio de Loyola, Universidad Federico Villarreal, y otros.

Como Conferencista es invitado periódicamente a exponer seminarios de difusión tecnológica en las principales ciudades del país, como la Universidad Nacional Mayor de San Marcos, Universidad Nacional de Ingeniería, Universidad Inca Gracilazo de la Vega, Universidad Nacional de Trujillo, Universidad Nacional "San Antonio Abad" del Cuzco, Universidad “Los Angeles” de Chimbote, GrapPeru, y otros.

En Octubre del 2000 sale a luz su primera publicación, “Diseño de Aplicaciones Cliente-Servidor con Visual Basic 6.0 & SQL Server 7.0” con el cual se hace conocido tanto a nivel local y nacional; en Enero del 2001 publica su segundo libro “Diseño de Aplicaciones Cliente-Servidor con Power Builder 7.0 y SQL Server 7.0”, con el que recibe el reconocimiento de varios países tales como Bolivia, Ecuador, Colombia, y otros.

En el 2005 es invitado por la **Empresa Editora MACRO**, y en Septiembre del mismo año sale a luz su libro titulado “**Desarrollo de Aplicaciones con PHP y MySQL**”, luego en Marzo del 2006 publica el libro “**Lenguaje de Programación Java2 Versión 5**”, en la actualidad forma parte del staff de escritores de esta editorial.

# Agradecimientos

---

Hacer esta obra me ha resultado muy interesante, me ha permitido conocer mucho más a Java y toda su filosofía de programación, pero sobre todo ingresar al mundo de la aplicación de los **Patrones de Diseño**, pienso que todo profesional que se dedica a crear soluciones con Java debe no sólo conocer, sino también aplicar correctamente los patrones de diseño, ya que son soluciones simples y elegantes, y sobre todo probadas sobre problemas comunes de la programación.

En mi labor como docente, empecé a desarrollar patrones de diseño como DAO y MVC, la pregunta natural de mis alumnos con respecto a estos temas siempre es: **¿Qué bibliografía me recomienda?**, por supuesto que hay mucha, pero existe el inconveniente primero del lenguaje, ya que la mayoría está en inglés, y si está en español el problema está en los ejemplos, son insuficientes o no son muy claros, espero que mi libro ayude a entender los conceptos y su aplicación en soluciones empresariales.

El resultado de un libro de este tipo es sin duda una tarea muy difícil, pero mi labor como docente me facilita dar a conocer mis ideas y recibir críticas, ya que es en las aulas donde primero las expongo y recibo la opinión de mis alumnos. Otra fuente muy importante son los comentarios de personas que han leído mis libros anteriores, ellos también han contribuido a enriquecer esta obra.

Por eso, esta obra es también de todas las personas que han participado desde el primer momento que empecé a diagramarla, son muchas las personas con las que converse y de las que recibí sus comentarios, pero debo agradecer de manera especial a mis amigos y colegas por sus aportes y tiempo dedicado:

- **Sergio Matsukawa Maeda:** Como siempre, Sergio está dispuesto a colaborar en mis publicaciones, sus aportes son muy precisos.
- **Ricardo Marcelo Villalobos:** Ricardo es el crítico mas agudo de mis libros, sus aportes sirvieron para hacer varios cambios, tanto en el enfoque, desarrollo de los temas, como en los ejemplos. Aprovecho para felicitar a Ricardo por sus 4 libros de "Fundamentos de Programación", con Java, PHP, C++ y Visual Basic.
- **Julio Flores Manco:** Julio como siempre bastante colaborador, sus sugerencias fueron bastante valiosas.
- **Hugo Valencia Morales:** Los aportes de Hugo siempre son bienvenidos, también es muy preciso en sus comentarios. Espero que se anime a escribir algún libro, sin duda sería un aporte muy interesante.
- **Jorge Guerra Guerra:** Jorge es una persona que sabe mucho de Java, sus aportes son bastante interesantes y precisos, sigo esperando su libro de Java.
- **Franciso Javier Wong Cabanillas:** Aunque mi amigo Javier no se dedica al área de las TI, su aporte esta en el área pedagógica, él es Doctor en Educación, y siempre estamos conversando de como abordar estos temas, después de cada tertulia terminamos renovados y con nuevos objetivos en este delicado campo de la educación.

Un agradecimiento muy especial a mi familia por apoyarme en todo momento, su respaldo y motivación fueron un pilar muy importante para poder concluir esta obra.

A usted amigo lector, espero me haga llegar sus comentarios, sugerencias, etc. a mi correo electrónico gcoronel@uni.edu.pe, ya que todos ellos servirán para mejorar mis siguientes publicaciones.

Finalmente, reiterando mi compromiso de seguir colaborando con el desarrollo de la computación e informática con mis publicaciones.

*Eric Gustavo Coronel Castillo*

*gcoronelc@gmail.com*

# 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 ventana 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**

### **Capitulo 01: Introducción al Desarrollo de Software**

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

### **Capitulo 02: Software a Utilizar**

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

### **Capitulo 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 temas 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 multitas 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.

**Eric Gustavo Coronel Castillo**  
**gcoronelc@gmail.com**  
**Lima – Perú, Febrero del 2009**

# 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.

Continuando con mis libros en el tema de Java este es el segundo, en una siguiente publicación abordando el tema de Patrones de Diseño con mas profundidad y los Frameworks de de desarrollo como Java Server Faces y Spring.

## **Parte 01: Programación Orientada a Objetos**

### **Capitulo 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

### **Capitulo 02: Software a Utilizar**

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

### **Capitulo 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 (navigability )
    - 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
    - 13.2.1.2. 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

## **Capitulo 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

## **Capitulo 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**

### **Capitulo 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

## **Capitulo 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

# Capítulo 5

## 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 imponen 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

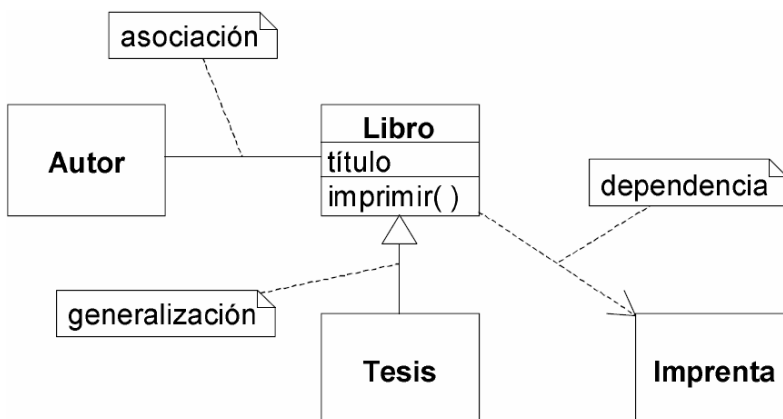
Las relaciones existentes entre las distintas clases de un sistema nos indican cómo se comunican los objetos de estas clases entre sí.

Los mensajes "navegan" por las relaciones existen 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 esta representada por una línea continua terminada en un triangulo blanco, y la asociación es representada por una línea continua simple.



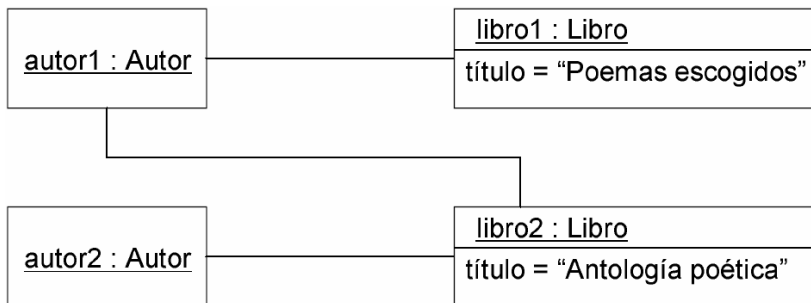
**Figura 5 . 1** Diagrama de clases con sus tres relaciones básicas.

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.

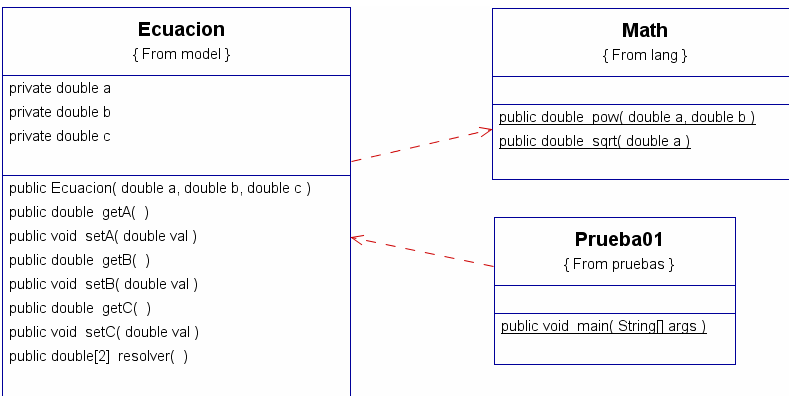
v Con la dependencia mostramos que un cambio en la clase independiente puede afectar al funcionamiento de la clase dependiente, pero no al contrario.

### 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 *Ecuacion* depende de la clase *Math* por que 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 *Ecuacion* es la clase dependiente. También existe una dependencia de la clase *Prueba01* y la clase *Ecuacion*, para este caso la clase *Ecuacion* 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 *Ecuacion* 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 );

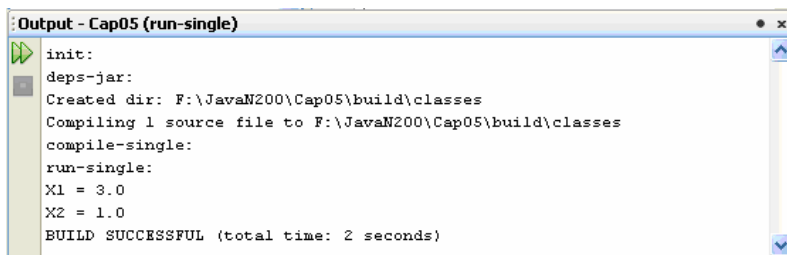
        // Se obtienen las dos raíces
        double rpta[] = obj.resolver();

        // 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.



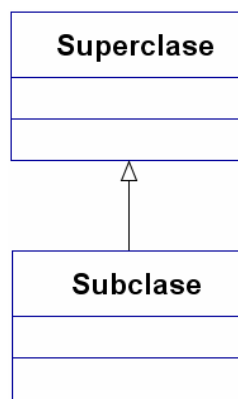
**Figura 5.4** Ejecución de la clase *Prueba01*.

## 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.



**Figura 5 . 5** Notación UML para representar la herencia.

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

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 grafico que se muestra en la Figura 5.5.

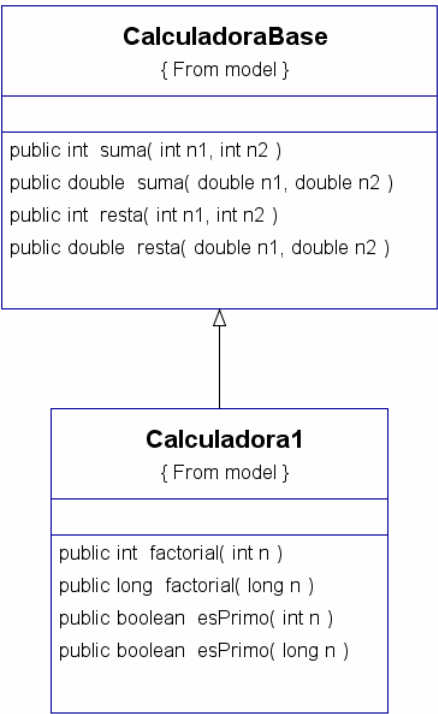
```
public class Superclase {  
  
}  
  
public class Subclase extends Superclase {  
  
}
```

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`.



**Figura 5 . 6** Representación UML de Herencia entre Clases.

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

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 esta 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;
}

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);

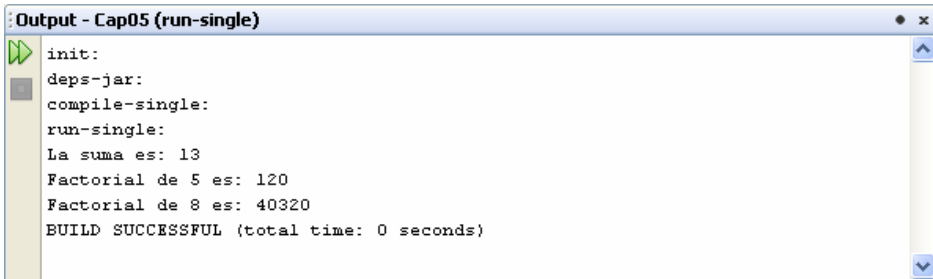
        // Reporte
        System.out.println("La suma es: " + suma);
        System.out.println("Factorial de " + n1 + " es: " + f1);
        System.out.println("Factorial de " + n2 + " es: " + f2);

    }

} // 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*.



```
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.

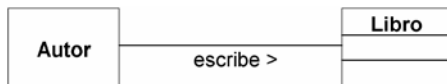
### 5.4.1. Asociación Binaria

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 si 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.

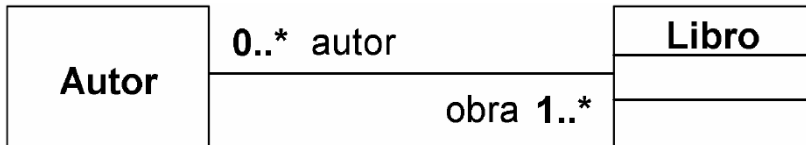
El nombre de la asociación puede contener un pequeño triángulo negro, denominado “dirección del nombre” (*name direction*), que apunta en la dirección en la que se debe leer la asociación, tal como se aprecia en la Figura 5.8.



**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. Indica 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**

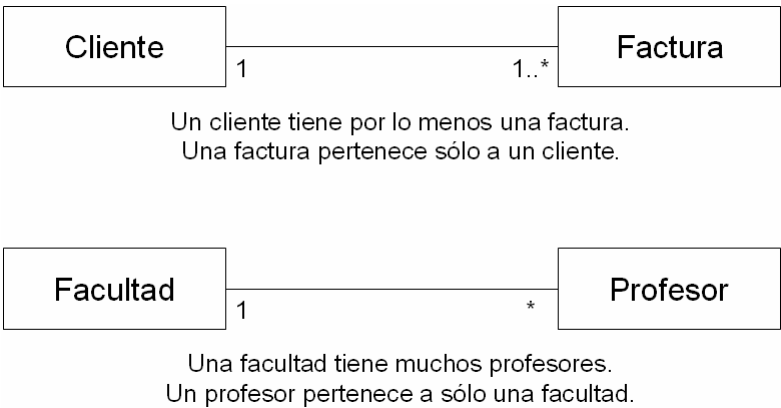
Donde *limite\_inferior* y *limite\_superior* son valores enteros literales que especifican un intervalo cerrado de enteros. Además, el asterisco (\*) se puede usar como limite superior para denotar un valor ilimitado (muchos).

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.



**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 de estas propiedades.



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

### 5.4.1.6. Navegabilidad (na vigability )

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:

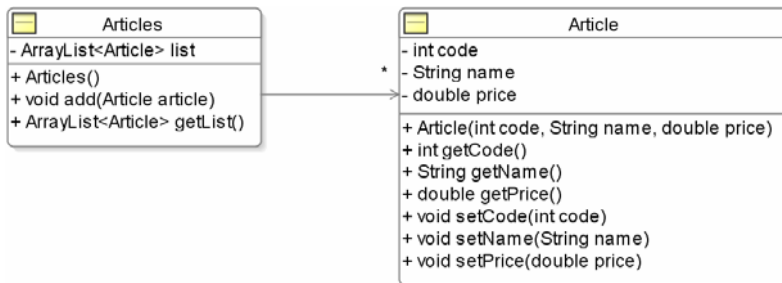
- ( + ) public
- ( ~ ) package
- ( # ) protected
- ( - ) private

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 esta 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;

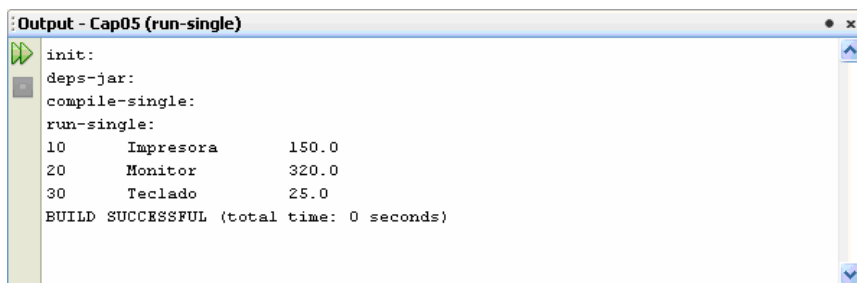
        // Le agregamos elementos
        lista.add( new Article(10,"Impresora",150.0) );
        lista.add( new Article(20,"Monitor",320.0) );
        lista.add( new Article(30,"Teclado",25.0) );

        // 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:



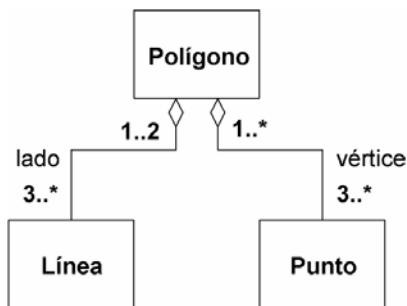
```
init:
deps-jar:
compile-single:
run-single:
10      Impresora      150.0
20      Monitor       320.0
30      Teclado        25.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

**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.

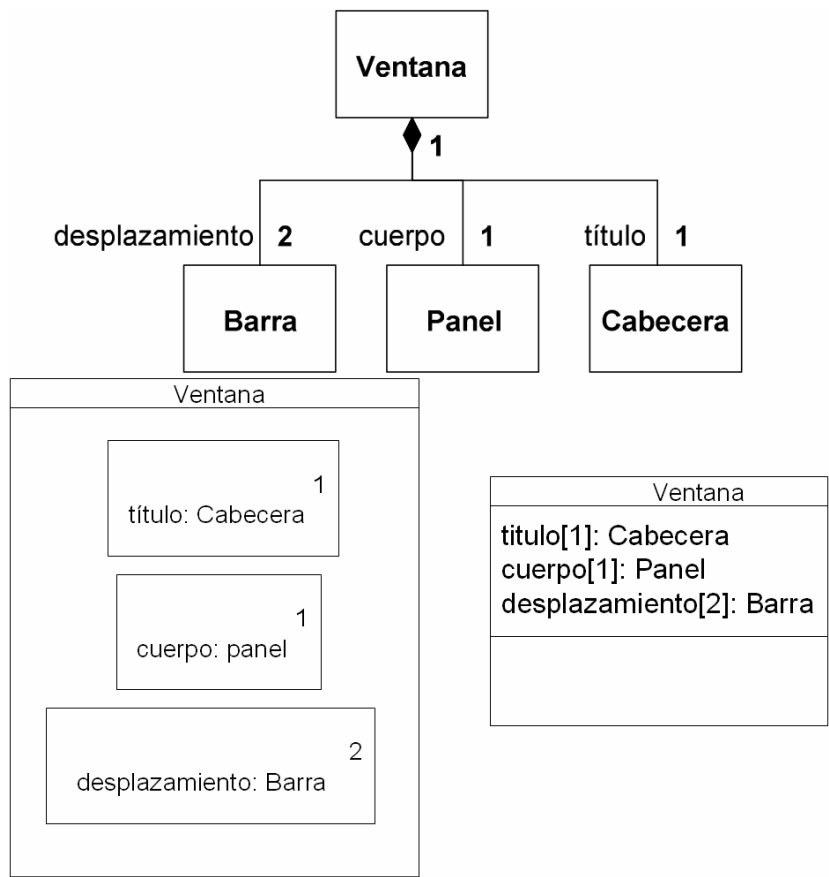
Dos o más agregaciones con el mismo “todo” se pueden dibujar en forma de árbol, juntando los extremos en uno solo, cuando coinciden las demás propiedades en el extremo agregado (multiplicidad, navegabilidad, etc.). La representación como árbol es meramente una opción de presentación, y no conlleva ninguna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

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.



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

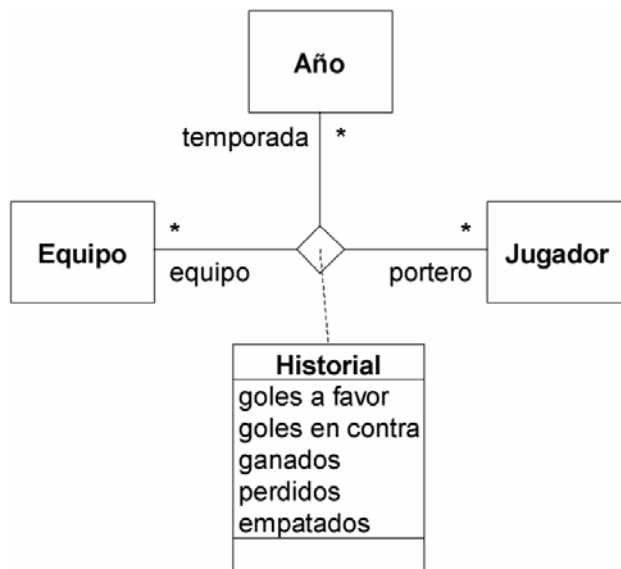
Las partes de una composición pueden ser tanto clases como asociaciones. Una asociación como parte de un compuesto significa que tanto los objetos conectados como el propio enlace pertenecen todos al mismo compuesto. En la notación convencional, será necesario representar la asociación-parte como clase-asociación, 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.



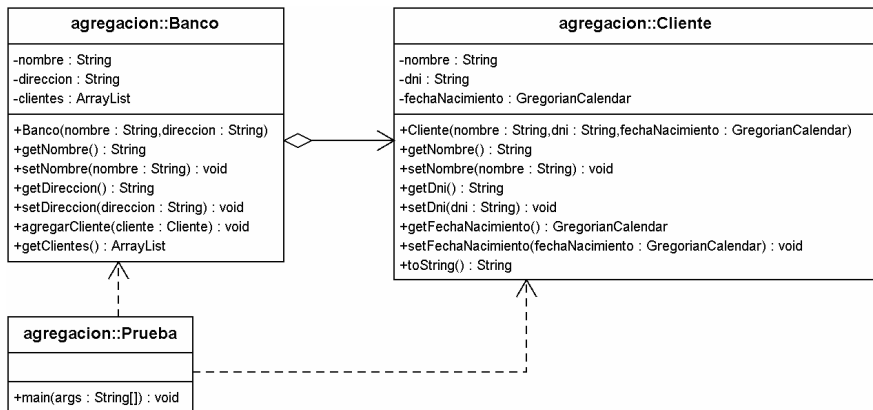
**Figura 5 . 17** Una asociación ternaria que también es clase-asociación. En este ejemplo se muestra el historial de un equipo en cada temporada con un portero concreto. Se supone que el portero puede ser intercambiado durante la temporada y por tanto puede aparecer en varios equipos

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 está conformado por cliente. Un cliente puede ser 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) {

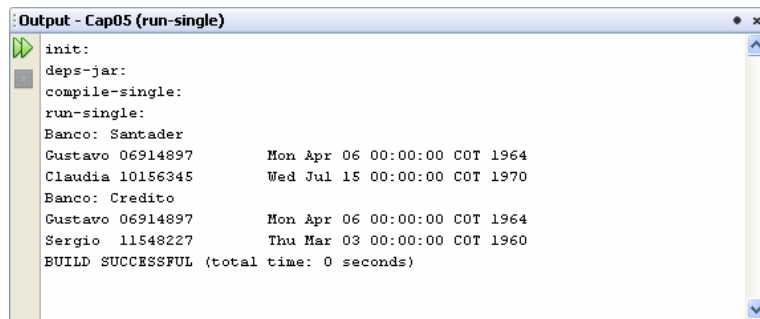
        // Creación de Objetos
        Banco banco01 = new Banco("Santader", "España");
        Banco banco02 = new Banco("Credito", "Perú");
        Cliente clie01 = new Cliente("Gustavo", "06914897", new GregorianCalendar(1964, 3, 6) );
        Cliente clie02 = new Cliente("Claudia", "10156345", new GregorianCalendar(1970,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



```
init:
deps-jar:
compile-single:
run-single:
Banco: Santander
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 quisiéramos 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.



*Página en Blanco*