

ISIL TECH

Java Enterprise Developer



Java™

Cliente / Servidor

# Contenido

<b>CAPÍTULO 1 HILOS .....</b>	<b>7</b>
INTRODUCCIÓN.....	7
EL MODELO DE HILO DE JAVA .....	7
<i>Prioridades de hilo.....</i>	8
<i>Sincronización.....</i>	8
<i>Intercambio de mensajes.....</i>	8
CLASE: Thread .....	8
INTERFACE: Runnable .....	8
PRIORIDADES DE LOS HILOS .....	10
SINCRONIZACIÓN .....	10
<i>La sentencia synchronized .....</i>	11
COMUNICACIÓN ENTRE HILOS .....	12
<i>Bloqueos.....</i>	14
RESUMEN DE LA INTERFAZ DE PROGRAMACIÓN (API) DE HILOS.....	15
<i>Métodos de clase .....</i>	15
<i>Métodos de instancia.....</i>	15
<b>CAPÍTULO 2 EXCEPCIONES.....</b>	<b>17</b>
MANEJO DE ERRORES UTILIZANDO EXCEPCIONES.....	17
¿QUÉ ES UN EXCEPCIÓN Y POR QUÉ DEBO TENER CUIDADO? .....	17
<i>Ventaja 1: Separar el Manejo de Errores del Código "Normal" .....</i>	18
<i>Ventaja 2: Propagar los Errores sobre la Pila de Llamadas.....</i>	20
<i>Ventaja 3: Agrupar Errores y Diferenciación.....</i>	22
PRIMER ENCUENTRO CON LAS EXCEPCIONES JAVA .....	24
REQUERIMIENTOS PARA CAPTURAR O ESPECIFICAR EXCEPCIONES .....	26
<i>Capturar.....</i>	26
<i>Especificar.....</i>	26
<i>Excepciones Chequeadas .....</i>	27
<i>Excepciones que pueden ser lanzadas desde el ámbito de un método .....</i>	27
TRATAR CON LAS EXCEPCIONES JAVA .....	28
<i>El ejemplo: ListOfNumbers.....</i>	28
<i>Capturar y Manejar Excepciones.....</i>	28
<i>Especificar las Excepciones que pueden ser Lanzadas por un Método .....</i>	28
EL EJEMPLO: ListOfNumbers.....	29
CAPTURAR Y MANEJAR EXCEPCIONES .....	30
<i>El Bloque try .....</i>	30
<i>Los bloques catch.....</i>	30

<i>El bloque finally</i> .....	31
<i>Poniéndolo todo junto</i> .....	31
EL BLOQUE Try .....	31
LOS BLOQUES catch .....	32
<i>Ocorre una IOException</i> .....	33
<i>Capturar Varios Tipos de Excepciones con Un Manejador</i> .....	34
EL BLOQUE finally .....	35
<i>¿Es realmente necesaria la sentencia finally?</i> .....	36
PONIÉNDOLO TODO JUNTO .....	37
<i>Escenario 1: Ocorre una excepción IOException</i> .....	37
<i>Escenario 2: Ocorre una excepción ArrayIndexOutOfBoundsException</i> .....	39
<i>Escenario 3: El bloque try sale normalmente</i> .....	39
<i>Especificar las Excepciones Lanzadas por un Método</i> .....	40
LA SENTENCIAS throw .....	40
<i>La clausula throws</i> .....	41
LA CLASE Throwable Y SUS SUBCLASES .....	42
<i>Error</i> .....	42
<i>Exception</i> .....	42
<i>Excepciones en Tiempo de Ejecución</i> .....	43
CREAR CLASES DE EXCEPCIONES .....	43
<i>¿Qué puede ir mal?</i> .....	44
<i>Elegir el Tipo de Excepción Lanzada</i> .....	44
<i>Elegir una Superclase</i> .....	46
<i>Convenciones de Nombres</i> .....	46
EXCEPCIONES EN TIEMPO DE EJECUCIÓN - LA CONTROVERSIA .....	46
<i>Reglas del Pulgar:</i> .....	48
<b>CAPÍTULO 3 REMOTE METHOD INVOCATION .....</b>	<b>49</b>
INTRODUCCIÓN .....	49
CREAR UN SERVIDOR RMI .....	50
<i>Definir el interfaz remoto</i> .....	50
<i>Implementar el interfaz remoto</i> .....	51
<i>Compilar y ejecutar el servidor</i> .....	52
CREAR UN CLIENTE RMI .....	53
<i>Definir la clase para obtener los objetos remotos necesarios</i> .....	53
<i>Compilar y ejecutar el cliente</i> .....	54
RESUMEN .....	55
<b>CAPÍTULO 4 SOCKETS .....</b>	<b>56</b>
FUNDAMENTOS .....	56
FUNCIONAMIENTO GENÉRICO .....	57

JAVA SOCKETS .....	58
<i>Introducción</i> .....	58
<i>Modelo de comunicaciones con Java</i> .....	59
<i>Apertura de Sockets</i> .....	59
<i>Creación de Streams</i> .....	60
Creación de Streams de Entrada .....	60
Creación de Streams de Salida .....	61
<i>Cierre de Sockets</i> .....	62
<i>Clases útiles en comunicaciones</i> .....	63
<i>Ejemplo de uso</i> .....	64
Programa Cliente .....	64
Programa Servidor .....	65
Ejecución .....	66
<b>CAPÍTULO 5 JAVA DATADASE CONNECTIVITY.....</b>	<b>68</b>
INTRODUCCIÓN .....	68
EMPEZAR CON JDBC .....	68
SELECCIONAR UNA BASE DE DATOS .....	69
ESTABLECER UNA CONEXIÓN .....	69
<i>Cargar los Drivers</i> .....	70
<i>Hacer la Conexión</i> .....	70
SELECCIONAR UNA TABLA .....	71
<i>Crear sentencias JDBC</i> .....	74
<i>Ejecutar Sentencias</i> .....	75
<i>Introducir Datos en una Tabla</i> .....	75
<i>Obtener Datos desde una Tabla</i> .....	76
RECUPERAR VALORES DESDE UNA HOJA DE RESULTADOS .....	78
<i>Utilizar el Método next</i> .....	78
<i>Utilizar los métodos getXXX</i> .....	78
<i>Utilizar el método getString</i> .....	80
ACTUALIZAR TABLAS .....	80
<i>Utilizar Sentencias Preparadas</i> .....	82
<i>Cuándo utilizar un Objeto PreparedStatement</i> .....	82
<i>Crear un Objeto PreparedStatement</i> .....	83
<i>Suministrar Valores para los Parámetros de un PreparedStatement</i> .....	83
<i>Utilizar una Bucle para asignar Valores</i> .....	85
<i>Valores de retorno del método executeUpdate</i> .....	85
<i>Utilizar Uniones</i> .....	86
UTILIZAR TRANSACCIONES .....	88
<i>Desactivar el modo Auto-entrega</i> .....	88
<i>Entregar una Transacción</i> .....	88
<i>Utilizar Transacciones para Preservar al Integridad de los Datos</i> .....	89
<i>Cuándo llamar al método rollback</i> .....	90
PROCEDIMIENTOS ALMACENADOS .....	91

Utilizar Sentencias SQL .....	91
Llamar a un Procedimiento Almacenado desde JDBC .....	92
CREAR APLICACIONES JDBC COMPLETAS .....	93
Poner Código en una Definición de Clase .....	93
Importar Clases para Hacerlas Visibles .....	93
Utilizar el Método main() .....	94
Utilizar bloques try y catch .....	94
Recuperar Excepciones .....	94
Recuperar Avisos .....	96
EJECUTAR LA APLICACIÓN DE EJEMPLO .....	98
<b>CAPÍTULO 6 PATRONES DE DISEÑO EN JEE .....</b>	<b>100</b>
INTRODUCCIÓN .....	100
BREVE HISTORIA DE LOS PATRONES DE DISEÑO .....	100
PATRONES J2EE .....	102
CATÁLOGO DE PATRONES J2EE .....	103
Capa de Presentación .....	103
Capa de Negocios .....	104
Capa de Integración .....	104
PATRÓN "DATA ACCESS OBJECT" .....	105
Descripción .....	105
Ejemplo de código: los bean (Transfer Object) .....	107
Ejemplo de código: los DAOs .....	110
Ejemplo de código: la factoria de DAOs .....	112
Ejemplo de código: usando lo anterior .....	114
Nota final .....	115
DAO + FACTORY .....	117
El problema .....	117
La solución .....	118
<b>CAPÍTULO 7 IREPORT &amp; JASPERREPORT .....</b>	<b>121</b>
INTRODUCCIÓN .....	121
Requerimientos de JasperReports .....	121
FUNCIONAMIENTO DE JASPERREPORTS .....	122
Compilación, exportación de reportes de JasperReports .....	123
IREPORT .....	123
Funcionamiento de iReport .....	124
Requerimientos de instalación (Windows 2000, NT, XP) .....	124
Instalación y configuración ((Windows 2000, NT, XP)) .....	124
¿Qué necesito descargar? .....	126
Configuración de la conexión a una base de datos .....	127
Creación del Reporte .....	129
Secciones de un Reporte en iReport .....	130

---

<i>Diseño del Reporte.....</i>	<i>131</i>
<i>Compilación y Ejecución del Reporte.....</i>	<i>135</i>
CREACIÓN DE GRÁFICOS EN IREPORT .....	139
<b>CAPÍTULO 8 APACHE POI.....</b>	<b>147</b>
INTRODUCCIÓN.....	147
LEER UN ARCHIVO EXCEL .....	147
CREAR UN ARCHIVO EXCEL .....	150

# Capítulo 1

## Hilos

### INTRODUCCIÓN

---

La programación multihilo es un paradigma conceptual de la programación que divide los programas en dos o más procesos que se pueden ejecutar en paralelo. En un momento dado pueden haber datos de entrada de usuario a los que responder, animaciones y visualizaciones de interfaz de usuario, también cálculos grandes que podrían tardar varios segundos en terminar, y nuestros programas tendrán que tratar con estos temas sin provocar retrasos desagradables al usuario.

Lo interesante de todos estos procesos en paralelo es que la mayor parte de ellos realmente no necesitan los recursos completos de la computadora durante su vida operativa. El problema en los entornos de hilo único tradicionales es que se tiene que esperar a que se terminen cada una de estas tareas antes de proseguir con la siguiente. Aunque la CPU esté libre la mayor parte del tiempo, tiene que colocar las tareas en la cola ordenadamente.

### EL MODELO DE HILO DE JAVA

---

Los sistemas multihilo aprovechan la circunstancia de que la mayoría de los hilos computacionales invierten la mayor parte del tiempo esperando a que un recurso quede disponible, o bien esperando a que se cumpla alguna condición de temporización. Si fuésemos capaces de describir todas las tareas como hilos de control independientes, conmutando de manera automática entre una tarea que esté lista para pasar a un modo de espera, y otra que sí tenga algo que hacer, conseguiríamos realizar una cantidad mayor de trabajo en el mismo intervalo de tiempo.

Java se diseñó partiendo de cero, en un mundo en el que el entorno multihilo, a nivel de sistema operativo, era una realidad. El intérprete de Java hace uso intensivo de hilos para multitud de propósitos, y todas las bibliotecas de clases se diseñaron teniendo en mente el modelo multihilo. Una vez que un hilo comienza su tarea, puede suspenderse, lo que equivale a detener temporalmente su actividad. El hilo suspendido puede reanudarse, lo que supone que continúa su tarea allí donde la dejó. En cualquier momento, un hilo puede deteriorarse, finalizando su ejecución de manera inmediata. Una vez detenido, el proceso no puede reiniciarse.

## Prioridades de hilo

El intérprete de Java utiliza prioridades para determinar cómo debe comportarse cada hilo con respecto a los demás. Las prioridades de hilo son valores entre 1 y 10 que indican la prioridad relativa de un hilo con respecto a los demás.

## Sincronización

Ya que los hilos permiten y potencian el comportamiento asíncrono de los programas, debe existir alguna manera de forzar el sincronismo allí donde sea necesario. Por ejemplo, si desease que dos hilos se comunicasen para compartir una estructura de datos compleja (como una lista enlazada), necesitará alguna manera de garantizar que cada uno se aparte del camino del otro. Java incorpora una versión rebuscada de un modelo clásico para la sincronización, el monitor. La mayor parte de los sistemas multihilo implementan los monitores a modo de objetos, pero Java proporciona una solución más elegante: no existe la clase monitor, cada objeto lleva asociado su propio monitor implícito, en el que puede entrar sin más que hacer una llamada a los métodos `synchronized` del objeto. Una vez que el hilo está dentro del método `synchronized`, ningún otro hilo puede efectuar una llamada a otro método `synchronized` sobre el mismo objeto.

## Intercambio de mensajes

Una vez que el programa se ha dividido en sus partes lógicas, a modo de hilo, es preciso definir exactamente como se comunicarán entre si dichos hilos. Java utiliza los métodos `wait` y `notify` para el intercambio de información entre hilos.

## CLASE: Thread

---

En Java los hilos se representan mediante una clase. La clase `Thread` encapsula todo el control necesario sobre los hilos. Hay que tomar la precaución de distinguir claramente un objeto `Thread` de un hilo en ejecución. Un objeto `Thread` se define como el panel de control o proxy de un hilo en ejecución. En el objeto `Thread` hay métodos que controlan si el hilo se está ejecutando, está durmiendo, en suspenso o detenido. La clase `Thread` es la única manera de controlar el comportamiento de los hilos. En la siguiente instrucción se muestra como acceder al hilo en ejecución actual:

```
Thread t = Thread.currentThread(); // El hilo actual se almacena en la variable t
```

## INTERFACE: Runnable

---

Si queremos tener más de un hilo necesitamos crear otra instancia de `Thread`. Cuando construimos una nueva instancia de `Thread`, necesitamos decirle que código ejecutar en el



nuevo hilo de control. Se puede comenzar un hilo sobre cualquier objeto que implemente la interfaz Runnable.

Runnable es una interfaz simple que abstrae la noción de que se desea que algún código se "ejecute" asincrónicamente. Para implementar Runnable, a una clase le basta con implementar un solo método llamado run. Este es un ejemplo que crea un nuevo hilo.

```
class ThreadDemo implements Runnable {

    ThreadDemo() {
        Thread ct = Thread.currentThread();
        Thread t = new Thread(this, "demo Thread");
        System.out.println("hilo actual: " + ct);
        System.out.println("Hilo creado: " + t);
        t.start();
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            System.out.println("Interrumpido");
        }
        System.out.println("saliendo del hilo main");
    }

    public void run() {
        try {
            for (int y = 5; y > 0; y--) {
                System.out.println(" " + y);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("hijo interrumpido");
        }
        System.out.println("saliendo del hilo hijo");
    }

    public static void main (String args []) {
        new ThreadDemo();
    }
}
```

El hilo main crea un nuevo objeto Thread, con new Thread (this, "Demo Thread"), pasando this como primer argumento para indicar que queremos que el nuevo hilo llame al método run sobre este (this) objeto. A continuación llamamos a start, lo que inicia el hilo de la

ejecución a partir del método run. Después, el hilo main se duerme durante 3000 milisegundos antes de imprimir un mensaje y después termina. Demo Thread todavía está contando desde cinco cuando sucede esto. Se continúa ejecutando hasta que termina con el bucle de run. Esta es la salida después de cinco segundos:

```
C:\> java ThreadDemo
Hilo actual: Thread[main, 5, main]
Hilo creado: Thread[demo Thread, 5, main]
5
4
3
saliendo del hilo main
2
1
saliendo del hilo hijo
```

## PRIORIDADES DE LOS HILOS

---

El planificador de hilos hace uso de las prioridades de los mismos para decidir cuándo debe dejar a cada hilo que se ejecute, de manera que los hilos con mayor prioridad deben ejecutarse más a menudo que lo de menor prioridad. Cuando está ejecutándose un hilo de baja prioridad, y otro de mayor prioridad se despierta de su sueño, o de la espera por un operación de E/S, debe dejarse que se ejecute de manera inmediata, desalojando al hilo de menor prioridad. Cuando los hilos son de igual prioridad deben desalojarse los unos a los otros, cada cierto tiempo, utilizando el algoritmo circular round-robin para gestionar el acceso al la CPU.

En JDK 1.0 la planificación de hilos es un problema que no está completamente resuelto. Por lo que si pretendemos tener un comportamiento predecible sobre las aplicaciones deberemos utilizar hilos que, voluntariamente, cedan el control de la CPU.

## SINCRONIZACIÓN

---

Cuando dos o más hilos necesitan acceder de manera simultánea a un recurso de datos compartido necesitan asegurarse de que sólo uno de ellos accede al mismo cada vez. Java proporciona un soporte único, el monitor, es un objeto que se utiliza como cerrojo exclusivo. Solo uno de los hilos puede ser el propietario de un monitor en un instante dado. Los restantes hilos que estuviesen intentando acceder al monitor bloqueado quedan en suspenso hasta que el hilo propietario salga del monitor.

Todos los objetos de Java disponen de un monitor propio implícitamente asociado a ellos. La manera de acceder a un objeto monitor es llamando a un método marcado con la palabra

clave synchronized. Durante todo el tiempo en que un hilo permanezca en un método sincronizado, los demás hilos que intenten llamar a un método sincronizado sobre la misma instancia tendrán que esperar. Para salir del monitor y permitir el control del objeto al siguiente hilo en espera, el propietario del monitor sólo tiene que volver del método.

## La sentencia synchronized

Si se utiliza una clase que no fue diseñada para accesos multihilo y, por ello, dispone de métodos no sincronizados que manipulan el estado interno, puede envolver la llamada al método en un bloque sincronizado. El formato general de la sentencia sincronizada es el siguiente:

```
synchronized(objeto) sentencia;
```

En el ejemplo, objeto es cualquier referencia al objeto, y sentencia suele ser un bloque que incluye una llamada al método de objeto, que solo tendrá lugar una vez que el hilo haya entrado con éxito en el monitor de objeto. Ahora veremos las formas de sincronización con un ejemplo:

```
class Callme {
    void call (String msg) { * también podía haber puesto synchronized antes de void *
        System.out.print "[" + msg);
        try Thread.sleep(1000); catch (Exception e);
        System.out.println("]");
    }
}

class caller implements Runnable {
    String msg;
    Callme target;
    public caller(Callme t, String s) {
        target = t;
        msg = s;
        new Thread(this).start();
    }
    public void run() {
        synchronized(target) {
            target.call(msg);
        }
    }
}

class Synch {
    public static void main(String args[]) {
```

```
Callme target = new Callme();
new caller(target, "Hola");
new caller(target, "Mundo");
new caller(target, "Sincronizado");
}
}
```

Este programa imprime por pantalla el literal "Hola Mundo Sincronizado", cada palabra en una línea y entre comillas, se crea una instancia de Callme y tres instancias de caller que cada una de ellas referencia al mismo Callme con lo que necesitamos de una sincronización para el acceso a Callme, pues sino se mezclarían las tres llamadas al haber una sentencia sleep que retrasa la ejecución de Callme dando lugar a que antes de que acabe un proceso deje libre el acceso a dicho objeto.

## COMUNICACIÓN ENTRE HILOS

Veamos, por ejemplo, el problema clásico de las colas, donde uno de los hilos produce datos y otro los consume. Para que el problema sea más interesante supongamos que el productor tiene que esperar a que el consumidor haya terminado, para empezar a producir más datos. En un sistema basado en sondeo el consumidor estaría desperdiciando ciclos de CPU mientras espera a que el productor produzca. Una vez que el productor ha terminado, se queda sondeando hasta ver que el consumidor ha finalizado, y así sucesivamente.

Evidentemente, hay una forma mejor de hacerlo. Java proporciona un mecanismo elegante de comunicación entre procesos, a través de los métodos wait, notify y notifyAll. Estos métodos se implementan como métodos de final en Object, de manera que todas las clases disponen de ellos. Cualquiera de los tres métodos sólo puede ser llamado desde dentro de un método synchronized.

- **wait:** le indica al hilo en curso que abandone el monitor y se vaya a dormir hasta que otro hilo entre en el mismo monitor y llame a notify.
- **notify:** despierta al primer hilo que realizó una llamada a wait sobre el mismo objeto.
- **notifyAll\_:** despierta todos los hilos que realizaron una llamada a wait sobre el mismo objeto. El hilo con mayor prioridad de los despertados es el primero en ejecutarse.

El ejemplo del productor y el consumidor es en Java como sigue:

```
class Q {
```

```
int n;
boolean valueSet = false;

synchronized int get() {
    if (!valueSet)
        try wait(); catch (InterruptedException e);
    System.out.println("Obtenido: " + n);
    valueSet = false;
    notify();
    return n;
}

synchronized void put(int n) {
    if (valueSet)
        try wait(); catch (InterruptedException e);
    this.n = n;
    valueSet = true;
    System.out.println("Colocado: " + n);
    notify();
}
}
```

```
class Producer implements Runnable {

    Q q;

    Producer (Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int y = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

```
class Consumer implements Runnable {  
  
    Q q;  
  
    Consumer(Q q) {  
        this.q = q;  
        new Thread(this, "Consumer").start();  
    }  
  
    public void run() {  
        while(true) {  
            q.get();  
        }  
    }  
}
```

```
class PC {  
  
    public static void main(String args[]) {  
        Q q = new Q();  
        new Producer(q);  
        new Consumer(q);  
    }  
}
```

## Bloqueos

Los bloqueos son condiciones anómalas inusuales, pero muy difíciles de depurar, donde dos hilos presentan una dependencia circular sobre un par de objetos sincronizados. Por ejemplo, si un hilo entra en el monitor sobre el objeto X y otro hilo entra en el monitor sobre el objeto Y, y si X intenta llamar a cualquier método sincronizado sobre Y, tal y como cabe esperar quedará detenido. Sin embargo, si Y, por su parte, intenta llamar a cualquier método sincronizado con X, entonces quedará esperando indefinidamente, ya que para conseguir el cerrojo de X tendría antes que liberar su propio cerrojo en Y, con el fin de que el primer hilo pudiera completarse.

---

## RESUMEN DE LA INTERFAZ DE PROGRAMACIÓN (API) DE HILOS

---

Se incluye a continuación una referencia rápida a todos los métodos de la clase Thread que se han comentado en este capítulo.

### Métodos de clase

Estos son los métodos estáticos que deben llamarse de manera directa en la clase Thread.

- **currentThread:** el método estático devuelve el objeto Thread que representa al hilo que se ejecuta actualmente.
- **yield:** este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que los hilos de menor prioridad no sufran inanición.
- **Sleep(int n):** el método sleep provoca que el intérprete ponga al hilo en curso a dormir durante n milisegundos. Una vez transcurridos los n milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes Java nos serán capaces de obtener precisiones mayores de 10 milisegundos.

### Métodos de instancia

- **start:** indica al intérprete de Java que cree un contexto de hilo del sistema y comience a ejecutarlo. A continuación, el método run objeto de este hilo será llamado en el nuevo contexto del hilo. Debe tomarse la precaución de no llamar al método start más de una vez sobre un objeto hilo dado.
- **run:** constituye el cuerpo de un hilo en ejecución. Este es el único método de la interfaz Runnable. Es llamado por el método start después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método run devuelva el control, el hilo actual se detendrá.
- **stop:** provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la línea inmediatamente posterior a la llamada al método stop no llega a ejecutarse jamás, pues el contexto del hilo muere antes de que stop devuelva el control.
- **suspend:** es distinto de stop. suspend toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo

anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a `resume` sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

- **resume:** se utiliza para revivir un hilo suspendido.
- **setPriority(int p):** asigna al hilo la prioridad indicada por el valor entero pasado como parámetro.
- **getPriority:** devuelve la prioridad del hilo en curso, que es un valor entre 1 y 10.
- **setName(String nombre):** identifica al hilo con un nombre mnemónico. De esta manera se facilita la depuración de programas multihilo.
- **getName:** devuelve el valor actual, de tipo cadena, asignado como nombre al hilo mediante `setName`.



# Capítulo 2

## Excepciones

### MANEJO DE ERRORES UTILIZANDO EXCEPCIONES

---

Existe una regla de oro en el mundo de la programación: en los programas ocurren errores.

Esto es sabido. Pero ¿qué sucede realmente después de que ha ocurrido el error? ¿Cómo se maneja el error? ¿Quién lo maneja?, ¿Puede recuperarlo el programa?

El lenguaje Java utiliza excepciones para proporcionar capacidades de manejo de errores. En esta lección aprenderás qué es una excepción, cómo lanzar y capturar excepciones, qué hacer con una excepción una vez capturada, y cómo hacer un mejor uso de las excepciones heredadas de las clases proporcionadas por el entorno de desarrollo de Java.

### ¿QUÉ ES UN EXCEPCIÓN Y POR QUÉ DEBO TENER CUIDADO?

---

El término excepción es una forma corta de la frase "suceso excepcional" y puede definirse de la siguiente forma.

#### **Definición:**

*Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.*

Muchas clases de errores pueden utilizar excepciones -- desde serios problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites. Cuando dicho error ocurre dentro de un método Java, el método crea un objeto 'exception' y lo maneja fuera, en el sistema de ejecución. Este objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando ocurrió el error. El sistema de ejecución es el responsable de buscar algún código para manejar el error. En terminología java, crear un objeto exception y manejarlo por el sistema de ejecución se llama lanzar una excepción.

Después de que un método lance una excepción, el sistema de ejecución entra en acción para buscar el manejador de la excepción. El conjunto de "algunos" métodos posibles para manejar la excepción es el conjunto de métodos de la pila de llamadas del método donde ocurrió el error. El sistema de ejecución busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el "manejador de excepción" adecuado.

Un manejador de excepción es considerado adecuado si el tipo de la excepción lanzada es el mismo que el de la excepción manejada por el manejador. Así la excepción sube sobre la pila de llamadas hasta que encuentra el manejador apropiado y una de las llamadas a métodos maneja la excepción, se dice que el manejador de excepción elegido captura la excepción.

Si el sistema de ejecución busca exhaustivamente por todos los métodos de la pila de llamadas sin encontrar el manejador de excepción adecuado, el sistema de ejecución finaliza (y consecuentemente y el programa Java también).

Mediante el uso de excepciones para manejar errores, los programas Java tienen las siguientes ventajas frente a las técnicas de manejo de errores tradicionales.

- Ventaja 1: Separar el Manejo de Errores del Código "Normal"
- Ventaja 2: Propagar los Errores sobre la Pila de Llamadas
- Ventaja 3: Agrupar los Tipos de Errores y la Diferenciación de éstos

### **Ventaja 1: Separar el Manejo de Errores del Código "Normal"**

En la programación tradicional, la detección, el informe y el manejo de errores se convierten en un código muy liado. Por ejemplo, supongamos que tenemos una función que lee un fichero completo dentro de la memoria. En pseudo-código, la función se podría parecer a esto.

```
leerFichero {  
    abrir el fichero;  
    determinar su tamaño;  
    asignar suficiente memoria;  
    leer el fichero a la memoria;  
    cerrar el fichero;  
}
```

A primera vista esta función parece bastante sencilla, pero ignora todos aquellos errores potenciales.

- ¿Qué sucede si no se puede abrir el fichero?
- ¿Qué sucede si no se puede determinar la longitud del fichero?
- ¿Qué sucede si no hay suficiente memoria libre?
- ¿Qué sucede si la lectura falla?

- ¿Qué sucede si no se puede cerrar el fichero?

Para responder a estas cuestiones dentro de la función, tendríamos que añadir mucho código para la detección y el manejo de errores. El aspecto final de la función se parecería esto.

```
codigodeError leerFichero {
  inicializar codigodeError = 0;
abrir el fichero;
  if (ficheroAbierto) {
    determinar la longitud del fichero;
    if (obtenerLongitudDelFichero) {
      asignar suficiente memoria;
      if (obtenerSuficienteMemoria) {
        leer el fichero a memoria;
        if (falloDeLectura) {
          codigodeError = -1;
        }
      } else {
        codigodeError = -2;
      }
    } else {
      codigodeError = -3;
    }
  }
  cerrar el fichero;
  if (ficheroNoCerrado && codigodeError == 0) {
    codigodeError = -4;
  } else {
    codigodeError = codigodeError and -4;
  }
} else {
  codigodeError = -5;
}
return codigodeError;
}
```

Con la detección de errores, las 7 líneas originales (en negrita) se han convertido en 29 líneas de código, aumentado casi un 400 %. Lo peor, existe tanta detección y manejo de errores y de retorno que en las 7 líneas originales y el código está totalmente atestado. Y aún peor, el flujo lógico del código también se pierde, haciendo difícil poder decir si el código hace lo correcto (si ¿se cierra el fichero realmente si falla la asignación de memoria?) e incluso es difícil asegurar que el código continúe haciendo las cosas correctas cuando se modifique la función tres meses después de haberla escrito. Muchos programadores

"resuelven" este problema ignorándolo, se informa de los errores cuando el programa no funciona.

Java proporciona una solución elegante al problema del tratamiento de errores: las excepciones. Las excepciones le permiten escribir el flujo principal de su código y tratar los casos excepcionales en otro lugar. Si la función **leerFichero** utilizara excepciones en lugar de las técnicas de manejo de errores tradicionales se podría parecer a esto.

```
leerFichero {  
    try {  
        abrir el fichero;  
        determinar su tamaño;  
        asignar suficiente memoria;  
        leer el fichero a la memoria;  
        cerrar el fichero;  
    } catch (falloAbrirFichero) {  
        hacerAlgo;  
    } catch (falloDeterminacionTamaño) {  
        hacerAlgo;  
    } catch (falloAsignaciondeMemoria) {  
        hacerAlgo;  
    } catch (falloLectura) {  
        hacerAlgo;  
    } catch (falloCerrarFichero) {  
        hacerAlgo;  
    }  
}
```

Observa que las excepciones no evitan el esfuerzo de hacer el trabajo de detectar, informar y manejar errores. Lo que proporcionan las excepciones es la posibilidad de separar los detalles oscuros de qué hacer cuando ocurre algo fuera de la normal.

Además, el factor de aumento de código de este es programa es de un 250% -- comparado con el 400% del ejemplo anterior.

## Ventaja 2: Propagar los Errores sobre la Pila de Llamadas

Una segunda ventaja de las excepciones es la posibilidad del propagar el error encontrado sobre la pila de llamadas a métodos. Supongamos que el método leerFichero es el cuarto método en una serie de llamadas a métodos anidadas realizadas por un programa principal: metodo1 llama a metodo2, que llama a metodo3, que finalmente llama a leerFichero.

```
metodo1 {  
    call metodo2;
```

```
}  
  
metodo2 {  
    call metodo3;  
}  
  
metodo3 {  
    call leerFichero;  
}
```

Supongamos también que metodo1 es el único método interesado en el error que ocurre dentro de leerFichero. Tradicionalmente las técnicas de notificación del error forzarían a metodo2 y metodo3 a propagar el código de error devuelto por leerFichero sobre la pila de llamadas hasta que el código de error llegue finalmente a metodo1 -- el único método que está interesado en él.

```
metodo1 {  
    codigodeErrorType error;  
    error = call metodo2;  
    if (error)  
        procesodelError;  
    else  
        proceder;  
}  
  
codigodeErrorType metodo2 {  
    codigodeErrorType error;  
    error = call metodo3;  
    if (error)  
        return error;  
    else  
        proceder;  
}  
  
codigodeErrorType metodo3 {  
    codigodeErrorType error;  
    error = call leerFichero;  
    if (error)  
        return error;  
    else  
        proceder;  
}
```

Como se aprendió anteriormente, el sistema de ejecución Java busca hacia atrás en la pila de llamadas para encontrar cualquier método que esté interesado en manejar una excepción particular. Un método Java puede "esquivar" cualquier excepción lanzada dentro de él, por lo tanto permite a los métodos que están por encima de él en la pila de llamadas poder capturarlo. Sólo los métodos interesados en el error deben preocuparse de detectarlo.

```
metodo1 {  
    try {  
        call metodo2;  
    } catch (excepcion) {  
        procesodelError;  
    }  
}  
  
metodo2 throws excepcion {  
    call metodo3;  
}  
  
metodo3 throws excepcion {  
    call leerFichero;  
}
```

Sin embargo, como se puede ver desde este pseudo-código, requiere cierto esfuerzo por parte de los métodos centrales. Cualquier excepción chequeada que pueda ser lanzada dentro de un método forma parte del interface de programación público del método y debe ser especificado en la clausula throws del método. Así el método informa a su llamador sobre las excepciones que puede lanzar, para que el llamador pueda decidir concienzuda e inteligentemente qué hacer con esa excepción.

Observa de nuevo la diferencia del factor de aumento de código y el factor de ofuscación entre las dos técnicas de manejo de errores. El código que utiliza excepciones es más compacto y más fácil de entender.

### Ventaja 3: Agrupar Errores y Diferenciación

Frecuentemente las excepciones se dividen en categorías o grupos. Por ejemplo, podríamos imaginar un grupo de excepciones, cada una de las cuales representara un tipo de error específico que pudiera ocurrir durante la manipulación de un array: el índice está fuera del rango del tamaño del array, el elemento que se quiere insertar en el array no es del tipo correcto, o el elemento que se está buscando no está en el array. Además, podemos imaginar que algunos métodos querrían manejar todas las excepciones de esa categoría (todas las excepciones de array), y otros métodos podría manejar sólo algunas excepciones específicas (como la excepción de índice no válido).

Como todas las excepciones lanzadas dentro de los programas Java son objetos de primera clase, agrupar o categorizar las excepciones es una salida natural de las clases y las superclases. Las excepciones Java deben ser ejemplares de la clase Throwable, o de cualquier descendiente de ésta. Como de las otras clases Java, se pueden crear subclases de la clase Throwable y subclases de estas subclases. Cada clase 'hoja' (una clase sin subclases) representa un tipo específico de excepción y cada clase 'nodo' (una clase con una o más subclases) representa un grupo de excepciones relacionadas.

InvalidIndexException, ElementTypeException, y NoSuchElementException son todas clases hojas. Cada una representa un tipo específico de error que puede ocurrir cuando se manipula un array. Un método puede capturar una excepción basada en su tipo específico (su clase inmediata o interface). Por ejemplo, un manejador de excepción que sólo controle la excepción de índice no válido, tiene una sentencia catch como esta.

```
catch (InvalidIndexException e) {  
    ...  
}
```

ArrayException es una clase nodo y representa cualquier error que pueda ocurrir durante la manipulación de un objeto array, incluyendo aquellos errores representados específicamente por una de sus subclases. Un método puede capturar una excepción basada en este grupo o tipo general especificando cualquiera de las superclases de la excepción en la sentencia catch. Por ejemplo, para capturar todas las excepciones de array, sin importar sus tipos específicos, un manejador de excepción especificaría un argumento ArrayException.

```
catch (ArrayException e) {  
    ...  
}
```

Este manejador podría capturar todas las excepciones de array, incluyendo InvalidIndexException, ElementTypeException, y NoSuchElementException. Se puede descubrir el tipo de excepción preciso que ha ocurrido comprobando el parámetro del manejador e. Incluso podríamos seleccionar un manejador de excepciones que controlara cualquier excepción con este manejador.

```
catch (Exception e) {  
    ...  
}
```

Los manejadores de excepciones que son demasiado generales, como el mostrado aquí, pueden hacer que el código sea propenso a errores mediante la captura y manejo de excepciones que no se hubieran anticipado y por lo tanto no son manejadas correctamente

dentro de manejador. Como regla no se recomienda escribir manejadores de excepciones generales.

Como has visto, se pueden crear grupos de excepciones y manejarlas de una forma general, o se puede especificar un tipo de excepción específico para diferenciar excepciones y manejarlas de un modo exacto.

## PRIMER ENCUENTRO CON LAS EXCEPCIONES JAVA

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException
must be caught, or it must be declared in throws clause of this method.
    fis = new FileInputStream(filename);
        ^
```

El mensaje de error anterior es uno de los dos mensajes similares que verás si intentas compilar la clase `InputFile`, porque la clase `InputFile` contiene llamadas a métodos que lanzan excepciones cuando se produce un error. El lenguaje Java requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas desde dentro del ámbito de ese método. (Los detalles sobre lo que ocurre los puedes ver en la próxima página [Requerimientos de Java para Capturar o Especificar](#).)

Si el compilador detecta un método, como los de `InputFile`, que no cumplen este requerimiento, muestran un error como el anterior y no compila el programa.

Echemos un vistazo a `InputFile` en más detalle y veamos que sucede.

La clase `InputFile` envuelve un canal `FileInputStream` y proporciona un método, `getLine()`, para leer una línea en la posición actual del canal de entrada.

```
// Nota: Esta clase no se compila por diseño!
import java.io.*;

class InputFile {

    FileInputStream fis;

    InputFile(String filename) {
        fis = new FileInputStream(filename);
    }

    String getLine() {
        int c;
        StringBuffer buf = new StringBuffer();
```



```
do {
    c = fis.read();
    if (c == '\n')           // nueva línea en UNIX
        return buf.toString();
    else if (c == '\r') {    // nueva línea en Windows 95/NT
        c = fis.read();
        if (c == '\n')
            return buf.toString();
        else {
            buf.append((char)\r);
            buf.append((char)c);
        }
    } else
        buf.append((char)c);
} while (c != -1);
return null;
}
```

El compilador dará el primer error en la primera línea que está en **negrita**. Esta línea crea un objeto `FileInputStream` y lo utiliza para abrir un fichero (cuyo nombre se pasa dentro del constructor del `FileInputStream`).

Entonces, ¿Qué debe hacer el `FileInputStream` si el fichero no existe? Bien, eso depende de lo que quiera hacer el programa que utiliza el `FileInputStream`. Los implementadores de `FileInputStream` no tenían ni idea de lo que quiere hacer la clase `InputFile` si no existe el fichero. ¿Debe `FileInputStream` terminar el programa? ¿Debe intentar un nombre alternativo? o ¿debera crear un fichero con el nombre indicado? No existe una forma posible de que los implementadores de `FileInputStream` pudieran elegir una solución que sirviera para todos los usuarios de `FileInputStream`. Por eso ellos lanzaron una excepción. Esto es, si el fichero nombrado en el argumento del constructor de `FileInputStream` no existe, el constructor lanza una excepción `java.io.FileNotFoundException`. Mediante el lanzamiento de esta excepción, `FileInputStream` permite que el método llamador maneje ese error de la forma que considere más apropiada.

Como puedes ver en el listado, la clase `InputFile` ignora completamente el hecho de que el constructor de `FileInputStream` puede lanzar una excepción. Sin embargo, El lenguaje Java requiere que un método o bien lance o especifique todas las excepciones chequeadas que pueden ser lanzadas desde dentro de su ámbito. Como la Clase `InputFile` no hace ninguna de las dos cosas, el compilador rehusa su compilación e imprime el mensaje de error.

Además del primer error mostrado arriba, se podrá ver el siguiente mensaje de error cuando se compile la clase InputFile.

```
InputFile.java:15: Warning: Exception java.io.IOException must be caught,  
or it must be declared in throws clause of this method.
```

```
    while ((c = fis.read()) != -1) {  
        ^
```

El método `getLine()` de la clase `InputFile` lee una línea desde el `FileInputStream` que fue abierto por el constructor de `InputFile`. El método `read()` de `FileInputStream` lanza la excepción `java.io.IOException` si por alguna razón no pudiera leer el fichero. De nuevo, la clase `InputFile` no hace ningún intento por capturar o especificar esta excepción lo que se convierte en el segundo mensaje de error.

En este punto, tenemos dos opciones. Se puede capturar las excepciones con los métodos apropiados en la clase `InputFile`, o se puede esquivarlas y permitir que otros métodos anteriores en la pila de llamadas las capturen. De cualquier forma, los métodos de `InputFile` deben hacer algo, o capturar o especificar las excepciones, antes de poder compilar la clase `InputFile`. Aquí tiene la clase `InputFileDeclared`, que corrige los errores de `InputFile` mediante la especificación de las excepciones.

La siguiente página le describe con más detalles los Requerimientos de Java para Capturar o Especificar. Las páginas siguientes le enseñarán cómo cumplir estos requerimientos.

## REQUERIMIENTOS PARA CAPTURAR O ESPECIFICAR EXCEPCIONES

---

Como se mencionó anteriormente, Java requiere que un método o capture o especifique todas las excepciones chequeadas que se pueden lanzar dentro de su ámbito. Este requerimiento tiene varios componentes que necesitan una mayor descripción.

### Capturar

Un método puede capturar una excepción proporcionando un manejador para ese tipo de excepción. La página siguiente, *Tratar con Excepciones*, introduce un programa de ejemplo, le explica cómo capturar excepciones, y le muestra cómo escribir un manejador de excepciones para el programa de ejemplo.

### Especificar

Si un método decide no capturar una excepción, debe especificar que puede lanzar esa excepción. ¿Por qué hicieron este requerimiento los diseñadores de Java? Porque una excepción que puede ser lanzada por un método es realmente una parte del interface de

programación público del método: los llamadores de un método deben conocer las excepciones que ese método puede lanzar para poder decidir inteligente y concienzudamente qué hacer son esas excepciones. Así, en la firma del método debe especificar las excepciones que el método puede lanzar.

La siguiente página, Tratar con Excepciones, le explica la especificación de excepciones que un método puede lanzar y le muestra cómo hacerlo.

## Excepciones Chequeadas

Java tiene diferentes tipos de excepciones, incluyendo las excepciones de I/O, las excepciones en tiempo de ejecución, y las de su propia creación. Las que nos interesan a nosotros para esta explicación son las excepciones en tiempo de ejecución, Estas excepciones son aquellas que ocurren dentro del sistema de ejecución de Java. Esto incluye las excepciones aritméticas (como dividir por cero), excepciones de puntero (como intentar acceder a un objeto con una referencia nula), y excepciones de indexación (como intentar acceder a un elemento de un array con un índice que es muy grande o muy pequeño).

Las excepciones en tiempo de ejecución pueden ocurrir en cualquier parte de un programa y en un programa típico pueden ser muy numerosas. Muchas veces, el costo de chequear todas las excepciones en tiempo de ejecución excede de los beneficios de capturarlas o especificarlas. Así el compilador no requiere que se capturen o especifiquen estas excepciones, pero se puede hacer.

Las excepciones chequeadas son excepciones que no son excepciones en tiempo de ejecución y que son chequeadas por el compilador (esto es, el compilador comprueba que esas excepciones son capturadas o especificadas).

Algunas veces esto se considera como un bucle cerrado en el mecanismo de manejo de excepciones de Java y los programadores se ven tentados a convertir todas las excepciones en excepciones en tiempo de ejecución. En general, esto no está recomendado.

La controversia -- Excepciones en Tiempo de Ejecución contiene una explicación detallada sobre cómo utilizar las excepciones en tiempo de ejecución.

## Excepciones que pueden ser lanzadas desde el ámbito de un método

Esta sentencia podría parecer obvia a primera vista: sólo hay que fijarse en la sentencia throw. Sin embargo, esta sentencia incluye algo más no sólo las excepciones que pueden ser lanzadas directamente por el método: la clave esta en la frase dentro del ámbito de. Esta frase incluye cualquier excepción que pueda ser lanzada mientras el flujo de control permanezca dentro del método. Así, esta sentencia incluye.

- excepciones que son lanzadas directamente por el método con la sentencia throw de Java, y
- las excepciones que son lanzadas por el método indirectamente a través de llamadas a otros métodos.

## TRATAR CON LAS EXCEPCIONES JAVA

---

Primer Encuentro con las Excepciones de Java describió brevemente cómo fue introducido en las excepciones Java: con un error del compilador indicando que las excepciones deben ser capturadas o especificadas. Luego Requerimientos de Java para la Captura o Especificación explicó qué significan exactamente los mensajes de error y por qué los diseñadores de Java decidieron hacer estos requerimientos. Ahora vamos a ver cómo capturar una excepción y cómo especificar otra.

### El ejemplo: ListOfNumbers

Las secciones posteriores, sobre como capturar y especificar excepciones, utilizan el mismo ejemplo. Este ejemplo define e implementa una clase llamada ListOfNumbers. Esta clase llama a dos clases de los paquetes de Java que pueden lanzar excepciones. Capturar y Manejar Excepciones mostrará cómo escribir manejadores de excepciones para las dos excepciones, y Especificar las Excepciones Lanzadas por un Método mostrará cómo especificar esas excepciones en lugar de capturarlas.

### Capturar y Manejar Excepciones

Una vez que te has familiarizado con la clase ListOfNumbers y con las excepciones que pueden ser lanzadas, puedes aprender cómo escribir manejadores de excepción que puedan capturar y manejar esas excepciones.

Esta sección cubre los tres componentes de un manejador de excepción -- los bloques try, catch, y finally -- y muestra cómo utilizarlos para escribir un manejador de excepción para el método writeList() de la clase ListOfNumbers. Además, esta sección contiene una página que pasea a lo largo del método writeList() y analiza lo que ocurre dentro del método en varios escenarios.

### Especificar las Excepciones que pueden ser Lanzadas por un Método

Si no es apropiado que un método capture y maneje una excepción lanzada por un método que él ha llamado, o si el método lanza su propia excepción, debe especificar en la firma del método que éste puede lanzar una excepción. Utilizando la clase ListOfNumbers, esta sección le muestra cómo especificar las excepciones lanzadas por un método.

## EL EJEMPLO: ListOfNumbers

Las dos secciones siguientes que cubren la captura y especificación de excepciones utilizan este ejemplo.

```
import java.io.*;
import java.util.Vector;

class ListOfNumbers {

    private Vector victor;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        victor = new Vector(size);
        for (i = 0; i < size; i++)
            victor.addElement(new Integer(i));
    }

    public void writeList() {
        PrintStream pStr = null;
        System.out.println("Entering try statement");
        int i;
        pStr = new PrintStream(new BufferedOutputStream(new FileOutputStream("OutFile.txt")));
        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));
        pStr.close();
    }
}
```

Este ejemplo define e implementa una clase llamada ListOfNumbers. Sobre su construcción, esta clase crea un Vector que contiene diez elementos enteros con valores secuenciales del 0 al 9. Esta clase también define un método llamado writeList() que escribe los números de la lista en un fichero llamado "OutFile.txt".

El método writeList() llama a dos métodos que pueden lanzar excepciones. Primero la siguiente línea invoca al constructor de FileOutputStream, que lanza una excepción IOException si el fichero no puede ser abierto por cualquier razón.

```
pStr = new PrintStream(new BufferedOutputStream(new FileOutputStream("OutFile.txt")));
```

Segundo, el método `elementAt()` de la clase `Vector` lanza una excepción `ArrayIndexOutOfBoundsException` si se le pasa un índice cuyo valor sea demasiado pequeño (un número negativo) o demasiado grande (mayor que el número de elementos que contiene realmente el `Vector`). Aquí está cómo `ListOfNumbers` invoca a `elementAt()`.

```
pStr.println("Value at: " + i + " = " + vector.elementAt(i));
```

Si se intenta compilar la clase `ListOfNumbers`, el compilador dará un mensaje de error sobre la excepción lanzada por el constructor de `FileOutputStream`, pero no muestra ningún error sobre la excepción lanzada por `elementAt()`.

Esto es porque la excepción lanzada por `FileOutputStream`, es una excepción chequeada y la lanzada por `elementAt()` es una ejecución de tiempo de ejecución. Java sólo requiere que se especifiquen o capturen las excepciones chequeadas. Para más información, puedes ver [Requerimientos de Java para Capturar o Especificar](#).

La siguiente sección, [Captura y Manejo de Excepciones](#), le mostrará cómo escribir un manejador de excepción para el método `writeList()` de `ListOfNumbers`.

Después de esto, una sección llamada [Especificar las Excepciones Lanzadas por un Método](#), mostrará cómo especificar que el método `writeList()` lanza excepciones en lugar de capturarlas.

---

## CAPTURAR Y MANEJAR EXCEPCIONES

---

Las siguientes páginas muestran cómo construir un manejador de excepciones para el método `writeList()` descrito en El ejemplo: `ListOfNumbers`. Las tres primeras páginas listadas abajo describen tres componentes diferentes de un manejador de excepciones y le muestran cómo pueden utilizarse esos componentes en el método `writeList()`. La cuarta página trata sobre el método `writeList()` resultante y analiza lo que ocurre dentro del código de ejemplo a través de varios escenarios.

### El Bloque try

El primer paso en la escritura de un manejador de excepciones es poner la sentencia Java dentro de la cual se puede producir la excepción dentro de un bloque `try`. Se dice que el bloque `try` gobierna las sentencias encerradas dentro de él y define el ámbito de cualquier manejador de excepciones (establecido por el bloque `catch` subsecuente) asociado con él.

### Los bloques catch

Después se debe asociar un manejador de excepciones con un bloque `try` proporcionándole uno o más bloques `catch` directamente después del bloque `try`.

## El bloque finally

El bloque finally de Java proporciona un mecanismo que permite a sus métodos limpiarse a sí mismos sin importar lo que sucede dentro del bloque try. Se utiliza el bloque finally para cerrar ficheros o liberar otros recursos del sistema.

## Poniéndolo todo junto

Las secciones anteriores describen cómo construir los bloques de código try, catch, y finally para el ejemplo writeList(). Ahora, pasearemos sobre el código para investigar que sucede en varios escenarios.

## EL BLOQUE Try

---

El primer paso en la construcción de un manejador de excepciones es encerrar las sentencias que podrían lanzar una excepción dentro de un bloque try. En general, este bloque se parece a esto.

```
try {  
    sentencias Java  
}
```

El segmento de código etiquetado sentencias java está compuesto por una o más sentencias legales de Java que podrían lanzar una excepción.

Para construir un manejador de excepción para el método writeList() de la clase ListOfNumbers, se necesita encerrar la sentencia que lanza la excepción en el método writeList() dentro de un bloque try.

Existe más de una forma de realizar esta tarea. Podríamos poner cada una de las sentencias que potencialmente pudieran lanzar una excepción dentro de su propio bloque try, y proporcionar manejadores de excepciones separados para cada uno de los bloques try. O podríamos poner todas las sentencias de writeList() dentro de un sólo bloque try y asociar varios manejadores con él. El siguiente listado utiliza un sólo bloque try para todo el método porque el código tiende a ser más fácil de leer.

```
PrintStream pstr;  
try {  
    int i;  
    System.out.println("Entering try statement");  
    pStr = new PrintStream(new BufferedOutputStream(new FileOutputStream("OutFile.txt")));  
    for (i = 0; i < size; i++)  
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));  
}
```

```
}
```

Se dice que el bloque try gobierna las sentencias encerradas dentro del él y define el ámbito de cualquier manejador de excepción (establecido por su subsecuente bloque catch) asociado con él. En otras palabras, si ocurre una excepción dentro del bloque try, esta excepción será manejada por el manejador de excepción asociado con esta sentencia try.

Una sentencia try debe ir acompañada de al menos un bloque catch o un bloque finally.

## LOS BLOQUES catch

Como se aprendió en la página anterior, la sentencia try define el ámbito de sus manejadores de excepción asociados. Se pueden asociar manejadores de excepción a una sentencia try proporcionando uno o más bloques catch directamente después del bloque try.

```
try {  
    ...  
} catch ( ... ) {  
    ...  
} catch ( ... ) {  
    ...  
} ...
```

No puede haber ningún código entre el final de la sentencia try y el principio de la primera sentencia catch. La forma general de una sentencia catch en Java es esta.

```
catch (AlgunObjetoThrowable nombreVariable) {  
    Sentencias Java  
}
```

Como puedes ver, la sentencia catch requiere un sólo argumento formal. Este argumento parece un argumento de una declaración de método. El tipo del argumento AlgunObjetoThrowable declara el tipo de excepción que el manejador puede manejar y debe ser el nombre de una clase heredada de la clase Throwable definida en el paquete java.lang. (Cuando los programas Java lanzan una excepción realmente están lanzando un objeto, sólo pueden lanzarse los objetos derivados de la clase Throwable. Aprenderás cómo lanzar excepciones en la lección ¿Cómo Lanzar Excepciones?.)

nombreVariable es el nombre por el que el manejador puede referirse a la excepción capturada. Por ejemplo, los manejadores de excepciones para el método writeList() (mostrados más adelante) llaman al método getMessage() de la excepción utilizando el nombre de excepción declarado e.

```
e.getMessage()
```



Se puede acceder a las variables y métodos de las excepciones en la misma forma que accede a los de cualquier otro objeto. `getMessage()` es un método proporcionado por la clase `Throwable` que imprime información adicional sobre el error ocurrido. La clase `Throwable` también implementa dos métodos para rellenar e imprimir el contenido de la pila de ejecución cuando ocurre la excepción. Las subclases de `Throwable` pueden añadir otros métodos o variables de ejemplar. Para buscar qué métodos implementar en una excepción, se puede comprobar la definición de la clase y las definiciones de las clases antecesoras.

El bloque `catch` contiene una serie de sentencias Java legales. Estas sentencias se ejecutan cuando se llama al manejador de excepción. El sistema de ejecución llama al manejador de excepción cuando el manejador es el primero en la pila de llamadas cuyo tipo coincide con el de la excepción lanzada.

El método `writeList()` de la clase de ejemplo `ListOfNumbers` utiliza dos manejadores de excepción para su sentencia `try`, con un manejador para cada uno de los tipos de excepciones que pueden lanzarse dentro del bloque `try` -- `ArrayIndexOutOfBoundsException` y `IOException`.

```
try {  
    ...  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Caught IOException: " + e.getMessage());  
}
```

## Ocurre una `IOException`

Supongamos que ocurre una excepción `IOException` dentro del bloque `try`. El sistema de ejecución inmediatamente toma posesión e intenta localizar el manejador de excepción adecuado. El sistema de ejecución empieza buscando al principio de la pila de llamadas. Sin embargo, el constructor de `FileOutputStream` no tiene un manejador de excepción apropiado por eso el sistema de ejecución comprueba el siguiente método en la pila de llamadas -- el método `writeList()`. Este método tiene dos manejadores de excepciones: uno para `ArrayIndexOutOfBoundsException` y otro para `IOException`.

El sistema de ejecución comprueba los manejadores de `writeList()` por el orden en el que aparecen después del bloque `try`. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una `ArrayIndexOutOfBoundsException`, pero la excepción que se ha lanzado era una `IOException`. Una excepción `IOException` no puede asignarse legalmente a una

`ArrayIndexOutOfBoundsException`, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de `writeList()` es una `IOException`. La excepción lanzada por el constructor de `FileOutputStream` también es una `IOException` y por eso puede ser asignada al argumento del manejador de excepciones de `IOException`. Así, este manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia.

Caught `IOException`: `OutFile.txt`

El sistema de ejecución sigue un proceso similar si ocurre una excepción `ArrayIndexOutOfBoundsException`. Para más detalles puedes ver.

Poniéndolo todo Junto que te lleva a través de método `writeList()` después de haberlo completado (queda un paso más) e investiga lo que sucede en varios escenarios.

## Capturar Varios Tipos de Excepciones con Un Manejador

Los dos manejadores de excepción utilizados por el método `writeList()` son muy especializados. Cada uno sólo maneja un tipo de excepción. El lenguaje Java permite escribir manejadores de excepciones generales que pueden manejar varios tipos de excepciones.

Como ya sabes, las excepciones Java son objetos de la clase `Throwable` (son ejemplares de la clase `Throwable` o de alguna de sus subclases). Los paquetes Java contienen numerosas clases derivadas de la clase `Throwable` y así construyen un árbol de clases `Throwable`.

El manejador de excepción puede ser escrito para manejar cualquier clase heredada de `Throwable`. Si se escribe un manejador para una clase 'hoja' (una clase que no tiene subclases), se habrá escrito un manejador especializado: sólo maneja excepciones de un tipo específico. Si se escribe un manejador para una clase 'nodo' (una clase que tiene subclases), se habrá escrito un manejador general: se podrá manejar cualquier excepción cuyo tipo sea el de la clase nodo o de cualquiera de sus subclases.

Modifiquemos de nuevo el método `writeList()`. Sólo esta vez, escribamoslo para que maneje las dos excepciones `IOExceptions` y `ArrayIndexOutOfBoundsExceptions`. El antecesor más cercano de estas dos excepciones es la clase `Exception`. Así un manejador de excepción que quisiera manejar los dos tipos se parecería a esto.

```
try {  
    ...  
} catch (Exception e) {  
    System.err.println("Exception caught: " + e.getMessage());  
}
```

```
}
```

La clase `Exception` está bastante arriba en el árbol de herencias de la clase `Throwable`. Por eso, además de capturar los tipos de `IOException` y `ArrayIndexOutOfBoundsException` este manejador de excepciones, puede capturar otros muchos tipos. Generalmente hablando, los manejadores de excepción deben ser más especializados.

Los manejadores que pueden capturar la mayoría o todas las excepciones son menos utilizados para la recuperación de errores porque el manejador tiene que determinar qué tipo de excepción ha ocurrido de todas formas (para determinar la mejor estrategia de recuperación). Los manejadores de excepciones que son demasiado generales pueden hacer el código más propenso a errores mediante la captura y manejo de excepciones que no fueron anticipadas por el programador y para las que el manejador no está diseñado.

## EL BLOQUE `finally`

El paso final en la creación de un manejador de excepción es proporcionar un mecanismo que limpie el estado del método antes (posiblemente) de permitir que el control pase a otra parte diferente del programa. Se puede hacer esto encerrando el código de limpieza dentro de un bloque `finally`.

El bloque `try` del método `writeList()` ha estado trabajando con un `PrintStream` abierto. El programa debería cerrar ese canal antes de permitir que el control salga del método `writeList()`. Esto plantea un problema complicado, ya que el bloque `try` del `writeList()` tiene tres posibles salidas.

1. La sentencia `new FileOutputStream` falla y lanza una `IOException`.
2. La sentencia `victor.elementAt(i)` falla y lanza una `ArrayIndexOutOfBoundsException`.
3. Todo tiene éxito y el bloque `try` sale normalmente.

El sistema de ejecución siempre ejecuta las sentencias que hay dentro del bloque `finally` sin importar lo que suceda dentro del bloque `try`. Esto es, sin importar la forma de salida del bloque `try` del método `writeList()` debido a los escenarios 1, 2 ó 3 listados arriba, el código que hay dentro del bloque `finally` será ejecutado de todas formas.

Este es el bloque `finally` para el método `writeList()`. Limpia y cierra el canal `PrintStream`.

```
finally {  
    if (pStr != null) {  
        System.out.println("Closing PrintStream");  
        pStr.close();  
    }  
}
```

```
} else {  
    System.out.println("PrintStream not open");  
}  
}
```

### ¿Es realmente necesaria la sentencia finally?

La primera necesidad de la sentencia finally podría no aparecer de forma inmediata. Los programadores se preguntan frecuentemente "¿Es realmente necesaria la sentencia finally o es sólo azúcar para mi Java?" En particular los programadores de C++ dudan de la necesidad de esta sentencia porque C++ no la tiene.

Esta necesidad de la sentencia finally no aparece hasta que se considera lo siguiente: ¿cómo se podría cerrar el `PrintStream` en el método `writeList()` si no se proporcionara un manejador de excepción para la `ArrayIndexOutOfBoundsException` y ocurre una `ArrayIndexOutOfBoundsException`? (sería sencillo y legal omitir un manejador de excepción para `ArrayIndexOutOfBoundsException` porque es una excepción en tiempo de ejecución y el compilador no alerta de que `writeList()` contiene una llamada a un método que puede lanzar una).

La respuesta es que el `PrintStream` no se cerraría si ocurriera una excepción `ArrayIndexOutOfBoundsException` y `writeList()` no proporcionara un manejador para ella -- a menos que `writeList()` proporcionara una sentencia finally.

Existen otros beneficios de la utilización de la sentencia finally. En el ejemplo de `writeList()` es posible proporcionar un código de limpieza sin la intervención de una sentencia finally. Por ejemplo, podríamos poner el código para cerrar el `PrintStream` al final del bloque try y de nuevo dentro del manejador de excepción para `ArrayIndexOutOfBoundsException`, como se muestra aquí.

```
try {  
    ...  
    pStr.close();    // No haga esto, duplica el código  
} catch (ArrayIndexOutOfBoundsException e) {  
    pStr.close();    // No haga esto, duplica el código  
    System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Caught IOException: " + e.getMessage());  
}
```

Sin embargo, esto duplica el código, haciéndolo difícil de leer y propenso a errores si se modifica más tarde. Por ejemplo, si se añade código al bloque try que pudiera lanzar otro tipo de excepción, se tendría que recordar el cerrar el `PrintStream` dentro del nuevo manejador de excepción (lo que se olvidará seguro si se parece a mí).

## PONIÉNDOLO TODO JUNTO

Cuando se juntan todos los componentes, el método `writeList()` se parece a esto.

```
public void writeList() {  
  
    PrintStream pStr = null;  
    try {  
        int i;  
        System.out.println("Entrando en la Sentencia try");  
        pStr = new PrintStream(new BufferedOutputStream(new FileOutputStream("OutFile.txt")));  
        for (i = 0; i < size; i++)  
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());  
    } catch (IOException e) {  
        System.err.println("Caught IOException: " + e.getMessage());  
    } finally {  
        if (pStr != null) {  
            System.out.println("Cerrando PrintStream");  
            pStr.close();  
        } else {  
            System.out.println("PrintStream no está abierto");  
        }  
    }  
}
```

El bloque `try` de este método tiene tres posibilidades de salida diferentes.

1. La sentencia `new FileOutputStream` falla y lanza una `IOException`.
4. La sentencia `victor.elementAt(i)` falla y lanza una `ArrayIndexOutOfBoundsException`.
5. Todo tiene éxito y la sentencia `try` sale normalmente.

Esta página investiga en detalle lo que sucede en el método `writeList` durante cada una de esas posibilidades de salida.

### Escenario 1: Ocurre una excepción `IOException`

La sentencia `new FileOutputStream("OutFile.txt")` puede fallar por varias razones: el usuario no tiene permiso de escritura sobre el fichero o el directorio, el sistema de ficheros está

lleno, o no existe el directorio. Si cualquiera de estas situaciones es verdadera el constructor de `FileOutputStream` lanza una excepción `IOException`.

Cuando se lanza una excepción `IOException`, el sistema de ejecución para inmediatamente la ejecución del bloque `try`. Y luego intenta localizar un manejador de excepción apropiado para manejar una `IOException`.

El sistema de ejecución comienza su búsqueda al principio de la pila de llamadas. Cuando ocurrió la excepción, el constructor de `FileOutputStream` estaba al principio de la pila de llamadas. Sin embargo, este constructor no tiene un manejador de excepción apropiado por lo que el sistema comprueba el siguiente método que hay en la pila de llamadas -- el método `writeList()`. Este método tiene dos manejadores de excepciones: uno para `ArrayIndexOutOfBoundsException` y otro para `IOException`.

El sistema de ejecución comprueba los manejadores de `writeList()` por el orden en el que aparecen después del bloque `try`. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una `ArrayIndexOutOfBoundsException`, pero la excepción que se ha lanzado era una `IOException`. Una excepción `IOException` no puede asignarse legalmente a una `ArrayIndexOutOfBoundsException`, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de `writeList()` es una `IOException`. La excepción lanzada por el constructor de `FileOutputStream` también es una `IOException` y por eso puede ser asignada al argumento del manejador de excepciones de `IOException`. Así, este manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia.

Caught `IOException`: `OutFile.txt`

Después de que se haya ejecutado el manejador de excepción, el sistema pasa el control al bloque `finally`. En este escenario particular, el canal `PrintStream` nunca se ha abierto, así el `pStr` es `null` y no se cierra. Después de que se haya completado la ejecución del bloque `finally`, el programa continua con la primera sentencia después de este bloque.

La salida completa que se podrá ver desde el programa `ListOfNumbers` cuando se lanza un excepción `IOException` es esta.

Entrando en la sentencia `try`  
Caught `IOException`: `OutFile.txt`  
`PrintStream` no está abierto

## Escenario 2: Ocurre una excepción `ArrayIndexOutOfBoundsException`

Este escenario es el mismo que el primero excepto que ocurre un error diferente dentro del bloque try. En este escenario, el argumento pasado al método `elementAt()` de `Vector` está fuera de límites. Esto es, el argumento es menor que cero o mayor que el tamaño del array. (De la forma en que está escrito el código, esto es realmente imposible, pero supongamos que se ha introducido un error cuando alguien lo ha modificado).

Como en el escenario 1, cuando ocurre una excepción el sistema de ejecución para la ejecución del bloque try e intenta localizar un manejador de excepción apropiado para `ArrayIndexOutOfBoundsException`. El sistema busca como lo hizo anteriormente. Llega a la sentencia catch en el método `writeList()` que maneja excepciones del tipo `ArrayIndexOutOfBoundsException`. Como el tipo de la excepción corresponde con el del manejador, el sistema ejecuta el manejador de excepción.

Después de haber ejecutado el manejador de excepción, el sistema pasa el control al bloque finally. En este escenario particular, el canal `PrintStream` si que se ha abierto, así que el bloque finally lo cerrará. Después de que el bloque finally haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida completa que dará el programa `ListOfNumbers` si ocurre una excepción `ArrayIndexOutOfBoundsException`.

```
Entrando en la sentencia try  
Caught ArrayIndexOutOfBoundsException: 10 >= 10  
Cerrando PrintStream
```

## Escenario 3: El bloque try sale normalmente

En este escenario, todas las sentencias dentro del ámbito del bloque try se ejecutan de forma satisfactoria y no lanzan excepciones. La ejecución cae al final del bloque try y el sistema pasa el control al bloque finally. Como todo ha salido satisfactorio, el `PrintStream` abierto se cierra cuando el bloque finally consigue el control. De nuevo, Después de que el bloque finally haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida cuando el programa `ListOfNumbers` cuando no se lanzan excepciones.

```
Entrando en la sentencia try  
Cerrando PrintStream
```

## Especificar las Excepciones Lanzadas por un Método

Le sección anterior mostraba como escribir un manejador de excepción para el método `writeList()` de la clase `ListOfNumbers`. Algunas veces, es apropiado capturar las excepciones que ocurren pero en otras ocasiones, sin embargo, es mejor dejar que un método superior en la pila de llamadas maneje la excepción. Por ejemplo, si se está utilizando la clase `ListOfNumbers` como parte de un paquete de clases, probablemente no se querrá anticipar las necesidades de todos los usuarios de su paquete. En este caso, es mejor no capturar las excepciones y permitir que algún la capture más arriba en la pila de llamadas.

Si el método `writeList()` no captura las excepciones que pueden ocurrir dentro de él, debe especificar que puede lanzar excepciones. Modifiquemos el método `writeList()` para especificar que puede lanzar excepciones. Como recordatorio, aquí tienes la versión original del método `writeList()`.

```
public void writeList() {  
    System.out.println("Entrando en la sentencia try");  
    int i;  
    pStr = new PrintStream(new BufferedOutputStream(new FileOutputStream("OutFile.txt")));  
    for (i = 0; i < size; i++)  
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));  
}
```

Como recordarás, la sentencia `new FileOutputStream("OutFile.txt")` podría lanzar un excepción `IOException` (que no es una excepción en tiempo de ejecución). La sentencia `victor.elementAt(i)` puede lanzar una excepción `ArrayIndexOutOfBoundsException` (que es una subclase de la clase `RuntimeException`, y es una excepción en tiempo de ejecución).

Para especificar que `writeList()` lanza estas dos excepciones, se añade la cláusula `throws` a la firma del método de `writeList()`. La clausula `throws` está compuesta por la palabra clave `throws` seguida por una lista separada por comas de todas las excepciones lanzadas por el método. Esta cláusula va después del nombre del método y antes de la llave abierta que define el ámbito del método. Aquí tienes un ejemplo.

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
```

Recuerda que la excepción `ArrayIndexOutOfBoundsException` es una excepción en tiempo de ejecución, por eso no tiene porque especificarse en la sentencia `throws` pero puede hacerse si se quiere.

## LA SENTENCIAS throw

Todos los métodos Java utilizan la sentencia `throw` para lanzar una excepción.



Esta sentencia requiere un sólo argumento, un objeto Throwable. En el sistema Java, los objetos lanzables son ejemplares de la clase Throwable definida en el paquete java.lang. Aquí tienes un ejemplo de la sentencia throw.

```
throw algunObjetoThrowable;
```

Si se intenta lanzar un objeto que no es 'lanzable', el compilador rehúsa la compilación del programa y muestra un mensaje de error similar a éste.

```
testing.java:10: Cannot throw class java.lang.Integer; it must be a subclass
of class java.lang.Throwable.
    throw new Integer(4);
    ^
```

La página siguiente, La clase Throwable y sus Subclases, cuentan más cosas sobre la clase Throwable.

Echemos un vistazo a la sentencia throw en su contexto. El siguiente método está tomado de una clase que implementa un objeto pila normal. El método **pop()** saca el elemento superior de la pila y lo devuelve.

```
public Object pop() throws EmptyStackException {

    Object obj;

    if (size == 0)
        throw new EmptyStackException();
    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

El método pop() comprueba si hay algún elemento en la pila. Si la pila está vacía (su tamaño es igual a cero), ejemplariza un nuevo objeto de la clase EmptyStackException y lo lanza. Esta clase está definida en el paquete java.util. En páginas posteriores podrás ver cómo crear tus propias clases de excepciones. Por ahora, todo lo que necesitas recordar es que se pueden lanzar objetos heredados desde la clase Throwable.

## La clausula throws

Habrás observado que la declaración del método pop() contiene esta clausula.

```
throws EmptyStackException
```

La cláusula `throws` especifica que el método puede lanzar una excepción `EmptyStackException`. Como ya sabes, el lenguaje Java requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas dentro de su ámbito.

Se puede hacer esto con la cláusula `throws` de la declaración del método.

Para más información sobre estos requerimientos puedes ver [Requerimientos Java para Capturar o Especificar](#).

También puedes ver, [Especificar las Excepciones lanzadas por un Método](#) para obtener más detalles sobre cómo un método puede lanzar excepciones.

## LA CLASE `Throwable` Y SUS SUBCLASES

---

Como se aprendió en la página anterior, sólo se pueden lanzar objetos que estén derivados de la clase `Throwable`. Esto incluye descendientes directos (esto es, objetos de la clase `Throwable`) y descendiente indirectos (objetos derivados de hijos o nietos de la clase `Throwable`).

Este diagrama ilustra el árbol de herencia de la clase `Throwable` y sus subclases más importantes.

Como se puede ver en el diagrama, la clase `Throwable` tiene dos descendientes directos: `Error` y `Exception`.

### **Error**

Cuando falla un enlace dinámico, y hay algún fallo "hardware" en la máquina virtual, ésta lanza un error. Típicamente los programas Java no capturan los Errores. Pero siempre lanzarán errores.

### **Exception**

La mayoría de los programas lanzan y capturan objetos derivados de la clase `Exception`.

Una Excepción indica que ha ocurrido un problema pero que el problema no es demasiado serio.

La mayoría de los programas lanzarán y capturarán excepciones.

La clase `Exception` tiene muchos descendiente definidos en los paquetes Java. Estos descendientes indican varios tipos de excepciones que pueden ocurrir. Por ejemplo, `IllegalAccessException` señala que no se puede encontrar un método particular, y

NegativeArraySizeException indica que un programa intenta crear un array con tamaño negativo.

Una subclase de Exception tiene un significado especial en el lenguaje Java: RuntimeException.

## Excepciones en Tiempo de Ejecución

La clase RuntimeException representa las excepciones que ocurren dentro de la máquina virtual Java (durante el tiempo de ejecución). Un ejemplo de estas excepciones es NullPointerException, que ocurre cuando un método intenta acceder a un miembro de un objeto a través de una referencia nula. Esta excepción puede ocurrir en cualquier lugar en que un programa intente desreferenciar una referencia a un objeto. Frecuentemente el coste de chequear estas excepciones sobrepasa los beneficios de capturarlas.

Como las excepciones en tiempo de ejecución están omnipresentes e intentar capturar o especificarlas todas en todo momento podría ser un ejercicio infructuoso (y un código infructuoso, imposible de leer y de mantener), el compilador permite que estas excepciones no se capturen ni se especifiquen.

Los paquetes Java definen varias clases RuntimeException. Se pueden capturar estas excepciones al igual que las otras. Sin embargo, no se requiere que un método especifique que lanza excepciones en tiempo de ejecución. Además puedes crear sus propias subclases de RuntimeException.

Excepciones en Tiempo de Ejecución -- La Controversia contiene una explicación detallada sobre cómo utilizar las excepciones en tiempo de ejecución.

## CREAR CLASES DE EXCEPCIONES

---

Cuando diseñes un paquete de clases java que colabore para proporcionar alguna función útil a sus usuarios, deberás trabajar duro para asegurarte de que las clases interactúan correctamente y que sus interfaces son fáciles de entender y utilizar. Deberías estar mucho tiempo pensando sobre ello y diseñar las excepciones que esas clases pueden lanzar.

Supón que estás escribiendo una clase con una lista enlazada que estás pensando en distribuir como freeware. Entre otros métodos la clase debería soportar estos.

### **objectAt(int n)**

Devuelve el objeto en la posición n de la lista.

### **firstObject()**

Devuelve el primer objeto de la lista.

### **indexOf(Object o)**

Busca el Objeto especificado en la lista y devuelve su posición en ella.

## **¿Qué puede ir mal?**

Como muchos programadores utilizarán tu clase de lista enlazada, puedes estar seguro de que muchos de ellos la utilizarán mal o abusarán de los métodos de la clase. También, alguna llamada legítima a los métodos de la clase podría dar algún resultado indefinido. No importa, con respecto a los errores, querrás que tu clase sea lo más robusta posible, para hacer algo razonable con los errores, y comunicar los errores al programa llamador. Sin embargo, no puedes anticipar como quiere cada usuario de sus clases enlazadas se comporten sus objetos ante la adversidad. Por eso, lo mejor que puedes hacer cuando ocurre un error es lanzar una excepción.

Cada uno de los métodos soportados por la lista enlazada podría lanzar una excepción bajo ciertas condiciones, y cada uno podría lanzar un tipo diferente de excepción. Por ejemplo.

### **objectAt()**

Lanzará una excepción si se pasa un entero al método que sea menor que 0 o mayor que el número de objetos que hay realmente en la lista.

### **firstObject()**

Lanzará una excepción si la lista no contiene objetos.

### **indexOf()**

Lanzará una excepción si el objeto pasado al método no está en la lista.

Pero ¿qué tipo de excepción debería lanzar cada método? ¿Debería ser una excepción proporcionada por el entorno de desarrollo de Java? O ¿Deberían ser excepciones propias?

## **Elegir el Tipo de Excepción Lanzada**

Tratándose de la elección del tipo de excepción a lanzar, tienes dos opciones.

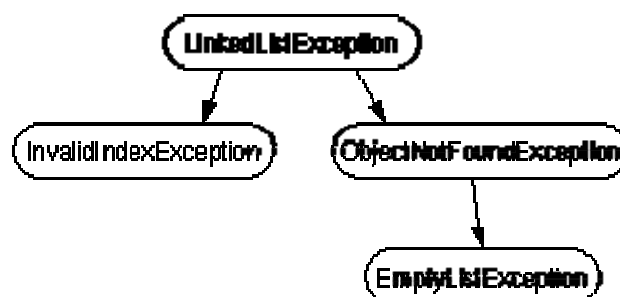
1. Utilizar una escrita por otra persona. Por ejemplo, el entorno de desarrollo de Java proporciona muchas clases de excepciones que podrías utilizar.
2. Escribirlas tu mismo.

Necesitarás escribir tus propias clases de excepciones si respondes "Si" a alguna de las siguientes preguntas. Si no es así, probablemente podrás utilizar alguna excepción ya escrita.

- ¿Necesitas un tipo de excepción que no está representada por lo existentes en el entorno de desarrollo de Java?
- ¿Ayudaría a sus usuarios si pudieran diferenciar sus excepciones de las otras lanzadas por clases escritas por por otros vendedores?
- ¿Lanza el código más una excepción relacionada?
- Si utilizas excepciones de otros, ¿Podrán sus usuarios tener acceso a estas excepciones? Una pregunta similar es "¿Debería tu paquete ser independiente y auto-contenedor?"

La clase de lista enlazada puede lanzar varias excepciones, y sería conveniente poder capturar todas las excepciones lanzadas por la lista enlazada con un manejador. Si planeas distribuir la lista enlazada en un paquete, todo el código relacionado debe empaquetarse junto. Así para la lista enlazada, deberías crear tu propio árbol de clases de excepciones.

El siguiente diagrama ilustra una posibilidad del árbol de clases para su lista enlazada.



LinkedListException es la clase padre de todas las posibles excepciones que pueden ser lanzadas por la clase de la lista enlazada. Los usuarios de esta clase pueden escribir un sólo manejador de excepciones para manejarlas todas con una sentencia catch como esta.

```
catch (LinkedListException) {
    ...
}
```

O, podrías escribir manejadores más especializados para cada una de las subclases de LinkedListException.

## Elegir una Superclase

El diagrama anterior no indica la superclase de la clase `LinkedListException`. Como ya sabes, las excepciones de Java deben ser ejemplares de la clase `Throwable` o de sus subclases.

Por eso podría tentarte hacer `LinkedListException` como una subclase de la clase `Throwable`.

Sin embargo, el paquete `java.lang` proporciona dos clases `Throwable` que dividen los tipos de problemas que pueden ocurrir en un programa java: Errores y Excepción. La mayoría de los applets y de las aplicaciones que escribes lanzan objetos que son Excepciones. (Los errores están reservados para problemas más serios que pueden ocurrir en el sistema.)

Teóricamente, cualquier subclase de `Exception` podría ser utilizada como padre de la clase `LinkedListException`. Sin embargo, un rápido examen de esas clases muestra que o son demasiado especializadas o no están relacionadas con `LinkedListException` para ser apropiadas.

Así que el padre de la clase `LinkedListException` debería ser `Exception`.

Como las excepciones en tiempo de ejecución no tienen por qué ser especificadas en la cláusula `throws` de un método, muchos desarrolladores de paquetes se preguntan.

"¿No es más sencillo hacer que todas mis excepciones sean heredadas de `RuntimeException`?" La respuesta a esta pregunta con más detalle en Excepciones en Tiempo de Ejecución -- La Controversia. La línea inferior dice que no deberías utilizar subclases de `RuntimeException` en tus clases a menos que tus excepciones sean realmente en tiempo de ejecución! Para la mayoría de nosotros, esto significa "NO, tus excepciones no deben descender de la clase `RuntimeException`."

## Convenciones de Nombres

Es una buena práctica añadir la palabra "Exception" al final del nombre de todas las clases heredadas (directa o indirectamente) de la clase `Exception`. De forma similar, los nombres de las clases que se hereden desde la clase `Error` deberían terminar con la palabra "Error".

## EXCEPCIONES EN TIEMPO DE EJECUCIÓN - LA CONTROVERSIDA

---

Como el lenguaje Java no requiere que los métodos capturen o especifiquen las excepciones en tiempo de ejecución, es una tentación para los programadores escribir código que lance sólo excepciones de tiempo de ejecución o hacer que todas sus subclases

de excepciones hereden de la clase `RuntimeException`. Estos atajos de programación permiten a los programadores escribir código Java sin preocuparse por los consiguientes.

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException must be caught,  
or it must be declared in throws clause of this method.
```

```
    fis = new FileInputStream(filename);  
    ^
```

Errores del compilador y sin preocuparse por especificar o capturar ninguna excepción.

Mientras esta forma parece conveniente para los programadores, esquivan los requerimientos de Java de capturar o especificar y pueden causar problemas a los programadores que utilicen tus clases.

¿Por qué decidieron los diseñadores de Java forzar a un método a especificar todas las excepciones chequeadas no capturadas que pueden ser lanzadas dentro de su ámbito?

Como cualquier excepción que pueda ser lanzada por un método es realmente una parte de la interfase de programación pública del método: los llamadores de un método deben conocer las excepciones que el método puede lanzar para poder decidir concienzuda e inteligentemente qué hacer con estas excepciones. Las excepciones que un método puede lanzar son como una parte de la interfase de programación del método como sus parámetros y devuelven un valor.

La siguiente pregunta podría ser: " Bien ¿Si es bueno documentar el API de un método incluyendo las excepciones que pueda lanzar, por qué no especificar también las excepciones de tiempo de ejecución?".

Las excepciones de tiempo de ejecución representan problemas que son detectados por el sistema de ejecución. Esto incluye excepciones aritméticas (como la división por cero), excepciones de punteros (como intentar acceder a un objeto con un referencia nula), y las excepciones de indexación (como intentar acceder a un elemento de un array a través de un índice demasiado grande o demasiado pequeño).

Las excepciones de tiempo de ejecución pueden ocurrir en cualquier lugar del programa y en un programa típico pueden ser muy numerosas. Típicamente, el coste del chequeo de las excepciones de tiempo de ejecución excede de los beneficios de capturarlas o especificarlas.

Así el compilador no requiere que se capturen o especifiquen las excepciones de tiempo de ejecución, pero se puede hacer.

Las excepciones chequeadas representan información útil sobre la operación legalmente especificada sobre la que el llamador podría no tener control y el llamador necesita estar

informado sobre ella -- por ejemplo, el sistema de ficheros está lleno, o el ordenador remoto ha cerrado la conexión, o los permisos de acceso no permiten esta acción.

¿Qué se consigue si se lanza una excepción RuntimeException o se crea una subclase de RuntimeException sólo porque no se quiere especificarla? Simplemente, se obtiene la posibilidad de lanzar una excepción sin especificar lo que se está haciendo. En otras palabras, es una forma de evitar la documentación de las excepciones que puede lanzar un método.

¿Cuándo es bueno esto? Bien, ¿cuándo es bueno evitar la documentación sobre el comportamiento de los métodos? La respuesta es "NUNCA".

### **Reglas del Pulgar:**

- Se puede detectar y lanzar una excepción de tiempo de ejecución cuando se encuentra un error en la máquina virtual, sin embargo, es más sencillo dejar que la máquina virtual lo detecte y lo lance. Normalmente, los métodos que escribas lanzarán excepciones del tipo Exception, no del tipo RuntimeException.
- De forma similar, puedes crear una subclase de RuntimeException cuando estas creando un error en la máquina virtual (que probablemente no lo hará), De otro modo utilizará la clase Exception.
- No lances una excepción en tiempo de ejecución o crees una subclase de RuntimeException simplemente porque no quieres preocuparte de especificarla.



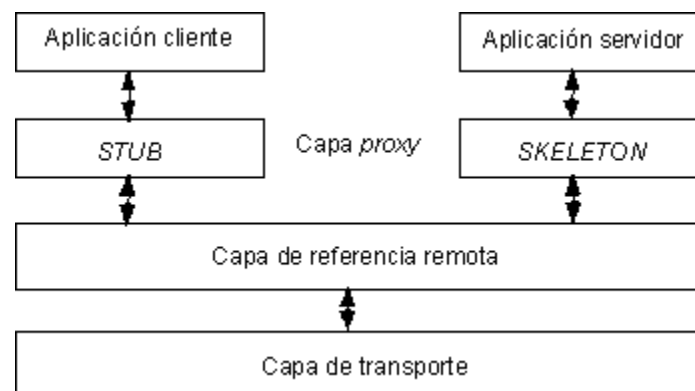
# Capítulo 3

## Remote Method Invocation

### INTRODUCCIÓN

Remote Method Invocation (RMI) es un mecanismo que permite realizar llamadas a métodos de objetos remotos situados en distintas (o la misma) máquinas virtuales de Java, compartiendo así recursos y carga de procesamiento a través de varios sistemas.

La arquitectura RMI puede verse como un modelo de cuatro capas:



La capa 1 es la aplicación y corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda *java.rmi.Remote*. Dicha interfaz se usa básicamente para "marcar" un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende la clase *UnicastRemoteObject* (paquete *java.rmi.server*), o puede hacerse de forma explícita con una llamada al método *exportObject()* del mismo paquete.

La capa 2 es la capa **proxy**, o capa stub-skeleton. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

La capa 3 es la de **referencia remota**, y es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la

recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo **stream** (**stream-oriented connection**) desde la capa de transporte.

La capa 4 es la de transporte. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP (*Java Remote Method Protocol*), que solamente es "comprendido" por programas Java.

Toda aplicación RMI normalmente se descompone en 2 partes:

- Un **servidor**, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
- Un **cliente**, que obtiene una referencia a objetos remotos en el servidor, y los invoca.

## CREAR UN SERVIDOR RMI

---

Un servidor RMI consiste en definir un objeto remoto que va a ser utilizado por los clientes. Para crear un objeto remoto, se define una interfaz, y el objeto remoto será una clase que implemente dicha interfaz. Veamos como crear un servidor de ejemplo mediante 3 pasos:

### 3. Definir el interfaz remoto

Implementar el interfaz remoto

### Definir el interfaz remoto

Cuando se crea un interfaz remoto:

- El interfaz debe ser público.
- Debe extender (heredar de) el interfaz *java.rmi.Remote*, para indicar que puede llamarse desde cualquier máquina virtual Java.
- Cada método remoto debe lanzar la excepción *java.rmi.RemoteException* en su cláusula throws, además de las excepciones que pueda manejar.

Veamos un ejemplo de interfaz remoto:

```
public interface MiInterfazRemoto extends java.rmi.Remote {  
  
    public void miMetodo1() throws java.rmi.RemoteException;  
  
}
```

```
public int miMetodo2() throws java.rmi.RemoteException;  
  
}
```

## Implementar el interfaz remoto

```
public class MiClaseRemota  
extends java.rmi.server.UnicastRemoteObject  
implements MiInterfazRemoto {  
  
    public MiClaseRemota() throws java.rmi.RemoteException{  
        // Código del constructor  
    }  
  
    public void miMetodo1() throws java.rmi.RemoteException{  
        // Aquí ponemos el código que queramos  
        System.out.println("Estoy en miMetodo1()");  
    }  
  
    public int miMetodo2() throws java.rmi.RemoteException{  
        return 5; // Aquí ponemos el código que queramos  
    }  
  
    public void otroMetodo() {  
        // Si definimos otro método, éste no podría llamarse  
        // remotamente al no ser del interfaz remoto  
    }  
  
    public static void main(String[] args){  
        try {  
            MiInterfazRemoto mir = new MiClaseRemota();  
            java.rmi.Naming.rebind("//" + java.net.InetAddress.getLocalHost().getHostAddress() +  
                ":" + args[0] + "/PruebaRMI", mir);  
        } catch (Exception e) {  
        }  
    }  
}
```

Como se puede observar, la clase *MiClaseRemota* implementa el interfaz *MiInterfazRemoto* que hemos definido previamente. Además, hereda de *UnicastRemoteObject*, que es una clase de Java que podemos utilizar como superclase para implementar objetos remotos.

Luego, dentro de la clase, definimos un constructor (que lanza la excepción `RemoteException` porque también la lanza la superclase `UnicastRemoteObject`), y los métodos de la/las interfaz/interfaces que implemente.

Finalmente, en el método `main`, definimos el código para crear el objeto remoto que se quiere compartir y hacer el objeto remoto visible para los clientes, mediante la clase `Naming` y su método `rebind(...)`.

**Nota:** Se ha puesto el método `main()` dentro de la misma clase por comodidad. Podría definirse otra clase aparte que fuera la encargada de registrar el objeto remoto.

## Compilar y ejecutar el servidor

Ya tenemos definido el servidor. Ahora tenemos que compilar sus clases mediante los siguientes pasos:

- Compilamos el interfaz remoto. Además lo agrupamos en un fichero JAR para tenerlo presente tanto en el cliente como en el servidor:

```
javac MiInterfazRemoto.java
jar cvf objRemotos.jar MiInterfazRemoto.class
```

- Luego, compilamos las clases que implementen los interfaces. Y para cada una de ellas generamos los ficheros Stub y Skeleton para mantener la referencia con el objeto remoto, mediante el comando `rmic`:

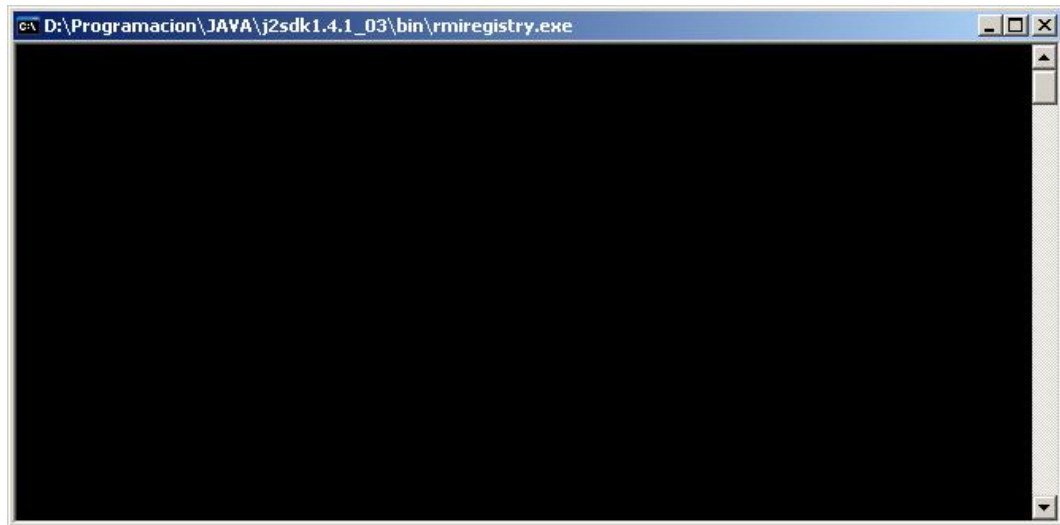
```
set CLASSPATH=%CLASSPATH%;.objRemotos.jar;.
javac MiClaseRemota.java
rmic -d . MiClaseRemota
```

Observamos en nuestro directorio de trabajo que se han generado automáticamente dos ficheros `.class` (`MiClaseRemota_Skel.class` y `MiClaseRemota_Stub.class`) correspondientes a la capa stub-skeleton de la arquitectura RMI.

Para ejecutar el servidor, seguimos los siguientes pasos:

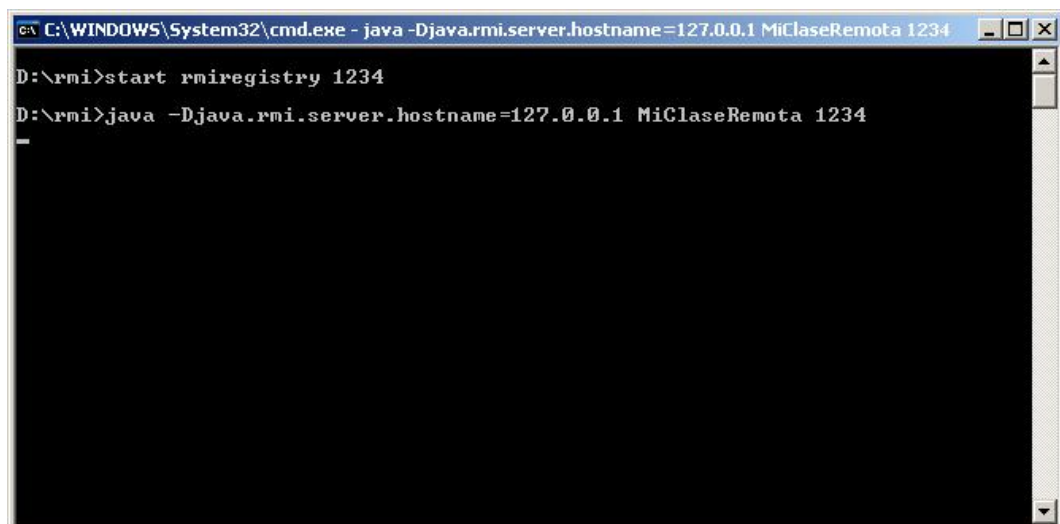
- Se arranca el **registro de RMI** para permitir registrar y buscar objetos remotos. El registro se encarga de gestionar un conjunto de objetos remotos a compartir, y buscarlos ante las peticiones de los clientes. Se ejecuta con la aplicación **rmiregistry** distribuida con Java, a la que podemos pasarle opcionalmente el puerto por el que conectar (por defecto, el 1099):

```
start rmiregistry 1234
```



- Por último, se lanza el servidor:

```
java -Djava.rmi.server.hostname=127.0.0.1 MiClaseRemota 1234
```



## CREAR UN CLIENTE RMI

Vamos ahora a definir un cliente que accederá a el/los objeto/s remoto/s que creemos. Para ello seguimos los siguientes pasos:

### Definir la clase para obtener los objetos remotos necesarios

La siguiente clase obtiene un objeto de tipo `MiInterfazRemoto`, implementado en nuestro servidor:

```
public class MiClienteRMI {
```

```
public static void main(String[] args) {  
    try {  
        MiInterfazRemoto mir = (MiInterfazRemoto)java.rmi.Naming.lookup("//" +  
            args[0] + ":" + args[1] + "/PruebaRMI");  
        // Imprimimos miMetodo1() tantas veces como devuelva miMetodo2()  
        for (int i=1;i<=mir.miMetodo2();i++)  
            mir.miMetodo1();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Como se puede observar, simplemente consiste en buscar el objeto remoto en el registro RMI de la máquina remota. Para ello usamos la clase *Naming* y su método *lookup(...)*.

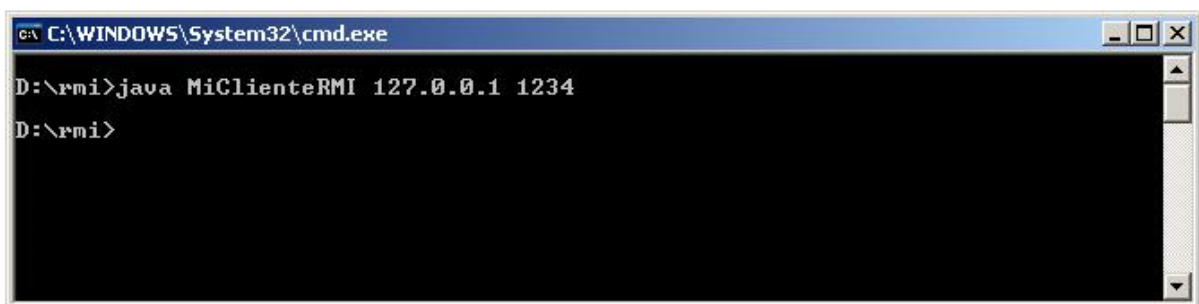
## Compilar y ejecutar el cliente

Una vez que ya tenemos definido el cliente, para compilarlo hacemos:

```
set CLASSPATH=%CLASSPATH%;.\objRemotos.jar;.  
javac MiClienteRMI.java
```

Luego, para ejecutar el cliente hacemos:

```
java MiClienteRMI 127.0.0.1 1234
```



Se debe poder acceder al fichero **Stub** de la clase remota. Para ello, o bien lo copiamos al cliente y lo incluimos en su **CLASSPATH**, o lo eliminamos del **CLASSPATH** del servidor e incluimos su ruta en el **java.rmi.codebase** del servidor (si no se elimina del **CLASSPATH** del servidor, se ignorará la opción **java.rmi.codebase**, y el cliente no podrá acceder al **Stub**).

Si echamos un vistazo a la ventana donde está ejecutándose el servidor RMI, veremos como se ha encontrado el objeto remoto y ejecutado sus métodos:



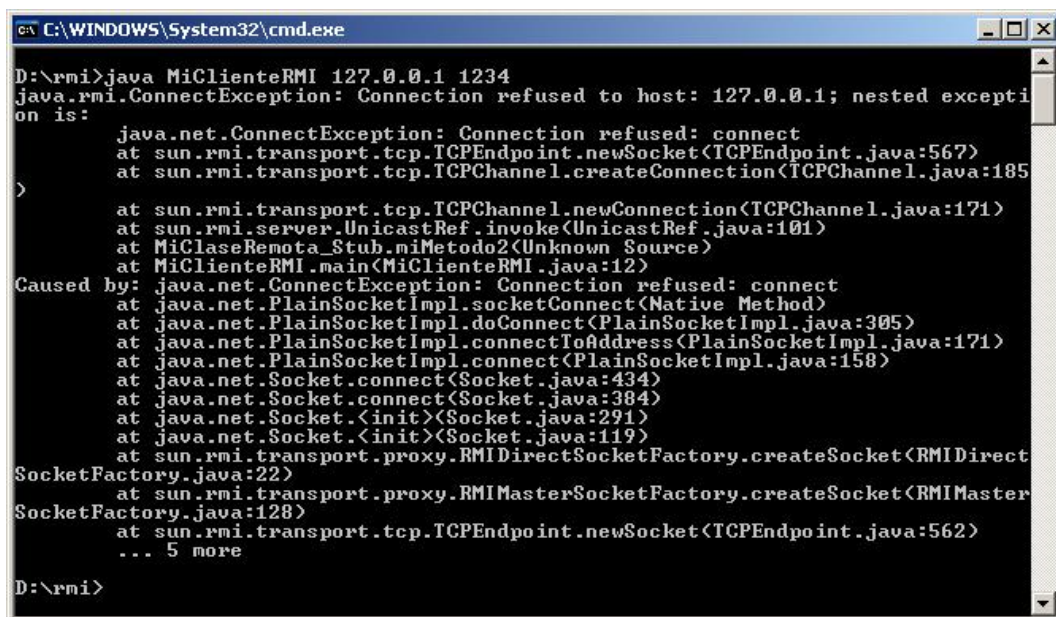
```
C:\WINDOWS\System32\cmd.exe - java -Djava.rmi.server.hostname=127.0.0.1 MiClaseRemota 1234

D:\rmi>start rmiregistry 1234

D:\rmi>java -Djava.rmi.server.hostname=127.0.0.1 MiClaseRemota 1234
Estoy en miMetodo1()
Estoy en miMetodo1()
Estoy en miMetodo1()
Estoy en miMetodo1()
Estoy en miMetodo1()
-
```

## RESUMEN

En este ejemplo sencillo hemos visto como utilizar la tecnología RMI para trabajar con objetos remotos que nos van a permitir crear aplicaciones distribuidas. Para ello, hemos registrado el objeto servidor mediante el servicio de nombres RMI (naming/registry service) desde una máquina virtual de Java en ejecución, de forma que si la máquina virtual Java termina, el objeto deja de existir y provocará una excepción al invocarlo:



```
C:\WINDOWS\System32\cmd.exe

D:\rmi>java MiClienteRMI 127.0.0.1 1234
java.rmi.ConnectException: Connection refused to host: 127.0.0.1; nested excepti
on is:
    java.net.ConnectException: Connection refused: connect
    at sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:567)
    at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:185)
    at sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:171)
    at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:101)
    at MiClaseRemota_Stub.miMetodo2(Unknown Source)
    at MiClienteRMI.main(MiClienteRMI.java:12)
Caused by: java.net.ConnectException: Connection refused: connect
    at java.net.PlainSocketImpl.socketConnect(Native Method)
    at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:305)
    at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:171)
    at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:158)
    at java.net.Socket.connect(Socket.java:434)
    at java.net.Socket.connect(Socket.java:384)
    at java.net.Socket.<init>(Socket.java:291)
    at java.net.Socket.<init>(Socket.java:119)
    at sun.rmi.transport.proxy.RMIDirectSocketFactory.createSocket(RMIDirect
SocketFactory.java:22)
    at sun.rmi.transport.proxy.RMIMasterSocketFactory.createSocket(RMIMaster
SocketFactory.java:128)
    at sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:562)
    ... 5 more

D:\rmi>
```

# Capítulo 4

## Sockets

### FUNDAMENTOS

---

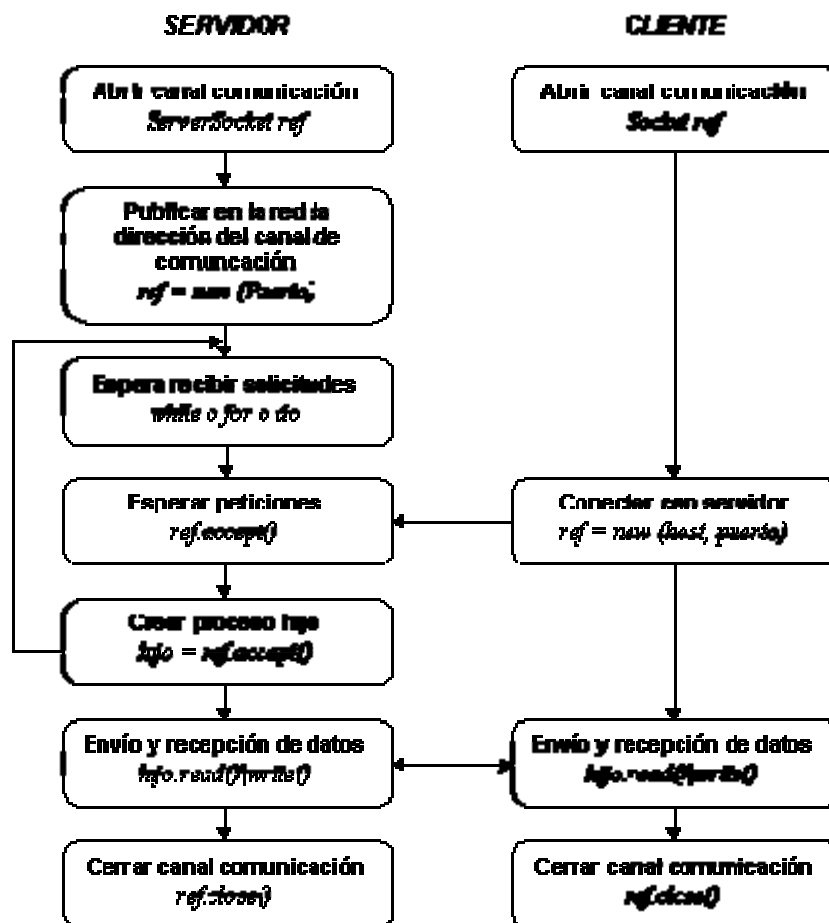
Los sockets son un sistema de comunicación entre procesos de diferentes máquinas de una red. Más exactamente, un socket es un punto de comunicación por el cual un proceso puede emitir o recibir información.

Fueron popularizados por Berkeley Software Distribution, de la universidad norteamericana de Berkeley. Los sockets han de ser capaces de utilizar el protocolo de streams TCP (Transfer Control Protocol) y el de datagramas UDP (User Datagram Protocol).

Utilizan una serie de primitivas para establecer el punto de comunicación, para conectarse a una máquina remota en un determinado puerto que esté disponible, para escuchar en él, para leer o escribir y publicar información en él, y finalmente para desconectarse.

Con todas las primitivas se puede crear un sistema de diálogo muy completo.



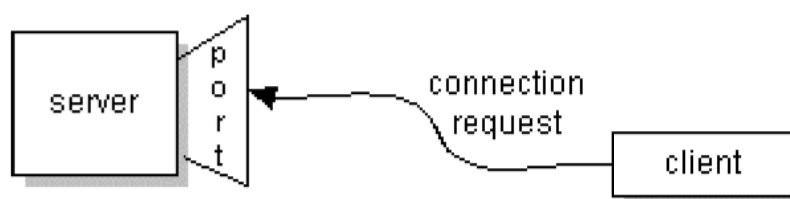


## FUNCIONAMIENTO GENÉRICO

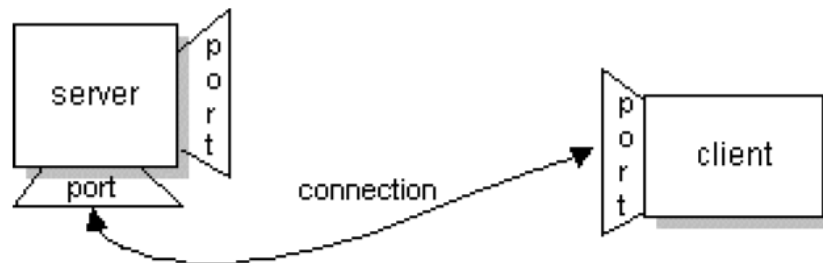
Normalmente, un servidor se ejecuta sobre una computadora específica y tiene un socket que responde en un puerto específico. El servidor únicamente espera, escuchando a través del socket a que un cliente haga una petición.

En el lado del cliente: el cliente conoce el nombre de host de la máquina en la cual el servidor se encuentra ejecutando y el número de puerto en el cual el servidor está conectado.

Para realizar una petición de conexión, el cliente intenta encontrar al servidor en la máquina servidora en el puerto especificado.



Si todo va bien, el servidor acepta la conexión. Además de aceptar, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que necesita un nuevo socket (y, en consecuencia, un número de puerto diferente) para seguir atendiendo al socket original para peticiones de conexión mientras atiende las necesidades del cliente que se conectó.



Por la parte del cliente, si la conexión es aceptada, un socket se crea de forma satisfactoria y puede usarlo para comunicarse con el servidor. Es importante darse cuenta que el socket en el cliente no está utilizando el número de puerto usado para realizar la petición al servidor. En lugar de éste, el cliente asigna un número de puerto local a la máquina en la cual está siendo ejecutado. Ahora el cliente y el servidor pueden comunicarse escribiendo o leyendo en o desde sus respectivos sockets.

## JAVA SOCKETS

---

### Introduccion

El paquete `java.net` de la plataforma Java proporciona una clase `Socket`, la cual implementa una de las partes de la comunicación bidireccional entre un programa Java y otro programa en la red.

La clase `Socket` se sitúa en la parte más alta de una implementación dependiente de la plataforma, ocultando los detalles de cualquier sistema particular al programa Java. Usando la clase `java.net.Socket` en lugar de utilizar código nativo de la plataforma, los programas Java pueden comunicarse a través de la red de una forma totalmente independiente de la plataforma.

De forma adicional, `java.net` incluye la clase `ServerSocket`, la cual implementa un socket el cual los servidores pueden utilizar para escuchar y aceptar peticiones de conexión de clientes.

Nuestro objetivo será conocer cómo utilizar las clases `Socket` y `ServerSocket`.

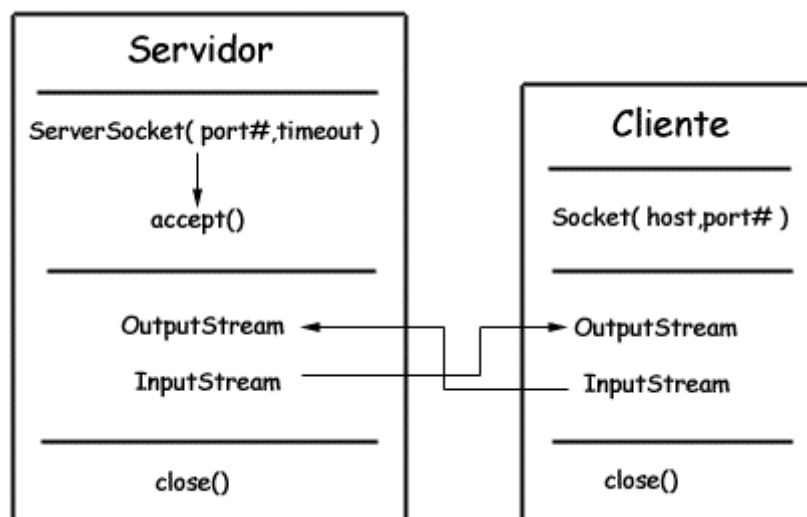
Por otra parte, si intentamos conectar a través de la Web, la clase `URL` y clases relacionadas (`URLConnection`, `URLEncoder`) son probablemente más apropiadas que las

clases de sockets. Pero de hecho , las clases URL no son más que una conexión a un nivel más alto a la Web y utilizan como parte de su implementación interna los sockets.

## Modelo de comunicaciones con Java

El modelo de sockets más simple es:

- El servidor establece un puerto y espera durante un cierto tiempo (timeout segundos), a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método `accept()`.
- El cliente establece una conexión con la máquina host a través del puerto que se designe en `puerto#`
- El cliente y el servidor se comunican con manejadores **InputStream** y **OutputStream**



## Apertura de Sockets

Si estamos programando un CLIENTE, el socket se abre de la forma:

```
Socket miCliente;  
miCliente = new Socket( "maquina", numeroPuerto );
```

Donde `maquina` es el nombre de la máquina en donde estamos intentando abrir la conexión y `numeroPuerto` es el puerto (un número) del servidor que está corriendo sobre el cual nos queremos conectar. Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios (superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema

como email, ftp o http. Para las aplicaciones que se desarrollen, asegurarse de seleccionar un puerto por encima del 1023.

En el ejemplo anterior no se usan excepciones; sin embargo, es una gran idea la captura de excepciones cuando se está trabajando con sockets. El mismo ejemplo quedaría como:

```
Socket miCliente;  
try {  
    miCliente = new Socket( "maquina",numeroPuerto );  
} catch( IOException e ) {  
    System.out.println( e );  
}
```

Si estamos programando un SERVIDOR, la forma de apertura del socket es la que muestra el siguiente ejemplo:

```
Socket miServicio;  
try {  
    miServicio = new ServerSocket( numeroPuerto );  
} catch( IOException e ) {  
    System.out.println( e );  
}
```

A la hora de la implementación de un servidor también necesitamos crear un objeto socket desde el ServerSocket para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
Socket socketServicio = null;  
try {  
    socketServicio = miServicio.accept();  
} catch( IOException e ) {  
    System.out.println( e );  
}
```

## Creación de Streams

### Creación de Streams de Entrada

En la parte CLIENTE de la aplicación, se puede utilizar la clase `DataInputStream` para crear un stream de entrada que esté listo a recibir todas las respuestas que el servidor le envíe.

```
DataInputStream entrada;  
try {  
    entrada = new DataInputStream( miCliente.getInputStream() );  
} catch( IOException e ) {
```

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

La clase `DataInputStream` permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: `read()`, `readChar()`, `readInt()`, `readDouble()` y `readLine()`. Debemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperemos recibir del servidor.

En el lado del SERVIDOR, también usaremos `DataInputStream`, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado:

```
InputStream entrada;  
try {  
    entrada =  
        new DataInputStream( socketServicio.getInputStream() );  
} catch( IOException e ) {  
    System.out.println( e );  
}
```

## Creación de Streams de Salida

En el lado del CLIENTE, podemos crear un stream de salida para enviar información al socket del servidor utilizando las clases `PrintStream` o `DataOutputStream`:

```
PrintStream salida;  
try {  
    salida = new PrintStream( miCliente.getOutputStream() );  
} catch( IOException e ) {  
    System.out.println( e );  
}
```

La clase `PrintStream` tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos `write` y `println()` tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también podemos utilizar `DataOutputStream`:

```
DataOutputStream salida;  
try {  
    salida = new DataOutputStream( miCliente.getOutputStream() );  
} catch( IOException e ) {  
    System.out.println( e );  
}
```

La clase `DataOutputStream` permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el stream de salida. De todos esos métodos, el más útil quizás sea `writeBytes()`.

En el lado del SERVIDOR, podemos utilizar la clase `PrintStream` para enviar información al cliente:

```
PrintStream salida;
try {
    salida = new PrintStream( socketServicio.getOutputStream() );
} catch( IOException e ) {
    System.out.println( e );
}
```

Pero también podemos utilizar la clase `DataOutputStream` como en el caso de envío de información desde el cliente.

## Cierre de Sockets

Siempre deberemos cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la parte del cliente:

```
try {
    salida.close();
    entrada.close();
    miCliente.close();
} catch( IOException e ) {
    System.out.println( e );
}
```

Y en la parte del servidor:

```
try {
    salida.close();
    entrada.close();
    socketServicio.close();
    miServicio.close();
} catch( IOException e ) {
    System.out.println( e );
}
```

Es importante destacar que el orden de cierre es relevante. Es decir, se deben cerrar primero los streams relacionados con un socket antes que el propio socket, ya que de esta forma evitamos posibles errores de escrituras o lecturas sobre descriptores ya cerrados.

## Clases útiles en comunicaciones

### ➤ **Socket**

Es el objeto básico en toda comunicación a través de Internet, bajo el protocolo TCP. Esta clase proporciona métodos para la entrada/salida a través de streams que hacen la lectura y escritura a través de sockets muy sencilla.

### ➤ **ServerSocket**

Es un objeto utilizado en las aplicaciones servidor para escuchar las peticiones que realicen los clientes conectados a ese servidor. Este objeto no realiza el servicio, sino que crea un objeto Socket en función del cliente para realizar toda la comunicación a través de él.

### ➤ **DatagramSocket**

La clase de sockets datagrama puede ser utilizada para implementar datagramas no fiables (sockets UDP), no ordenados. Aunque la comunicación por estos sockets es muy rápida porque no hay que perder tiempo estableciendo la conexión entre cliente y servidor.

### ➤ **DatagramPacket**

Clase que representa un paquete datagrama conteniendo información de paquete, longitud de paquete, direcciones Internet y números de puerto.

### ➤ **MulticastSocket**

Clase utilizada para crear una versión multicast de las clase socket datagrama. Múltiples clientes/servidores pueden transmitir a un grupo multicast (un grupo de direcciones IP compartiendo el mismo número de puerto).

### ➤ **NetworkServer**

Una clase creada para implementar métodos y variables utilizadas en la creación de un servidor TCP/IP.

### ➤ **NetworkClient**

Una clase creada para implementar métodos y variables utilizadas en la creación de un cliente TCP/IP.

### ➤ **SocketImpl**

Es un Interface que nos permite crearnos nuestro propio modelo de comunicación. Tendremos que implementar sus métodos cuando la usemos. Si vamos a desarrollar una aplicación con requerimientos especiales de comunicaciones, como pueden ser la implementación de un cortafuegos (TCP es un protocolo no seguro), o acceder a equipos especiales (como un lector de código de barras o un GPS diferencial), necesitaremos nuestra propia clase Socket.

## Ejemplo de uso

Para comprender el funcionamiento de los sockets no hay nada mejor que estudiar un ejemplo. El que a continuación se presenta establece un pequeño diálogo entre un programa servidor y sus clientes, que intercambiarán cadenas de información.

### Programa Cliente

El programa cliente se conecta a un servidor indicando el nombre de la máquina y el número puerto (tipo de servicio que solicita) en el que el servidor está instalado.

Una vez conectado, lee una cadena del servidor y la escribe en la pantalla:

```
import java.io.*;
import java.net.*;

class Cliente {

    static final String HOST = "localhost";
    static final int PUERTO=5000;
    public Cliente( ) {
        try{
            Socket skCliente = new Socket( HOST , Puerto );
            InputStream aux = skCliente.getInputStream();
            DataInputStream flujo = new DataInputStream( aux );
            System.out.println( flujo.readUTF() );
            skCliente.close();
        } catch( Exception e ) {
            System.out.println( e.getMessage() );
        }
    }

    public static void main( String[] arg ) {
        new Cliente();
    }
}
```



En primer lugar se crea el socket denominado skCliente, al que se le especifican el nombre de host (HOST) y el número de puerto (PORT) en este ejemplo son constantes.

Luego se asocia el flujo de datos de dicho socket (obtenido mediante `getInputStream()`), que es asociado a un flujo (flujo) `DataInputStream` de lectura secuencial. De dicho flujo capturamos una cadena ( `readUTF()` ), y la imprimimos por pantalla (`System.out`).

El socket se cierra, una vez finalizadas las operaciones, mediante el método `close()`.

Debe observarse que se realiza una gestión de excepción para capturar los posibles fallos tanto de los flujos de datos como del socket.

## Programa Servidor

El programa servidor se instala en un puerto determinado, a la espera de conexiones, a las que tratará mediante un segundo socket.

Cada vez que se presenta un cliente, le saluda con una frase "Hola cliente N".

Este servidor sólo atenderá hasta tres clientes, y después finalizará su ejecución, pero es habitual utilizar bucles infinitos ( `while(true)` ) en los servidores, para que atiendan llamadas continuamente.

Tras atender cuatro clientes, el servidor deja de ofrecer su servicio:

```
import java.io.* ;
import java.net.* ;

class Servidor {

    static final int PUERTO=5000;

    public Servidor() {
        try {
            ServerSocket skServidor = new ServerSocket(PUERTO);
            System.out.println("Escucho el puerto " + PUERTO );
            for ( int numCli = 0; numCli < 3; numCli++; ) {
                Socket skCliente = skServidor.accept(); // Crea objeto
                System.out.println("Sirvo al cliente " + numCli);
                OutputStream aux = skCliente.getOutputStream();
                DataOutputStream flujo= new DataOutputStream( aux );
                flujo.writeUTF( "Hola cliente " + numCli );
                skCliente.close();
            }
            System.out.println("Demasiados clientes por hoy");
        }
    }
}
```

```
        } catch( Exception e ) {  
            System.out.println( e.getMessage() );  
        }  
    }  
  
    public static void main( String[] arg ) {  
        new Servidor();  
    }  
}
```

Utiliza un objeto de la clase `ServerSocket` (`skServidor`), que sirve para esperar las conexiones en un puerto determinado (PUERTO), y un objeto de la clase `Socket` (`skCliente`) que sirve para gestionar una conexión con cada cliente.

Mediante un bucle `for` y la variable `numCli` se restringe el número de clientes a tres, con lo que cada vez que en el puerto de este servidor aparezca un cliente, se atiende y se incrementa el contador.

Para atender a los clientes se utiliza la primitiva `accept()` de la clase `ServerSocket`, que es una rutina que crea un nuevo `Socket` (`skCliente`) para atender a un cliente que se ha conectado a ese servidor.

Se asocia al socket creado (`skCliente`) un flujo (flujo) de salida `DataOutputStream` de escritura secuencial, en el que se escribe el mensaje a enviar al cliente.

El tratamiento de las excepciones es muy reducido en nuestro ejemplo, tan solo se captura e imprime el mensaje que incluye la excepción mediante `getMessage()`.

## Ejecución

Aunque la ejecución de los sockets está diseñada para trabajar con ordenadores en red, en sistemas operativos multitarea (por ejemplo Windows y UNIX) se puede probar el correcto funcionamiento de un programa de sockets en una misma máquina.

Para ellos se ha de colocar el servidor en una ventana, obteniendo lo siguiente:

```
>java Servidor  
Escucho el puerto 5000
```

En otra ventana se lanza varias veces el programa cliente, obteniendo:

```
>java Cliente  
Hola cliente 1  
>java cliente
```

```
Hola cliente 2
>java cliente
Hola cliente 3
>java cliente
connection refused: no further information
```

Mientras tanto en la ventana del servidor se ha impreso:

```
Sirvo al cliente 1
Sirvo al cliente 2
Sirvo al cliente 3
Demasiados clientes por hoy
```

Cuando se lanza el cuarto de cliente, el servidor ya ha cortado la conexión, con lo que se lanza una excepción.

Obsérvese que tanto el cliente como el servidor pueden leer o escribir del socket. Los mecanismos de comunicación pueden ser refinados cambiando la implementación de los sockets, mediante la utilización de las clases abstractas que el paquete java.net provee.

# Capítulo 5

## Java DataDase Connectivity

### INTRODUCCIÓN

---

JDBC (Java DataBase Connectivity) es un API de Java que permite al programador ejecutar instrucciones en lenguaje estándar de acceso a Bases de Datos, SQL (Structured Query Language, lenguaje estructurado de consultas), que es un lenguaje de muy alto nivel que permite crear, examinar, manipular y gestionar Bases de Datos relacionales. Para que una aplicación pueda hacer operaciones en una Base de Datos, ha de tener una conexión con ella, que se establece a través de un driver, que convierte el lenguaje de alto nivel a sentencias de Base de Datos. Es decir, las tres acciones principales que realizará JDBC son las de establecer la conexión a una base de datos, ya sea remota o no; enviar sentencias SQL a esa base de datos y, en tercer lugar, procesar los resultados obtenidos de la base de datos.

### EMPEZAR CON JDBC

---

Lo primero que tenemos que hacer es asegurarnos de que disponemos de la configuración apropiada. Esto incluye los siguientes pasos.

1. Instalar Java y el API JDBC en nuestra máquina.

Para instalar tanto la plataforma JAVA como el API JDBC, simplemente tenemos que seguir las instrucciones de descarga de la última versión del JDK (Java Development Kit). Junto con el JDK también viene el API JDBC. El código de ejemplo de demostración del API del JDBC 1.0 fue escrito para el JDK 1.1 y se ejecutará en cualquier versión de la plataforma Java compatible con el JDK 1.1, incluyendo el JDK1.2. Teniendo en cuenta que los ejemplos del API del JDBC 2.0 requieren el JDK 1.2 y no se podrán ejecutar sobre el JDK 1.1.

Podrás encontrar la última versión del JDK en la siguiente dirección:  
<http://java.sun.com/products/JDK/CurrentRelease>

2. Instalar un driver en nuestra máquina.

Nuestro Driver debe incluir instrucciones para su instalación. Para los drivers JDBC escritos para controladores de bases de datos específicos la instalación consiste sólo en copiar el driver en nuestra máquina; no se necesita ninguna configuración especial.

El driver "puente JDBC-ODBC" no es tan sencillo de configurar. Si descargamos las versiones Solaris o Windows de JDK 1.1, automáticamente obtendremos una versión del driver Bridge JDBC-ODBC, que tampoco requiere una configuración especial. Si embargo, ODBC, si lo necesita. Si no tenemos ODBC en nuestra máquina, necesitaremos preguntarle al vendedor del driver ODBC sobre su instalación y configuración.

3. Instalar nuestro Controlador de Base de Datos si es necesario.

Si no tenemos instalado un controlador de base de datos, necesitaremos seguir las instrucciones de instalación del vendedor. La mayoría de los usuarios tienen un controlador de base de datos instalado y trabajarán con un base de datos establecida.

## SELECCIONAR UNA BASE DE DATOS

---

A lo largo de la sección asumiremos que la base de datos COFFEEBREAK ya existe. (crear una base de datos no es nada difícil, pero requiere permisos especiales y normalmente lo hace un administrador de bases de datos). Cuando creamos las tablas utilizadas como ejemplos en este manual, serán la base de datos por defecto. Hemos mantenido un número pequeño de tablas para mantener las cosas manejables.

Supongamos que nuestra base de datos está siendo utilizada por el propietario de un pequeño café llamado "The Coffee Break", donde los granos de café se venden por kilos y el café líquido se vende por tazas. Para mantener las cosas sencillas, también supondremos que el propietario sólo necesita dos tablas, una para los tipos de café y otra para los suministradores.

Primero veremos como abrir una conexión con nuestro controlador de base de datos, y luego, ya que JDBC puede enviar código SQL a nuestro controlador, demostraremos algún código SQL. Después, veremos lo sencillo que es utilizar JDBC para pasar esas sentencias SQL a nuestro controlador de bases de datos y procesar los resultados devueltos.

Este código ha sido probado en la mayoría de los controladores de base de datos. Sin embargo, podríamos encontrar algunos problemas de compatibilidad si utilizamos antiguos drivers ODB con el puente JDBC.ODBC.

## ESTABLECER UNA CONEXIÓN

---

Lo primero que tenemos que hacer es establecer una conexión con el controlador de base de datos que queremos utilizar. Esto implica dos pasos: (1) cargar el driver y (2) hacer la conexión.

## Cargar los Drivers

Cargar el driver o drivers que queremos utilizar es muy sencillo y sólo implica una línea de código. Si, por ejemplo, queremos utilizar el puente JDBC-ODBC, se cargaría la siguiente línea de código.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

La documentación del driver nos dará el nombre de la clase a utilizar. Por ejemplo, si el nombre de la clase es jdbc.DriverXYZ, cargaríamos el driver con esta línea de código.

```
Class.forName("jdbc.DriverXYZ");
```

No necesitamos crear un ejemplar de un driver y registrarlo con el DriverManager porque la llamada a Class.forName lo hace automáticamente. Si hubiéramos creado nuestro propio ejemplar, crearíamos un duplicado innecesario, pero no pasaría nada.

Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos.

## Hacer la Conexión

El segundo paso para establecer una conexión es tener el driver apropiado conectado al controlador de base de datos. La siguiente línea de código ilustra la idea general.

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

Este paso también es sencillo, lo más duro es saber qué suministrar para url. Si estamos utilizando el puente JDBC-ODBC, el JDBC URL empezará con jdbc:odbc:. el resto de la URL normalmente es la fuente de nuestros datos o el sistema de base de datos. Por eso, si estamos utilizando ODBC para acceder a una fuente de datos ODBC llamada "Fred," por ejemplo, nuestro URL podría ser jdbc:odbc:Fred. En lugar de "myLogin" pondríamos el nombre utilizado para entrar en el controlador de la base de datos; en lugar de "myPassword" pondríamos nuestra password para el controlador de la base de datos. Por eso si entramos en el controlador con el nombre "Fernando" y la password of "J8," estas dos líneas de código establecerán una conexión.

```
String url = "jdbc:odbc:Fred";  
Connection con = DriverManager.getConnection(url, "Fernando", "J8");
```

Si estamos utilizando un puente JDBC desarrollado por una tercera parte, la documentación nos dirá el subprotocolo a utilizar, es decir, qué poner después de jdbc: en la URL. Por ejemplo, si el desarrollador ha registrado el nombre "acme" como el subprotocolo, la primera y segunda parte de la URL de JDBC serán jdbc:acme:. La documentación del driver también

nos dará las guías para el resto de la URL del JDBC. Esta última parte de la URL suministra información para la identificación de los datos fuente.

Si uno de los drivers que hemos cargado reconoce la URL suministrada por el método `DriverManager.getConnection`, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase `DriverManager`, como su nombre indica, maneja todos los detalles del establecimiento de la conexión detrás de la escena. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface `Driver`, y el único método de `DriverManager` que realmente necesitaremos conocer es `DriverManager.getConnection`.

La conexión devuelta por el método `DriverManager.getConnection` es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, con es una conexión abierta, y se utilizará en los ejemplos posteriores.

## SELECCIONAR UNA TABLA

Primero, crearemos una de las tablas de nuestro ejemplo. Esta tabla, `COFFEES`, contiene la información esencial sobre los cafés vendidos en "The Coffee Break", incluyendo los nombres de los cafés, sus precios, el número de libras vendidas la semana actual, y el número de libras vendidas hasta la fecha. Aquí puedes ver la tabla `COFFEES`, que describiremos más adelante.

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

La columna que almacena el nombre del café es `COF_NAME`, y contiene valores con el tipo `VARCHAR` de SQL y una longitud máxima de 32 caracteres. Como utilizamos nombres diferentes para cada tipo de café vendido, el nombre será un único identificador para un café particular y por lo tanto puede servir como clave primaria. La segunda columna, llamada `SUP_ID`, contiene un número que identifica al suministrador del café; este número será un

tipo INTEGER de SQL. La tercera columna, llamada PRICE, almacena valores del tipo FLOAT de SQL porque necesita contener valores decimales. (Observa que el dinero normalmente se almacena en un tipo DECIMAL o NUMERIC de SQL, pero debido a las diferencias entre controladores de bases de datos y para evitar la incompatibilidad con viejas versiones de JDBC, utilizamos el tipo más estándar FLOAT.) La columna llamada SALES almacena valores del tipo INTEGER de SQL e indica el número de libras vendidas durante la semana actual. La columna final, TOTAL, contiene otro valor INTEGER de SQL que contiene el número total de libras vendidas hasta la fecha.

SUPPLIERS, la segunda tabla de nuestra base de datos, tiene información sobre cada uno de los suministradores.

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Las tablas COFFEES y SUPPLIERS contienen la columna SUP\_ID, lo que significa que estas dos tablas pueden utilizarse en sentencias SELECT para obtener datos basados en la información de ambas tablas. La columna SUP\_ID es la clave primaria de la tabla SUPPLIERS, y por lo tanto, es un identificador único para cada uno de los suministradores de café. En la tabla COFFEES, SUP\_ID es llamada clave extranjera. (Se puede pensar en una clave extranjera en el sentido en que es importada desde otra tabla). Observa que cada número SUP\_ID aparece sólo una vez en la tabla SUPPLIERS; esto es necesario para ser una clave primaria. Sin embargo, en la tabla COFFEES, donde es una clave extranjera, es perfectamente correcto que haya números duplicados de SUP\_ID porque un suministrador puede vender varios tipos de café. Más adelante en este capítulo podremos ver cómo utilizar claves primarias y extranjeras en una sentencia SELECT.

La siguiente sentencia SQL crea la tabla COFFEES. Las entradas dentro de los paréntesis exteriores consisten en el nombre de una columna seguido por un espacio y el tipo SQL que se va a almacenar en esa columna. Una coma separa la entrada de una columna (que consiste en el nombre de la columna y el tipo SQL) de otra. El tipo VARCHAR se crea con una longitud máxima, por eso toma un parámetro que indica la longitud máxima. El parámetro debe estar entre paréntesis siguiendo al tipo. La sentencia SQL mostrada aquí, por ejemplo, especifica que los nombres de la columna COF-NAME pueden tener hasta 32 caracteres de longitud.

```
CREATE TABLE COFFEES(
```



```
COF_NAME VARCHAR(32),  
SUP_ID INTEGER,  
PRICE FLOAT,  
SALES INTEGER,  
TOTAL INTEGER  
);
```

Este código no termina con un terminador de sentencia de un controlador de base de datos, que puede variar de un controlador a otro. Por ejemplo, Oracle utiliza un punto y coma (;) para finalizar una sentencia, y Sybase utiliza la palabra go. El driver que estamos utilizando proporcionará automáticamente el terminador de sentencia apropiado, y no necesitaremos introducirlo en nuestro código JDBC.

Otra cosa que debíamos apuntar sobre las sentencias SQL es su forma. En la sentencia CREATE TABLE, las palabras clave se han impreso en letras mayúsculas, y cada ítem en una línea separada. SQL no requiere nada de esto, estas convenciones son sólo para una fácil lectura. El estándar SQL dice que las palabras claves no son sensibles a las mayúsculas, así, por ejemplo, la anterior sentencia SELECT puede escribirse de varias formas. Y como ejemplo, estas dos versiones son equivalentes en lo que concierne a SQL.

```
SELECT First_Name, Last_Name  
FROM Employees  
WHERE Last_Name LIKE "Washington"
```

```
select First_Name, Last_Name from Employees where  
Last_Name like "Washington"
```

Sin embargo, el material entre comillas sí es sensible a las mayúsculas: en el nombre "Washington", "W" debe estar en mayúscula y el resto de las letras en minúscula.

Los requerimientos pueden variar de un controlador de base de datos a otro cuando se trata de nombres de identificadores. Por ejemplo, algunos controladores, requieren que los nombres de columna y de tabla sean exactamente los mismos que se crearon en las sentencias CREATE y TABLE, mientras que otros controladores no lo necesitan. Para asegurarnos, utilizaremos mayúsculas para identificadores como COFFEES y SUPPLIERS porque así es como los definimos.

Hasta ahora hemos escrito la sentencia SQL que crea la tabla COFFEES. Ahora le pondremos comillas (crearemos un string) y asignaremos el string a la variable createTableCoffees para poder utilizarla en nuestro código JDBC más adelante. Como hemos visto, al controlador de base de datos no le importa si las líneas están divididas, pero en el lenguaje Java, un objeto String que se extienda más allá de una línea no será

compilado. Consecuentemente, cuando estamos entregando cadenas, necesitamos encerrar cada línea entre comillas y utilizar el signo más (+) para concatenarlas.

```
String createTableCoffees = "CREATE TABLE COFFEES " +  
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +  
    "SALES INTEGER, TOTAL INTEGER)";
```

Los tipos de datos que hemos utilizado en nuestras sentencias CREATE y TABLE son tipos genéricos SQL (también llamados tipos JDBC) que están definidos en la clase `java.sql.Types`. Los controladores de bases de datos generalmente utilizan estos tipos estándares, por eso cuando llegue el momento de probar alguna aplicación, sólo podremos utilizar la aplicación `CreateCoffees.java`, que utiliza las sentencias CREATE y TABLE. Si tu controlador utiliza sus propios nombres de tipos, te suministraremos más adelante una aplicación que hace eso.

Sin embargo, antes de ejecutar alguna aplicación, veremos lo más básico sobre el JDBC.

## Crear sentencias JDBC

Un objeto `Statement` es el que envía nuestras sentencias SQL al controlador de la base de datos. Simplemente creamos un objeto `Statement` y lo ejecutamos, suministrando el método SQL apropiado con la sentencia SQL que queremos enviar. Para una sentencia SELECT, el método a ejecutar es `executeQuery`. Para sentencias que crean o modifican tablas, el método a utilizar es `executeUpdate`.

Se toma un ejemplar de una conexión activa para crear un objeto `Statement`. En el siguiente ejemplo, utilizamos nuestro objeto `Connection`: con para crear el objeto `Statement`: `stmt`.

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

En este momento `stmt` existe, pero no tiene ninguna sentencia SQL que pasarle al controlador de la base de datos. Necesitamos suministrarle el método que utilizaremos para ejecutar `stmt`. Por ejemplo, en el siguiente fragmento de código, suministramos `executeUpdate` con la sentencia SQL del ejemplo anterior.

```
stmt.executeUpdate("CREATE TABLE COFFEES " +  
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +  
    "SALES INTEGER, TOTAL INTEGER)");
```

Como ya habíamos creado un `String` con la sentencia SQL y lo habíamos llamado `createTableCoffees`, podríamos haber escrito el código de esta forma alternativa.

```
stmt.executeUpdate(createTableCoffees);
```

## Ejecutar Sentencias

Utilizamos el método `executeUpdate` porque la sentencia SQL contenida en `createTableCoffees` es una sentencia DDL (data definition language). Las sentencias que crean, modifican o eliminan tablas son todos ejemplos de sentencias DDL y se ejecutan con el método `executeUpdate`. Como se podría esperar de su nombre, el método `executeUpdate` también se utiliza para ejecutar sentencias SQL que actualizan una tabla. En la práctica `executeUpdate` se utiliza más frecuentemente para actualizar tablas que para crearlas porque una tabla se crea sólo una vez, pero se puede actualizar muchas veces.

El método más utilizado para ejecutar sentencias SQL es `executeQuery`. Este método se utiliza para ejecutar sentencias `SELECT`, que comprenden la amplia mayoría de las sentencias SQL. Pronto veremos como utilizar este método.

## Introducir Datos en una Tabla

Hemos visto como crear la tabla `COFFEES` especificando los nombres de columnas y los tipos de datos almacenados en esas columnas, pero esto sólo configura la estructura de la tabla. La tabla no contiene datos todavía. Introduciremos datos en nuestra tabla una fila cada vez, suministrando la información a almacenar en cada columna de la fila. Observa que los valores insertados en las columnas se listan en el mismo orden en que se declararon las columnas cuando se creó la tabla, que es el orden por defecto.

El siguiente código inserta una fila de datos con `Colombian` en la columna `COF_NAME`, 101 en `SUP_ID`, 7.99 en `PRICE`, 0 en `SALES`, y 0 en `TOTAL`. (Como acabamos de inaugurar "The Coffee Break", la cantidad vendida durante la semana y la cantidad total son cero para todos los cafés). Al igual que hicimos con el código que creaba la tabla `COFFEES`, crearemos un objeto `Statement` y lo ejecutaremos utilizando el método `executeUpdate`.

Como la sentencia SQL es demasiado larga como para entrar en una sola línea, la hemos dividido en dos strings concatenándolas mediante un signo más (+) para que puedan compilarse. Presta especial atención a la necesidad de un espacio entre `COFFEES` y `VALUES`. Este espacio debe estar dentro de las comillas y debe estar después de `COFFEES` y antes de `VALUES`; sin un espacio, la sentencia SQL sería leída erróneamente como `"INSERT INTO COFFEESVALUES . . ."` y el controlador de la base de datos buscaría la tabla `COFFEESVALUES`. Observa también que utilizamos comilla simple alrededor del nombre del café porque está anidado dentro de las comillas dobles. Para la mayoría de controladores de bases de datos, la regla general es alternar comillas dobles y simples para indicar anidación.

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
    "INSERT INTO COFFEES " +
```

```
"VALUES ('Colombian', 101, 7.99, 0, 0)");
```

El siguiente código inserta una segunda línea dentro de la tabla COFFEES. Observa que hemos reutilizado el objeto Statement: stmt en vez de tener que crear uno nuevo para cada ejecución.

```
stmt.executeUpdate("INSERT INTO COFFEES " +  
"VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Los valores de las siguientes filas se pueden insertar de esta forma.

```
stmt.executeUpdate("INSERT INTO COFFEES " +  
"VALUES ('Espresso', 150, 9.99, 0, 0)");  
stmt.executeUpdate("INSERT INTO COFFEES " +  
"VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");  
stmt.executeUpdate("INSERT INTO COFFEES " +  
"VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

## Obtener Datos desde una Tabla

Ahora que la tabla COFFEES tiene valores, podemos escribir una sentencia SELECT para acceder a dichos valores. El asterisco (\*) en la siguiente sentencia SQL indica que la columna debería ser seleccionada. Como no hay cláusula WHERE que limite las columnas a seleccionar, la siguiente sentencia SQL selecciona la tabla completa.

```
SELECT * FROM COFFEES
```

El resultado, que es la tabla completa, se parecería a esto.

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

El resultado anterior es lo que veríamos en nuestro terminal si introdujéramos la petición SQL directamente en el sistema de la base de datos. Cuando accedemos a una base de

datos a través de una aplicación Java, como veremos pronto, necesitamos recuperar los resultados para poder utilizarlos. Veremos como hacer esto en la siguiente página.

Aquí tenemos otro ejemplo de una sentencia SELECT, ésta obtiene una lista de cafés y sus respectivos precios por libra.

```
SELECT COF_NAME, PRICE FROM COFFEES
```

El resultado de esta consulta se parecería a esto.

COF_NAME	-----	PRICE
Colombian		7.99
French_Roast		8.99
Espresso		9.99
Colombian_Decaf		8.99
French_Roast_Decaf		9.99

La sentencia SELECT genera los nombres y precios de todos los cafés de la tabla. La siguiente sentencia SQL limita los cafés seleccionados a aquellos que cuesten menos de \$9.00 por libra.

```
SELECT COF_NAME, PRICE  
FROM COFFEES  
WHERE PRICE < 9.00
```

El resultado se parecería es esto.

COF_NAME	-----	PRICE
Colombian		7.99
French_Roast		8.99
Colombian Decaf		8.99

## RECUPERAR VALORES DESDE UNA HOJA DE RESULTADOS

Ahora veremos como enviar la sentencia SELECT de la página anterior desde un programa escrito en Java y como obtener los resultados que hemos mostrado.

JDBC devuelve los resultados en un objeto ResultSet, por eso necesitamos declarar un ejemplar de la clase ResultSet para contener los resultados. El siguiente código presenta el objeto ResultSet: rs y le asigna el resultado de una consulta anterior.

```
ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

### Utilizar el Método next

La variable rs, que es un ejemplar de ResultSet, contiene las filas de cafés y sus precios mostrados en el juego de resultados de la página anterior. Para acceder a los nombres y los precios, iremos a la fila y recuperaremos los valores de acuerdo con sus tipos.

El método next mueve algo llamado cursor a la siguiente fila y hace que esa fila (llamada fila actual) sea con la que podamos operar. Como el cursor inicialmente se posiciona justo encima de la primera fila de un objeto ResultSet, primero debemos llamar al método next para mover el cursor a la primera fila y convertirla en la fila actual.

Sucesivas invocaciones del método next moverán el cursor de línea en línea de arriba a abajo. Observa que con el JDBC 2.0, cubierto en la siguiente sección, se puede mover el cursor hacia atrás, hacia posiciones específicas y a posiciones relativas a la fila actual además de mover el cursor hacia adelante.

### Utilizar los métodos getXXX

Los métodos getXXX del tipo apropiado se utilizan para recuperar el valor de cada columna. Por ejemplo, la primera columna de cada fila de rs es COF\_NAME, que almacena un valor del tipo VARCHAR de SQL. El método para recuperar un valor VARCHAR es getString. La segunda columna de cada fila almacena un valor del tipo FLOAT de SQL, y el método para recuperar valores de ese tipo es getFloat.

El siguiente código accede a los valores almacenados en la fila actual de rs e imprime una línea con el nombre seguido por tres espacios y el precio. Cada vez que se llama al método next, la siguiente fila se convierte en la actual, y el bucle continúa hasta que no haya más filas en rs.

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
```

```
Float n = rs.getFloat("PRICE");  
System.out.println(s + " " + n);  
}
```

La salida se parecerá a esto.

```
Colombian 7.99  
French_Roast 8.99  
Espresso 9.99  
Colombian_Decaf 8.99  
French_Roast_Decaf 9.99
```

Veamos cómo funcionan los métodos getXXX examinando las dos sentencias getXXX de este código. Primero examinaremos getString.

```
String s = rs.getString("COF_NAME");
```

El método getString es invocado sobre el objeto ResultSet: rs, por eso getString recuperará (obtendrá) el valor almacenado en la columna COF\_NAME de la fila actual de rs. El valor recuperado por getString se ha convertido desde un VARCHAR de SQL a un String de Java y se ha asignado al objeto String s. Observa que utilizamos la variable s en la expresión println mostrada arriba, de esta forma: println(s + " " + n)

La situación es similar con el método getFloat excepto en que recupera el valor almacenado en la columna PRICE, que es un FLOAT de SQL, y lo convierte a un float de Java antes de asignarlo a la variable n.

JDBC ofrece dos formas para identificar la columna de la que un método getXXX obtiene un valor. Una forma es dar el nombre de la columna, como se ha hecho arriba. La segunda forma es dar el índice de la columna (el número de columna), con un 1 significando la primera columna, un 2 para la segunda, etc. Si utilizáramos el número de columna en vez del nombre de columna el código anterior se podría parecer a esto.

```
String s = rs.getString(1);  
float n = rs.getFloat(2);
```

La primera línea de código obtiene el valor de la primera columna de la fila actual de rs (columna COF\_NAME), convirtiéndolo a un objeto String de Java y asignándolo a s. La segunda línea de código obtiene el valor de la segunda columna de la fila actual de rs, lo convierte a un float de Java y lo asigna a n. Recuerda que el número de columna se refiere al número de columna en la hoja de resultados no en la tabla original.

En suma, JDBC permite utilizar tanto el nombre como el número de la columna como argumento a un método getXXX. Utilizar el número de columna es un poco más eficiente, y hay algunos casos donde es necesario utilizarlo.

JDBC permite muchas lateralidades para utilizar los métodos getXXX para obtener diferentes tipos de datos SQL. Por ejemplo, el método getInt puede ser utilizado para recuperar cualquier tipo numérico de caracteres. Los datos recuperados serán convertidos a un int; esto es, si el tipo SQL es VARCHAR, JDBC intentará convertirlo en un entero. Se recomienda utilizar el método getInt sólo para recuperar INTEGER de SQL, sin embargo, no puede utilizarse con los tipos BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, o TIMESTAMP de SQL.

Métodos para Recuperar Tipos SQL muestra qué métodos pueden utilizarse legalmente para recuperar tipos SQL, y más importante, qué métodos están recomendados para recuperar los distintos tipos SQL. Observa que esta tabla utiliza el término "JDBC type" en lugar de "SQL type." Ambos términos se refieren a los tipos genéricos de SQL definidos en java.sql.Types, y ambos son intercambiables.

### Utilizar el método getString

Aunque el método getString está recomendado para recuperar tipos CHAR y VARCHAR de SQL, es posible recuperar cualquier tipo básico SQL con él. (Sin embargo, no se pueden recuperar los nuevos tipos de datos del SQL3. Explicaremos el SQL3 más adelante).

Obtener un valor con getString puede ser muy útil, pero tiene sus limitaciones. Por ejemplo, si se está utilizando para recuperar un tipo numérico, getString lo convertirá en un String de Java, y el valor tendrá que ser convertido de nuevo a número antes de poder operar con él.

## ACTUALIZAR TABLAS

Supongamos que después de una primera semana exitosa, el propietario de "The Coffee Break" quiere actualizar la columna SALES de la tabla COFFEES introduciendo el número de libras vendidas de cada tipo de café. La sentencia SQL para actualizar una columna se podría parecer a esto.

```
String updateString = "UPDATE COFFEES " +  
    "SET SALES = 75 " +  
    "WHERE COF_NAME LIKE 'Colombian'";
```

Utilizando el objeto stmt, este código JDBC ejecuta la sentencia SQL contenida en updateString.

```
stmt.executeUpdate(updateString);
```

La tabla COFFEES ahora se parecerá a esto.



COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Observa que todavía no hemos actualizado la columna TOTAL, y por eso tiene valor 0.

Ahora seleccionaremos la fila que hemos actualizado, recuperando los valores de las columnas COF\_NAME y SALES, e imprimiendo esos valores.

```
String query = "SELECT COF_NAME, SALES FROM COFFEES " +  
    "WHERE COF_NAME LIKE 'Colombian';"  
ResultSet rs = stmt.executeQuery(query);  
while (rs.next()) {  
    String s = rs.getString("COF_NAME");  
    int n = rs.getInt("SALES");  
    System.out.println(n + " pounds of " + s + " sold this week.");  
}
```

Esto imprimirá lo siguiente.

```
75 pounds of Colombian sold this week.
```

Cómo la cláusula WHERE limita la selección a una sólo línea, sólo hay una línea en la ResultSet: rs y una línea en la salida. Por lo tanto, sería posible escribir el código sin un bucle while.

```
rs.next();  
String s = rs.getString(1);  
int n = rs.getInt(2);  
System.out.println(n + " pounds of " + s + " sold this week.")
```

Aunque hay una sola línea en la hoja de resultados, necesitamos utilizar el método next para acceder a ella. Un objeto ResultSet se crea con un cursor apuntando por encima de la primera fila. La primera llamada al método next posiciona el cursor en la primera fila (y en

este caso, la única) de rs. En este código, sólo se llama una vez a next, si sucediera que existiera una línea, nunca se accedería a ella.

Ahora actualizaremos la columna TOTAL añadiendo la cantidad vendida durante la semana a la cantidad total existente, y luego imprimiremos el número de libras vendidas hasta la fecha.

```
String updateString = "UPDATE COFFEES " +  
    "SET TOTAL = TOTAL + 75 " +  
    "WHERE COF_NAME LIKE 'Colombian';  
stmt.executeUpdate(updateString);  
String query = "SELECT COF_NAME, TOTAL FROM COFFEES " +  
    "WHERE COF_NAME LIKE 'Colombian';  
ResultSet rs = stmt.executeQuery(query);  
while ( rs.next() ) {  
    String s = rs.getString(1);  
    int n = rs.getInt(2);  
    System.out.println(n + " pounds of " + s + " sold to date.");  
}
```

Observa que en este ejemplo, utilizamos el índice de columna en vez del nombre de columna, suministrando el índice 1 a getString (la primera columna de la hoja de resultados es COF\_NAME), y el índice 2 a getInt (la segunda columna de la hoja de resultados es TOTAL). Es importante distinguir entre un índice de columna en la tabla de la base de datos como opuesto al índice en la tabla de la hoja de resultados. Por ejemplo, TOTAL es la quinta columna en la tabla COFFEES pero es la segunda columna en la hoja de resultados generada por la petición del ejemplo anterior.

## Utilizar Sentencias Preparadas

Algunas veces es más conveniente o eficiente utilizar objetos PreparedStatement para enviar sentencias SQL a la base de datos. Este tipo especial de sentencias se deriva de una clase más general, Statement, que ya conocemos.

## Cuándo utilizar un Objeto PreparedStatement

Si queremos ejecutar muchas veces un objeto Statement, reduciremos el tiempo de ejecución si utilizamos un objeto PreparedStatement, en su lugar.

La característica principal de un objeto PreparedStatement es que, al contrario que un objeto Statement, se le entrega una sentencia SQL cuando se crea. La ventaja de esto es que en la mayoría de los casos, esta sentencia SQL se enviará al controlador de la base de datos inmediatamente, donde será compilado. Como resultado, el objeto PreparedStatement no sólo contiene una sentencia SQL, sino una sentencia SQL que ha sido precompilada. Esto

significa que cuando se ejecuta la `PreparedStatement`, el controlador de base de datos puede ejecutarla sin tener que compilarla primero.

Aunque los objetos `PreparedStatement` se pueden utilizar con sentencias SQL sin parámetros, probablemente nosotros utilizaremos más frecuentemente sentencias con parámetros. La ventaja de utilizar sentencias SQL que utilizan parámetros es que podemos utilizar la misma sentencia y suministrar distintos valores cada vez que la ejecutemos. Veremos un ejemplo de esto en las páginas siguientes.

## Crear un Objeto `PreparedStatement`

Al igual que los objetos `Statement`, creamos un objeto `PreparedStatement` con un objeto `Connection`. Utilizando nuestra conexión con abierta en ejemplos anteriores, podríamos escribir lo siguiente para crear un objeto `PreparedStatement` que tome dos parámetros de entrada.

```
PreparedStatement updateSales = con.prepareStatement(  
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

La variable `updateSales` contiene la sentencia SQL, "UPDATE COFFEES SET SALES = ? WHERE COF\_NAME LIKE ?", que también ha sido, en la mayoría de los casos, enviada al controlador de la base de datos, y ha sido precompilado.

## Suministrar Valores para los Parámetros de un `PreparedStatement`

Necesitamos suministrar los valores que se utilizarán en los lugares donde están las marcas de interrogación, si hay alguno, antes de ejecutar un objeto `PreparedStatement`. Podemos hacer esto llamado a uno de los métodos `setXXX` definidos en la clase `PreparedStatement`. Si el valor que queremos sustituir por una marca de interrogación es un `int` de Java, podemos llamar al método `setInt`. Si el valor que queremos sustituir es un `String` de Java, podemos llamar al método `setString`, etc. En general, hay un método `setXXX` para cada tipo Java.

Utilizando el objeto `updateSales` del ejemplo anterior, la siguiente línea de código selecciona la primera marca de interrogación para un `int` de Java, con un valor de 75.

```
updateSales.setInt(1, 75);
```

Cómo podríamos asumir a partir de este ejemplo, el primer argumento de un método `setXXX` indica la marca de interrogación que queremos seleccionar, y el segundo argumento el valor que queremos ponerle. El siguiente ejemplo selecciona la segunda marca de interrogación con el string "Colombian".

```
updateSales.setString(2, "Colombian");
```

Después de que estos valores hayan sido asignados para sus dos parámetros, la sentencia SQL de updateSales será equivalente a la sentencia SQL que hay en string updateString que utilizando en el ejemplo anterior. Por lo tanto, los dos fragmentos de código siguientes consiguen la misma cosa.

### Código 1.

```
String updateString = "UPDATE COFFEES SET SALES = 75 " +  
    "WHERE COF_NAME LIKE 'Colombian';  
stmt.executeUpdate(updateString);
```

### Código 2.

```
PreparedStatement updateSales = con.prepareStatement(  
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");  
updateSales.setInt(1, 75);  
updateSales.setString(2, "Colombian");  
updateSales.executeUpdate();
```

Utilizamos el método executeUpdate para ejecutar ambas sentencias stmt y updateSales. Observa, sin embargo, que no se suministran argumentos a executeUpdate cuando se utiliza para ejecutar updateSales. Esto es cierto porque updateSales ya contiene la sentencia SQL a ejecutar.

Mirando estos ejemplos podríamos preguntarnos por qué utilizar un objeto PreparedStatement con parámetros en vez de una simple sentencia, ya que la sentencia simple implica menos pasos. Si actualizáramos la columna SALES sólo una o dos veces, no sería necesario utilizar una sentencia SQL con parámetros. Si por otro lado, tuviéramos que actualizarla frecuentemente, podría ser más fácil utilizar un objeto PreparedStatement, especialmente en situaciones cuando la utilizamos con un bucle while para seleccionar un parámetro a una sucesión de valores. Veremos este ejemplo más adelante en esta sección.

Una vez que a un parámetro se ha asignado un valor, el valor permanece hasta que lo resetee otro valor o se llame al método clearParameters. Utilizando el objeto PreparedStatement: updateSales, el siguiente fragmento de código reutiliza una sentencia prepared después de resetar el valor de uno de sus parámetros, dejando el otro igual.

```
updateSales.setInt(1, 100);  
updateSales.setString(2, "French_Roast");  
updateSales.executeUpdate();  
// changes SALES column of French Roast row to 100  
updateSales.setString(2, "Espresso");  
updateSales.executeUpdate();  
// changes SALES column of Espresso row to 100 (the first
```

```
// parameter stayed 100, and the second parameter was reset  
// to "Espresso")
```

## Utilizar una Bucle para asignar Valores

Normalmente se codifica más sencillo utilizando un bucle for o while para asignar valores de los parámetros de entrada.

El siguiente fragmento de código demuestra la utilización de un bucle for para asignar los parámetros en un objeto PreparedStatement: updateSales. El array salesForWeek contiene las cantidades vendidas semanalmente. Estas cantidades corresponden con los nombres de los cafés listados en el array coffees, por eso la primera cantidad de salesForWeek (175) se aplica al primer nombre de café de coffees ("Colombian"), la segunda cantidad de salesForWeek (150) se aplica al segundo nombre de café en coffees ("French\_Roast"), etc. Este fragmento de código demuestra la actualización de la columna SALES para todos los cafés de la tabla COFFEES.

```
PreparedStatement updateSales;  
String updateString = "update COFFEES " +  
    "set SALES = ? where COF_NAME like ?";  
updateSales = con.prepareStatement(updateString);  
int [] salesForWeek = {175, 150, 60, 155, 90};  
String [] coffees = {"Colombian", "French_Roast", "Espresso",  
    "Colombian_Decaf", "French_Roast_Decaf"};  
int len = coffees.length;  
for(int i = 0; i < len; i++) {  
    updateSales.setInt(1, salesForWeek[i]);  
    updateSales.setString(2, coffees[i]);  
    updateSales.executeUpdate();  
}
```

Cuando el propietario quiera actualizar las ventas de la semana siguiente, puede utilizar el mismo código como una plantilla. Todo lo que tiene que hacer es introducir las nuevas cantidades en el orden apropiado en el array salesForWeek. Los nombres de cafés del array coffees permanecen constantes, por eso no necesitan cambiarse. (En una aplicación real, los valores probablemente serían introducidos por el usuario en vez de desde un array inicializado).

## Valores de retorno del método executeUpdate

Siempre que executeQuery devuelve un objeto ResultSet que contiene los resultados de una petición al controlador de la base datos, el valor devuelto por executeUpdate es un int que indica cuántas líneas de la tabla fueron actualizadas. Por ejemplo, el siguiente código muestra el valor de retorno de executeUpdate asignado a la variable n.

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it
```

La tabla COFFEES se ha actualizado poniendo el valor 50 en la columna SALES de la fila correspondiente a Espresso. La actualización afecta sólo a una línea de la tabla, por eso n es igual a 1.

Cuando el método `executeUpdate` es utilizado para ejecutar una sentencia DDL, como la creación de una tabla, devuelve el int: 0. Consecuentemente, en el siguiente fragmento de código, que ejecuta la sentencia DDL utilizada para crear la tabla COFFEES, n tendrá el valor 0.

```
int n = executeUpdate(createTableCoffees); // n = 0
```

Observa que cuando el valor devuelto por `executeUpdate` sea 0, puede significar dos cosas: (1) la sentencia ejecutada no ha actualizado ninguna fila, o (2) la sentencia ejecutada fue una sentencia DDL.

## Utilizar Uniones

Algunas veces necesitamos utilizar una o más tablas para obtener los datos que queremos. Por ejemplo, supongamos que el propietario del "The Coffee Break" quiere una lista de los cafés que le compra a Acme, Inc. Esto implica información de la tabla COFFEES y también de la que vamos a crear SUPPLIERS. Este es el caso en que se necesitan los "joins" (unión). Una unión es una operación de base de datos que relaciona dos o más tablas por medio de los valores que comparten. En nuestro ejemplo, las tablas COFFEES y SUPPLIERS tienen la columna SUP\_ID, que puede ser utilizada para unir las.

Antes de ir más allá, necesitamos crear la tabla SUPPLIERS y rellenarla con valores.

El siguiente código crea la tabla SUPPLIERS.

```
String createSUPPLIERS = "create table SUPPLIERS " +
    "(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
    "STREET VARCHAR(40), CITY VARCHAR(20), " +
    "STATE CHAR(2), ZIP CHAR(5))";
stmt.executeUpdate(createSUPPLIERS);
```

El siguiente código inserta filas para tres suministradores dentro de SUPPLIERS.

```
stmt.executeUpdate("insert into SUPPLIERS values (101, " +
    "'Acme, Inc.', '99 Market Street', 'Groundsville', " + "'CA', '95199'");
stmt.executeUpdate("Insert into SUPPLIERS values (49, " +
```

```
""Superior Coffee', '1 Party Place', 'Mendocino', 'CA', " + ""95460""");  
stmt.executeUpdate("Insert into SUPPLIERS values (150, " +  
""The High Ground', '100 Coffee Lane', 'Meadows', 'CA', " + ""93966""");
```

El siguiente código selecciona la tabla y nos permite verla.

```
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
```

El resultado sería algo similar a esto.

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
-----	-----	-----	-----	-----	-----
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Ahora que tenemos las tablas COFFEES y SUPPLIERS, podremos proceder con el escenario en que el propietario quería una lista de los cafés comprados a un suministrador particular. Los nombres de los suministradores están en la tabla SUPPLIERS, y los nombres de los cafés en la tabla COFFEES. Como ambas tablas tienen la columna SUP\_ID, podemos utilizar esta columna en una unión. Lo siguiente que necesitamos es la forma de distinguir la columna SUP\_ID a la que nos referimos.

Esto se hace precediendo el nombre de la columna con el nombre de la tabla, "COFFEES.SUP\_ID" para indicar que queremos referirnos a la columna SUP\_ID de la tabla COFFEES. En el siguiente código, donde stmt es un objeto Statement, seleccionamos los cafés comprados a Acme, Inc..

```
String query = "SELECT COFFEES.COF_NAME " +  
"FROM COFFEES, SUPPLIERS " +  
"WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.'" +  
"and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";  
ResultSet rs = stmt.executeQuery(query);  
System.out.println("Coffees bought from Acme, Inc.: ");  
while (rs.next()) {  
    String coffeeName = getString("COF_NAME");  
    System.out.println(" " + coffeeName);  
}
```

Esto producirá la siguiente salida.

```
Coffees bought from Acme, Inc..  
Colombian  
Colombian_Decaf
```

## UTILIZAR TRANSACCIONES

Hay veces que no queremos que una sentencia tenga efecto a menos que otra también suceda. Por ejemplo, cuando el propietario del "The Coffee Break" actualiza la cantidad de café vendida semanalmente, también querrá actualizar la cantidad total vendida hasta la fecha. Sin embargo, el no querrá actualizar una sin actualizar la otra; de otro modo, los datos serían inconsistentes. La forma para asegurarnos que ocurren las dos acciones o que no ocurre ninguna es utilizar una transacción. Una transacción es un conjunto de una o más sentencias que se ejecutan como una unidad, por eso o se ejecutan todas o no se ejecuta ninguna.

### Desactivar el modo Auto-entrega

Cuando se crea una conexión, está en modo auto-entrega. Esto significa que cada sentencia SQL individual es tratada como una transacción y será automáticamente entregada justo después de ser ejecutada. (Para ser más preciso, por defecto, una sentencia SQL será entregada cuando está completa, no cuando se ejecuta. Una sentencia está completa cuando todas sus hojas de resultados y cuentas de actualización han sido recuperadas. Sin embargo, en la mayoría de los casos, una sentencia está completa, y por lo tanto, entregada, justo después de ser ejecutada).

La forma de permitir que dos o más sentencias sean agrupadas en una transacción es desactivar el modo auto-entrega. Esto se demuestra en el siguiente código, donde con es una conexión activa.

```
con.setAutoCommit(false);
```

### Entregar una Transacción

Una vez que se ha desactivado la auto-entrega, no se entregará ninguna sentencia SQL hasta que llamemos explícitamente al método commit. Todas las sentencias ejecutadas después de la anterior llamada al método commit serán incluidas en la transacción actual y serán entregadas juntas como una unidad. El siguiente código, en el que con es una conexión activa, ilustra una transacción.

```
con.setAutoCommit(false);  
PreparedStatement updateSales = con.prepareStatement(
```



```
"UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

En este ejemplo, el modo auto-entrega se desactiva para la conexión con, lo que significa que las dos sentencias prepared updateSales y updateTotal serán entregadas juntas cuando se llame al método commit. Siempre que se llame al método commit (bien automáticamente, cuando está activado el modo auto-commit o explícitamente cuando está desactivado), todos los cambios resultantes de las sentencias de la transacción serán permanentes. En este caso, significa que las columnas SALES y TOTAL para el café Colombian han sido cambiadas a 50 (si TOTAL ha sido 0 anteriormente) y mantendrá este valor hasta que se cambie con otra sentencia de actualización.

La línea final del ejemplo anterior activa el modo auto-commit, lo que significa que cada sentencia será de nuevo entregada automáticamente cuando esté completa. Volvemos por lo tanto al estado por defecto, en el que no tenemos que llamar al método commit. Es bueno desactivar el modo auto-commit sólo mientras queramos estar en modo transacción. De esta forma, evitamos bloquear la base de datos durante varias sentencias, lo que incrementa los conflictos con otros usuarios.

## Utilizar Transacciones para Preservar al Integridad de los Datos

Además de agrupar las sentencias para ejecutarlas como una unidad, las transacciones pueden ayudarnos a preservar la integridad de los datos de una tabla. Por ejemplo, supongamos que un empleado se ha propuesto introducir los nuevos precios de los cafés en la tabla COFFEES pero lo retrasa unos días. Mientras tanto, los precios han subido, y hoy el propietario está introduciendo los nuevos precios. Finalmente el empleado empieza a introducir los precios ahora desfasados al mismo tiempo que el propietario intenta actualizar la tabla. Después de insertar los precios desfasados, el empleado se da cuenta de que ya no son válidos y llama el método rollback de la Connection para deshacer sus efectos. (El método rollback aborta la transacción y restaura los valores que había antes de intentar la actualización. Al mismo tiempo, el propietario está ejecutando una sentencia SELECT e imprime los nuevos precios. En esta situación, es posible que el propietario imprima los precios que más tarde serían devueltos a sus valores anteriores, haciendo que los precios impresos sean incorrectos.

Esta clase de situaciones puede evitarse utilizando Transacciones. Si un controlador de base de datos soporta transacciones, y casi todos lo hacen, proporcionará algún nivel de protección contra conflictos que pueden surgir cuando dos usuarios acceden a los datos a la misma vez.

Para evitar conflictos durante una transacción, un controlador de base de datos utiliza bloqueos, mecanismos para bloquear el acceso de otros a los datos que están siendo accedidos por una transacción. (Observa que en el modo auto-commit, donde cada sentencia es una transacción, el bloqueo sólo se mantiene durante una sentencia). Una vez activado, el bloqueo permanece hasta que la transacción sea entregada o anulada. Por ejemplo, un controlador de base de datos podría bloquear una fila de una tabla hasta que la actualización se haya entregado. El efecto de este bloqueo es evitar que usuario obtenga una lectura sucia, esto es, que lea un valor antes de que sea permanente. (Acceder a un valor actualizado que no haya sido entregado se considera una lectura sucia porque es posible que el valor sea devuelto a su valor anterior. Si leemos un valor que luego es devuelto a su valor antiguo, habremos leído un valor nulo).

La forma en que se configuran los bloqueos está determinado por lo que se llama nivel de aislamiento de transacción, que puede variar desde no soportar transacciones en absoluto a soportar todas las transacciones que fuerzan una reglas de acceso muy estrictas.

Un ejemplo de nivel de aislamiento de transacción es TRANSACTION\_READ\_COMMITTED, que no permite que se acceda a un valor hasta que haya sido entregado. En otras palabras, si nivel de aislamiento de transacción se selecciona a TRANSACTION\_READ\_COMMITTED, el controlador de la base de datos no permitirá que ocurran lecturas sucias. La interfase Connection incluye cinco valores que representan los niveles de aislamiento de transacción que se pueden utilizar en JDBC.

Normalmente, no se necesita cambiar el nivel de aislamiento de transacción; podemos utilizar el valor por defecto de nuestro controlador. JDBC permite averiguar el nivel de aislamiento de transacción de nuestro controlador de la base de datos (utilizando el método `getTransactionIsolation` de `Connection`) y permite configurarlo a otro nivel (utilizando el método `setTransactionIsolation` de `Connection`). Sin embargo, ten en cuenta, que aunque JDBC permite seleccionar un nivel de aislamiento, hacer esto no tendrá ningún efecto a no ser que el driver del controlador de la base de datos lo soporte.

## **Cuándo llamar al método `rollback`**

Como se mencionó anteriormente, llamar al método `rollback` aborta la transacción y devuelve cualquier valor que fuera modificado a sus valores anteriores. Si estamos intentando ejecutar una o más sentencias en una transacción y obtenemos una `SQLException`, deberíamos llamar al método `rollback` para abortar la transacción y empezarla de nuevo. Esta es la única forma para asegurarnos de cuál ha sido entregada y

cuál no ha sido entregada. Capturar una `SQLException` nos dice que hay algo erróneo, pero no nos dice si fue o no fue entregada. Como no podemos contar con el hecho de que nada fue entregado, llamar al método `rollback` es la única forma de asegurarnos.

## PROCEDIMIENTOS ALMACENADOS

Un procedimiento almacenado es un grupo de sentencias SQL que forman una unidad lógica y que realizan una tarea particular. Los procedimientos almacenados se utilizan para encapsular un conjunto de operaciones o peticiones para ejecutar en un servidor de base de datos. Por ejemplo, las operaciones sobre una base de datos de empleados (salarios, despidos, promociones, bloqueos) podrían ser codificados como procedimientos almacenados ejecutados por el código de la aplicación. Los procedimientos almacenados pueden compilarse y ejecutarse con diferentes parámetros y resultados, y podrían tener cualquier combinación de parámetros de entrada/salida.

Los procedimientos almacenados están soportados por la mayoría de los controladores de bases de datos, pero existe una gran cantidad de variaciones en su sintaxis y capacidades. Por esta razón, sólo mostraremos un ejemplo sencillo de lo que podría ser un procedimiento almacenado y cómo llamarlos desde JDBC, pero este ejemplo no está diseñado para ejecutarse.

### Utilizar Sentencias SQL

Esta página muestra un procedimiento almacenado muy sencillo que no tiene parámetros. Aunque la mayoría de los procedimientos almacenados hacen cosas más complejas que este ejemplo, sirve para ilustrar algunos puntos básicos sobre ellos. Como paso previo, la sintaxis para definir un procedimiento almacenado es diferente de un controlador de base de datos a otro. Por ejemplo, algunos utilizan **begin . . . end** u otras palabras clave para indicar el principio y final de la definición de procedimiento. En algunos controladores, la siguiente sentencia SQL crea un procedimiento almacenado.

```
create procedure SHOW_SUPPLIERS
as
  select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
  from SUPPLIERS, COFFEES
  where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
  order by SUP_NAME
```

El siguiente código pone la sentencia SQL dentro de un string y lo asigna a la variable `createProcedure`, que utilizaremos más adelante.

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
  "as " +
```

```
"select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " +  
"from SUPPLIERS, COFFEES " +  
"where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +  
"order by SUP_NAME";
```

El siguiente fragmento de código utiliza el objeto Connection, con para crear un objeto Statement, que es utilizado para enviar la sentencia SQL que crea el procedimiento almacenado en la base de datos.

```
Statement stmt = con.createStatement();  
stmt.executeUpdate(createProcedure);
```

El procedimiento SHOW\_SUPPLIERS será compilado y almacenado en la base de datos como un objeto de la propia base y puede ser llamado, como se llamaría a cualquier otro método.

## Llamar a un Procedimiento Almacenado desde JDBC

JDBC permite llamar a un procedimiento almacenado en la base de datos desde una aplicación escrita en Java. El primer paso es crear un objeto CallableStatement. Al igual que con los objetos Statement y PreparedStatement, esto se hace con una conexión abierta, Connection. Un objeto CallableStatement contiene una llamada a un procedimiento almacenado; no contiene el propio procedimiento. La primera línea del código siguiente crea una llamada al procedimiento almacenado SHOW\_SUPPLIERS utilizando la conexión con. La parte que está encerrada entre corchetes es la sintaxis de escape para los procedimientos almacenados. Cuando un controlador encuentra "{call SHOW\_SUPPLIERS}", traducirá esta sintaxis de escape al SQL nativo utilizado en la base de datos para llamar al procedimiento almacenado llamado SHOW\_SUPPLIERS.

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");  
ResultSet rs = cs.executeQuery();
```

La hoja de resultados de rs será similar a esto.

SUP_NAME	COF_NAME
Acme, Inc.	Colombian
Acme, Inc.	Colombian_Decaf
Superior Coffee	French_Roast
Superior Coffee	French_Roast_Decaf
The High Ground	Espresso

Observa que el método utilizado para ejecutar cs es executeQuery porque cs llama a un procedimiento almacenado que contiene una petición y esto produce una hoja de resultados. Si el procedimiento hubiera contenido una sentencia de actualización o una sentencia DDL,

se hubiera utilizado el método `executeUpdate`. Sin embargo, en algunos casos, cuando el procedimiento almacenado contiene más de una sentencia SQL producirá más de una hoja de resultados, o cuando contiene más de una cuenta de actualización o alguna combinación de hojas de resultados y actualizaciones. En estos casos, donde existen múltiples resultados, se debería utilizar el método `execute` para ejecutar `CallableStatement`.

La clase `CallableStatement` es una subclase de `PreparedStatement`, por eso un objeto `CallableStatement` puede tomar parámetros de entrada como lo haría un objeto `PreparedStatement`. Además, un objeto `CallableStatement` puede tomar parámetros de salida, o parámetros que son tanto de entrada como de salida. Los parámetros INOUT y el método `execute` se utilizan raramente.

## CREAR APLICACIONES JDBC COMPLETAS

---

Hasta ahora sólo hemos visto fragmentos de código. Más adelante veremos programas de ejemplo que son aplicaciones completas que podremos ejecutar.

El primer código de ejemplo crea la tabla `COFFEES`; el segundo inserta valores en la tabla e imprime los resultados de una petición. La tercera aplicación crea la tabla `SUPPLIERS`, y el cuarto la rellena con valores. Después de haber ejecutado este código, podemos intentar una petición que una las tablas `COFFEES` y `SUPPLIERS`, como en el quinto código de ejemplo. El sexto ejemplo de código es una aplicación que demuestra una transacción y también muestra como configurar las posiciones de los parámetros en un objeto `PreparedStatement` utilizando un bucle `for`.

Como son aplicaciones completas, incluyen algunos elementos del lenguaje Java que no hemos visto en los fragmentos anteriores. Aquí explicaremos estos elementos brevemente.

### Poner Código en una Definición de Clase

En el lenguaje Java, cualquier código que queramos ejecutar debe estar dentro de una definición de clase. Tecleamos la definición de clase en un fichero y a éste le damos el nombre de la clase con la extensión `.java`. Por eso si tenemos una clase llamada `MySQLStatement`, su definición debería estar en un fichero llamado `MySQLStatement.java`.

### Importar Clases para Hacerlas Visibles

Lo primero es importar los paquetes o clases que se van a utilizar en la nueva clase. Todas las clases de nuestros ejemplos utilizan el paquete `java.sql` (el API JDBC), que se hace visible cuando la siguiente línea de código precede a la definición de clase.

```
import java.sql.*;
```

El asterisco (\*) indica que todas las clases del paquete `java.sql` serán importadas. Importar una clase la hace visible y significa que no tendremos que escribir su nombre totalmente cualificado cuando utilicemos un método o un campo de esa clase. Si no incluimos `"import java.sql.*;"` en nuestro código, tendríamos que escribir `"java.sql."` más el nombre de la clase delante de todos los campos o métodos JDBC que utilicemos cada vez que los utilicemos. Observa que también podemos importar clases individuales selectivamente en vez de importar un paquete completo. Java no requiere que importemos clases o paquetes, pero al hacerlo el código se hace mucho más conveniente.

Cualquier línea que importe clases aparece en la parte superior de los ejemplos de código, que es donde deben estar para hacer visibles las clases importadas a la clase que está siendo definida. La definición real de la clase sigue a cualquier línea que importe clases.

### Utilizar el Método `main()`

Si una clase se va a ejecutar, debe contener un método **`static public main`**. Este método viene justo después de la línea que declara la clase y llama a los otros métodos de la clase. La palabra clave `static` indica que este método opera a nivel de clase en vez sobre ejemplares individuales de la clase. La palabra clave `public` significa que los miembros de cualquier clase pueden acceder a este método. Como no estamos definiendo clases sólo para ser ejecutadas por otras clases sino que queremos ejecutarlas, las aplicaciones de ejemplo de este capítulo incluyen un método `main`.

### Utilizar bloques `try` y `catch`

Algo que también incluyen todas las aplicaciones de ejemplo son los bloques `try` y `catch`. Este es un mecanismo del lenguaje Java para manejar excepciones. Java requiere que cuando un método lanza una excepción exista un mecanismo que la maneje. Generalmente un bloque `catch` capturará la excepción y especificará lo que sucederá (que podría ser no hacer nada). En el código de ejemplo, utilizamos dos bloques `try` y dos bloques `catch`. El primer bloque `try` contiene el método `Class.forName`, del paquete `java.lang`. Este método lanza una `ClassNotFoundException`, por eso el bloque `catch` que le sigue maneja esa excepción. El segundo bloque `try` contiene métodos JDBC, todos ellos lanzan `SQLException`, por eso el bloque `catch` del final de la aplicación puede manejar el resto de las excepciones que podrían lanzarse ya que todas serían objetos `SQLException`.

### Recuperar Excepciones

JDBC permite ver los avisos y excepciones generados por nuestro controlador de base de datos y por el compilador Java. Para ver las excepciones, podemos tener un bloque `catch` que las imprima. Por ejemplo, los dos bloques `catch` del siguiente código de ejemplo imprimen un mensaje explicando la excepción.

```
try {
    // Aquí va el código que podría generar la excepción.
    // Si se genera una excepción, el bloque catch imprimirá
    // información sobre ella.
} catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}

try {
    Class.forName("myDriverClassName");
} catch(java.lang.ClassNotFoundException e) {
    System.err.print("ClassNotFoundException: ");
    System.err.println(e.getMessage());
}
```

Si ejecutáramos CreateCOFFEES.java dos veces, obtendríamos un mensaje de error similar a éste.

```
SQLException: There is already an object named 'COFFEES' in the database.
Severity 16, State 1, Line 1
```

Este ejemplo ilustra la impresión del componente mensaje de un objeto SQLException, lo que es suficiente para la mayoría de las situaciones.

Sin embargo, realmente existen tres componentes, y para ser completos, podemos imprimirlos todos. El siguiente fragmento de código muestra un bloque catch que se ha completado de dos formas. Primero, imprime las tres partes de un objeto SQLException: el mensaje (un string que describe el error), el SQLState (un string que identifica el error de acuerdo a los convenciones X/Open de SQLState), y un código de error del vendedor (un número que es el código de error del vendedor del driver). El objeto SQLException, ex es capturado y se accede a sus tres componentes con los métodos getMessage, getSQLState, y getErrorCode.

La segunda forma del siguiente bloque catch completo obtiene todas las excepciones que podrían haber sido lanzada. Si hay una segunda excepción, sería encadenada a ex, por eso se llama a ex.getNextException para ver si hay más excepciones. Si las hay, el bucle while continúa e imprime el mensaje de la siguiente excepción, el SQLState, y el código de error del vendedor. Esto continúa hasta que no haya más excepciones.

```
try {
    // Aquí va el código que podría generar la excepción.
    // Si se genera una excepción, el bloque catch imprimirá
    // información sobre ella.
} catch(SQLException ex) {
```



```
System.out.println("\n--- SQLException caught ---\n");
while (ex != null) {
    System.out.println("Message: " + ex.getMessage ());
    System.out.println("SQLState: " + ex.getSQLState ());
    System.out.println("ErrorCode: " + ex.getErrorCode ());
    ex = ex.getNextException();
    System.out.println("");
}
}
```

Si hubiéramos sustituido el bloque catch anterior en el Código de ejemplo 1 (CreateCoffees) y lo hubiéramos ejecutado después de que la tabla COFFEES ya se hubiera creado, obtendríamos la siguiente información.

```
--- SQLException caught ---
Message: There is already an object named 'COFFEES' in the database.
Severity 16, State 1, Line 1
SQLState: 42501
ErrorCode: 2714
```

SQLState es un código definido en X/Open y ANSI-92 que identifica la excepción. Aquí podemos ver dos ejemplos de códigos SQLState.

```
08001 -- No suitable driver
HY011 -- Operation invalid at this time
```

El código de error del vendedor es específico de cada driver, por lo que debemos revisar la documentación del driver buscando una lista con el significado de estos códigos de error.

## Recuperar Avisos

Los objetos SQLWarning son una subclase de SQLException que trata los avisos de accesos a bases de datos. Los Avisos no detienen la ejecución de una aplicación, como las excepciones; simplemente alertan al usuario de que algo no ha salido como se esperaba. Por ejemplo, un aviso podría hacernos saber que un privilegio que queríamos revocar no ha fue revocado. O un aviso podría decirnos que ha ocurrido algún error durante una petición de desconexión.

Un aviso puede reportarse sobre un objeto Connection, un objeto Statement (incluyendo objetos PreparedStatement y CallableStatement), o un objeto ResultSet. Cada una de esas clases tiene un método getWarnings, al que debemos llamar para ver el primer aviso reportado en la llamada al objeto. Si getWarnings devuelve un aviso, podemos llamar al método getNextWarning de SQLWarning para obtener avisos adicionales. Al ejecutar una sentencia se borran automáticamente los avisos de la sentencia anterior, por eso no se



apilan. Sin embargo, esto significa que si queremos recuperar los avisos reportados por una sentencia, debemos hacerlo antes de ejecutar otra sentencia.

El siguiente fragmento de código ilustra como obtener información completa sobre los avisos reportados por el objeto Statement, stmt y también por el objeto ResultSet, rs.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select COF_NAME from COFFEES");

while (rs.next()) {

    String coffeeName = rs.getString("COF_NAME");
    System.out.println("Coffees available at the Coffee Break: ");
    System.out.println(" " + coffeeName);
    SQLWarning warning = stmt.getWarnings();
    if (warning != null) {
        System.out.println("\n---Warning---\n");
        while (warning != null) {
            System.out.println("Message: " + warning.getMessage());
            System.out.println("SQLState: " + warning.getSQLState());
            System.out.print("Vendor error code: ");
            System.out.println(warning.getErrorCode());
            System.out.println("");
            warning = warning.getNextWarning();
        }
    }
    SQLWarning warn = rs.getWarnings();
    if (warn != null) {
        System.out.println("\n---Warning---\n");
        while (warn != null) {
            System.out.println("Message: " + warn.getMessage());
            System.out.println("SQLState: " + warn.getSQLState());
            System.out.print("Vendor error code: ");
            System.out.println(warn.getErrorCode());
            System.out.println("");
            warn = warn.getNextWarning();
        }
    }
}
```

Los avisos no son muy comunes, de aquellos que son reportados, el aviso más común es un DataTruncation, una subclase de SQLWarning. Todos los objetos DataTruncation tienen un SQLState 01004, indicando que ha habido un problema al leer o escribir datos. Los métodos

de DataTruncation permiten encontrar en que columna o parámetro se truncaron los datos, si la ruptura se produjo en una operación de lectura o de escritura, cuántos bytes deberían haber sido transmitidos, y cuántos bytes se transmitieron realmente.

## EJECUTAR LA APLICACIÓN DE EJEMPLO

---

Ahora estamos listos para probar algún código de ejemplo. El directorio `book.html`, contiene una aplicación completa, ejecutable, que ilustra los conceptos presentados en este capítulo y el siguiente. Puedes descargar este código de ejemplo del site de JDBC situado en <http://java.sun.com/products/jdbc/reference/codesamples/index.html>.

Antes de poder ejecutar una de esas aplicaciones, necesitamos editar el fichero substituyendo la información apropiada para las siguientes variables.

- **url**

La URL JDBC, las partes uno y dos son suministradas por el driver, y la tercera parte especifica la fuente de datos.

- **myLogin**

Tu nombre de usuario o login.

- **myPassword**

Tu password para el controlador de base de datos.

- **myDriver.ClassName**

El nombre de clase suministrado con tu driver

La primera aplicación de ejemplo es la clase `CreateCoffees`, que está en el fichero llamado `CreateCoffees.java`. Abajo tienes las instrucciones para ejecutar `CreateCoffees.java` en las dos plataformas principales.

La primera línea compila el código del fichero `CreateCoffees.java`. Si la compilación tiene éxito, se producirá un fichero llamado `CreateCoffees.class`, que contendrá los bytecodes traducidos desde el fichero `CreateCoffees.java`. Estos bytecodes serán interpretados por la máquina virtual Java, que es la que hace posible que el código Java se pueda ejecutar en cualquier máquina que la tenga instalada.

La segunda línea de código ejecuta el código. Observa que se utiliza el nombre de la clase, `CreateCoffees`, no el nombre del fichero `CreateCoffees.class`.

## UNIX

```
javac CreateCoffees.java  
java CreateCoffees
```

### **Windows 95/NT**

```
javac CreateCoffees.java  
java CreateCoffees
```

# Capítulo 6

## Patrones de Diseño en JEE

### INTRODUCCIÓN

---

Como analistas y programadores vamos desarrollando a diario nuestras habilidades para resolver problemas usuales que se presentan en el desarrollo del software. Por cada problema que se nos presenta pensamos distintas formas de resolverlo, incluyendo soluciones exitosas que ya hemos usado anteriormente en problemas similares. Es así que a mayor experiencia que tengamos, nuestro abanico de posibilidades para resolver un problema crece, pero al final siempre habrá una sola solución que mejor se adapte a nuestra aplicación. Si documentamos esta solución, podemos reutilizarla y compartir esa información que hemos aprendido para resolver de la mejor manera un problema específico.

Los patrones del diseño tratan los problemas del diseño que se repiten y que se presentan en situaciones particulares del diseño, con el fin de proponer soluciones a ellas. Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes. Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones son llamadas estrategias.

### BREVE HISTORIA DE LOS PATRONES DE DISEÑO

---

Un patrón de diseño es una abstracción de una solución en un nivel alto. Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que abarcan las distintas etapas del desarrollo; desde el análisis hasta el diseño y desde la arquitectura hasta la implementación.

Muchos diseñadores y arquitectos de software han definido el término de patrón de diseño de varias formas que corresponden al ámbito a la cual se aplican los patrones. Luego, se dividió los patrones en diferentes categorías de acuerdo a su uso.

Los diseñadores de software extendieron la idea de patrones de diseño al proceso de desarrollo de software. Debido a las características que proporcionaron los lenguajes orientados a objetos (como herencia, abstracción y encapsulamiento) les permitieron relacionar entidades de los lenguajes de programación a entidades del mundo real fácilmente, los diseñadores empezaron a aplicar esas características para crear soluciones comunes y reutilizables para problemas frecuentes que exhibían patrones similares.

Fue por los años 1994, que apareció el libro "Design Patterns: Elements of Reusable Object Oriented Software" escrito por los ahora famosos Gang of Four (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Ellos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos. Desde luego que ellos no son los inventores ni los únicos involucrados, pero ese fue luego de la publicación de ese libro que empezó a difundirse con más fuerza la idea de patrones de diseño.

El grupo de GoF clasificaron los patrones en 3 grandes categorías basadas en su PROPÓSITO: creacionales, estructurales y de comportamiento.

- **Creacionales:** Patrones creacionales tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
- **Estructurales:** Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionales pueden ser incluso objetos simples u objetos compuestos.
- **Comportamiento:** Los patrones de comportamiento nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

En el segundo nivel, ellos clasificaron los patrones en 2 ámbitos: Clases y objetos. Es así que, tenemos 6 tipos de patrones:

- **Creacionales**

**Creacional de la Clase:** Los patrones creacionales de Clases usan la herencia como un mecanismo para lograr la instanciación de la Clase. Por ejemplo el método Factoría.

**Creacional del objeto:** Los patrones creacionales de objetos son más escalables y dinámicos comparados de los patrones creacionales de Clases. Por ejemplo la Factoría abstracta y el patrón Singleton.

- **Estructurales**

**Estructural de la Clase:** Los patrones estructurales de Clases usan la herencia para proporcionar interfaces más útiles combinando la funcionalidad de múltiples Clases. Por ejemplo el patrón Adaptador (Clase).

**Estructural de Objetos:** Los patrones estructurales de objetos crean objetos complejos agregando objetos individuales para construir grandes estructuras. La composición de l patrón estructural del objeto puede ser cambiado en tiempo de ejecución, el cual nos da flexibilidad adicional sobre los patrones estructurales de Clases. Por ejemplo el Adaptador (Objeto), Facade, Bridge, Composite.

- **Comportamiento**

**Comportamiento de Clase:** Los patrones de comportamiento de Clases usan la herencia para distribuir el comportamiento entre Clases. Por ejemplo Interpreter.

**Comportamiento de Objeto:** Los patrones de comportamiento de objetos nos permiten analizar los patrones de comunicación entre objetos interconectados, como objetos incluidos en un objeto complejo. Ejemplo Iterator, Observer, Visitor.

## PATRONES J2EE

---

Con la aparición del J2EE, todo un nuevo catálogo de patrones de diseño apareció. Desde que J2EE es una arquitectura por si misma que involucra otras arquitecturas, incluyendo servlets, JavaServer Pages, Enterprise JavaBeans, y más, merece su propio conjunto de patrones específicos para diferentes aplicaciones empresariales.

De acuerdo al libro "J2EE PATTERNS Best Practices and Design Strategies", existen 5 capas en la arquitectura J2EE:

- Cliente
- Presentación
- Negocios
- Integración
- Recurso

El libro explica 15 patrones J2EE que están divididos en 3 de las capas: presentación, negocios e integración.

## CATÁLOGO DE PATRONES J2EE

### Capa de Presentación

<b>Decorating Filter / Intercepting Filter</b>	Un objeto que está entre el cliente y los componentes Web. Este procesa las peticiones y las respuestas.
<b>Front Controller/ Front Component</b>	Un objeto que acepta todos los requerimientos de un cliente y los direcciona a manejadores apropiados. El patrón Front Controller podría dividir la funcionalidad en 2 diferentes objetos: el Front Controller y el Dispatcher. En ese caso, El Front Controller acepta todos los requerimientos de un cliente y realiza la autenticación, y el Dispatcher direcciona los requerimientos a manejadores apropiada.
<b>View Helper</b>	Un objeto helper que encapsula la lógica de acceso a datos en beneficio de los componentes de la presentación. Por ejemplo, los JavaBeans pueden ser usados como patrón View Helper para las páginas JSP.
<b>Composite view</b>	Un objeto vista que está compuesto de otros objetos vista. Por ejemplo, una página JSP que incluye otras páginas JSP y HTML usando la directiva include o el action include es un patrón Composite View.
<b>Service To Worker</b>	Es como el patrón de diseño MVC con el Controlador actuando como Front Controller pero con una cosa importante: aquí el Dispatcher (el cual es parte del Front Controller) usa View Helpers a gran escala y ayuda en el manejo de la vista.
<b>Dispatcher View</b>	Es como el patrón de diseño MVC con el controlador actuando como Front Controller pero con un asunto importante: aquí el Dispatcher (el cual es parte del Front Controller) no usa View Helpers y realiza muy poco trabajo en el manejo de la vista. El manejo de la vista es manejado por los mismos componentes de la Vista.

## Capa de Negocios

<b>Business Delegate</b>	Un objeto que reside en la capa de presentación y en beneficio de los otros componentes de la capa de presentación llama a métodos remotos en los objetos de la capa de negocios.
<b>Value Object/ Data Transfer Object/ Replicate Object</b>	Un objeto serializable para la transferencia de datos sobre la red.
<b>Session Facade/ Session Entity Facade/ Distributed Facade</b>	El uso de un bean de sesión como una fachada (facade) para encapsular la complejidad de las interacciones entre los objetos de negocio y participantes en un flujo de trabajo. El Session Facade maneja los objetos de negocio y proporciona un servicio de acceso uniforme a los clientes.
<b>Aggregate Entity</b>	Un bean entidad que es construido o es agregado a otros beans de entidad.
<b>Value Object Assembler</b>	Un objeto que reside en la capa de negocios y crea Value Objects cuando es requerido.
<b>Value List Handler/ Page-by-Page Iterator/ Paged List</b>	Es un objeto que maneja la ejecución de consultas SQL, caché y procesamiento del resultado. Usualmente implementado como beans de sesión.
<b>Service Locator</b>	Consiste en utilizar un objeto Service Locator para abstraer toda la utilización JNDI y para ocultar las complejidades de la creación del contexto inicial, de búsqueda de objetos home EJB y recreación de objetos EJB. Varios clientes pueden reutilizar el objeto Service Locator para reducir la complejidad del código, proporcionando un punto de control.

## Capa de Integración

<b>Data Access Object</b>	Consiste en utilizar un objeto de acceso a datos para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos.
---------------------------	--



<b>Service Activator</b>	Se utiliza para recibir peticiones y mensajes asíncronos de los clientes. Cuando se recibe un mensaje, el Service Activator localiza e invoca a los métodos de los componentes de negocio necesarios para cumplir la petición de forma asíncrona.
--------------------------	---

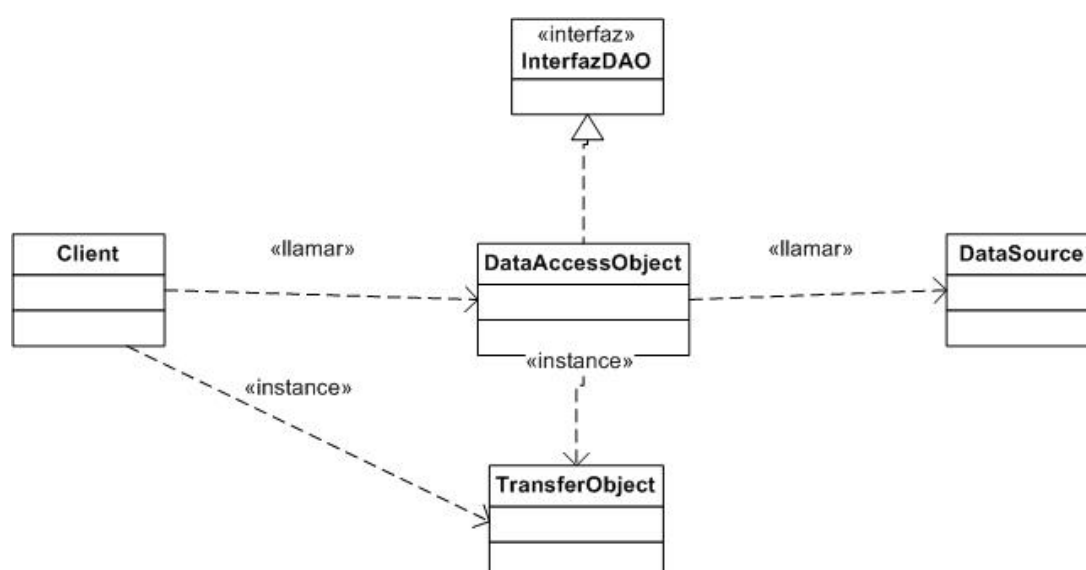
## PATRÓN "DATA ACCESS OBJECT"

### Descripción

El problema que viene a resolver este patrón es el de contar con diversas fuentes de datos (base de datos, archivos, servicios externos, etc). De tal forma que se encapsula la forma de acceder a la fuente de datos. Este patrón surge históricamente de la necesidad de gestionar una diversidad de fuentes de datos, aunque su uso se extiende al problema de encapsular no sólo la fuente de datos, sino además ocultar la forma de acceder a los datos. Se trata de que el software cliente se centre en los datos que necesita y se olvide de cómo se realiza el acceso a los datos o de cual es la fuente de almacenamiento.

Las aplicaciones pueden utilizar el API JDBC para acceder a los datos de una base de datos relacional. Este API permite una forma estándar de acceder y manipular datos en una base de datos relacional. El API JDBC permite a las aplicaciones J2EE utilizar sentencias SQL, que son el método estándar para acceder a tablas y vistas. La idea de este patrón es ocultar la fuente de datos y la complejidad del uso de JDBC a la capa de presentación o de negocio.

Un DAO define la relación entre la lógica de presentación y empresa por una parte y por otra los datos. El DAO tiene un interfaz común, sea cual sea el modo y fuente de acceso a datos.

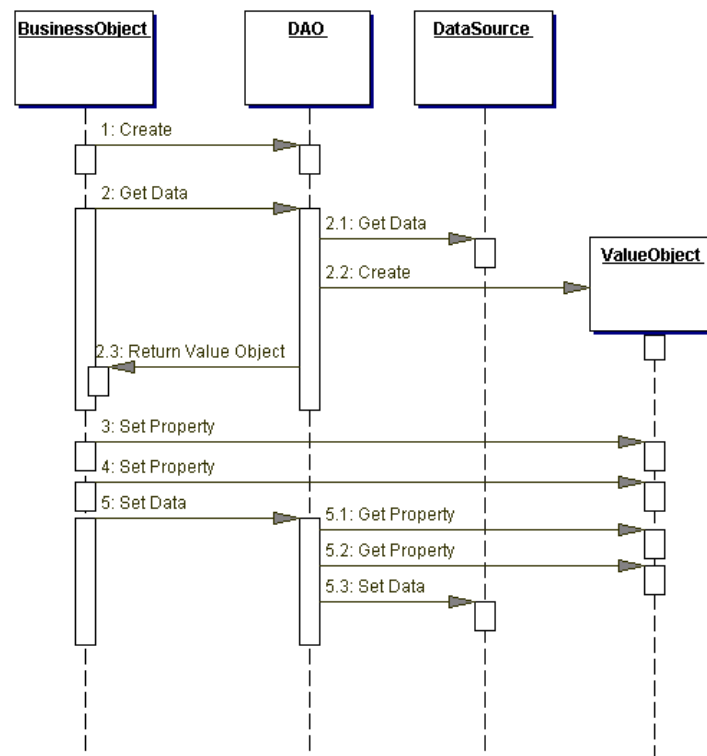


Algunas características:

1. No es imprescindible, pero en proyectos de cierta complejidad resulta útil que el DAO implemente un interfaz. De esta forma los objetos cliente tienen una forma unificada de acceder a los DAO.
2. El DAO accede a la fuente de datos y la encapsula para los objetos clientes. Entendiendo que oculta tanto la fuente como el modo (JDBC) de acceder a ella.
3. El TransferObject encapsula una unidad de información de la fuente de datos. El ejemplo sencillo es entenderlo como un "bean de tabla", es decir, como una representación de una tabla de la base de datos, por lo que representamos las columnas de la tabla como atributos del TransferObject. El DAO crea un TransferObject (o una colección de ellos) como consecuencia de una transacción contra la fuente de datos. Por ejemplo, una consulta sobre ventas debe crear tantos objetos (TransferObject) de la clase Venta como registros de la consulta; el DAO devolverá la colección de TransferObject de la clase Venta al objeto Cliente. También puede ocurrir que el objeto Cliente mande un TransferObject para parametrizar una consulta o actualización de datos por parte del DAO.

En el siguiente gráfico se muestran las interacciones entre los elementos del patrón. En este gráfico el TransferObject se denomina ValueObject. Puede observarse las llamadas que recibe y genera el DAO para una consulta y actualización de datos:

1. El DAO es creado por el cliente (BusinessObject) (llamada 1 del gráfico).
2. A continuación el cliente solicita los datos al DAO (getData) (2).
3. El DAO responde a la llamada pidiendo los datos a la fuente de datos (2.1).
4. Para cada fila recibida, el DAO crea un TransferObject (ValueObject del gráfico) (2.2).
5. El DAO devuelve al cliente el(los) TransferObject (2.3).
6. A continuación el cliente define un TransferObject mediante llamadas a setProperty. Por ejemplo, supongamos que buscamos personas de sexo varón y 36 años; para ello el BusinessObject define en el objeto de la clase Persona la edad y sexo que busca. Lo siguiente es fácil de imaginar: el BusinessObject invoca al DAO, pasando a la persona como argumento (3,4, y 5 del gráfico).
7. En DAO.setData() se solicita (5.1 y 5.2) al TransferObject o ValueObject (nuestra persona del ejemplo) los datos (edad, sexo, etc.) para realizar el acceso a datos (dataSource.setData()), (5.3).



## Ejemplo de código: los bean (Transfer Object)

En el siguiente sencillo ejemplo tenemos dos tablas en nuestra base de datos. La tabla de clientes incluye:

- Código (clave primaria)
- Nombre, apellido 1 y apellido 2
- Edad

En la tabla de ventas tenemos las ventas realizadas a cada cliente:

- Id de la venta (clave primaria)
- Código de cliente (clave externa)
- Precio y coste de la venta

Vamos a representar estas tablas en clases que de manera informal se conocen como "beans de tabla". Antes de crear estas tablas vamos a ver su interface común (**Bean.java**):

```
package dao.bean;

public interface Bean {
```

```
// Me indica si los objetos corresponden al mismo registro
// de base de datos (identidad de clave primaria)
public boolean esIgual( Bean bean );
}
```

A continuación escribimos los bean de tabla que implementan la interface anterior, empezamos por la clase **Cliente.java**:

```
/* *****
 * Bean de tabla "cliente"
 * ***** */
public class Cliente implements Bean {

    private String codigo = null;
    private String nombre = null;
    private String ape1 = null;
    private String ape2 = null;
    private Integer edad = null;
    private Vector ventas = null;

    public Cliente( String codigo, String nombre, String ape1, String ape2, Integer edad ) {
        setCodigo( codigo );
        setNombre( nombre );
        setApe1( ape1 );
        setApe2( ape2 );
        setEdad( edad );
    }

    public Cliente() { }

    public String getApe1() {
        return ape1;
    }

    public void setApe1(String ape1) {
        this.ape1 = ape1;
    }

    ... Otros métodos set/get ...

    /// Me indica si los objetos corresponden al mismo registro (identidad de clave primaria)
    public boolean esIgual( Bean bean ) {
        Cliente cli = (Cliente) bean;
        if ( cli.getCodigo().equals( this.getCodigo() ) ){
```

```
        return true;
    }
    return false;
}

public String toString() {
    return (codigo + ", " + nombre + ", " + ape1 + ", " + ape2 + ", " + edad);
}
}
```

En **Cliente.java** se puede observar que uno de los atributos es un vector de ventas. La utilidad de este Vector es representar en modos "objetos" una relación 1:N de tablas (cliente y ventas) de la base de datos. Seguimos con la clase **Venta.java**:

```
/* *****
 * Bean de tabla "venta"
 * ***** */
public class Venta implements Bean {

    Integer idVenta = null;
    String codigo = null;
    Float precio = null;
    Float coste = null;
    Cliente cliente = null;

    public Venta(Integer idVenta, String codigo, Float precio, Float coste ) {
        setIdVenta( idVenta );
        setCodigo( codigo );
        setPrecio( precio );
        setCoste( coste );
    }
    public Venta() { }

    public Float getCoste() {
        return coste;
    }

    public void setCoste(Float coste) {
        this.coste = coste;
    }

    ... Otros métodos set/get ...
}
```

```
//// Me indica si los objetos corresponden al mismo registro (identidad de clave primaria)
public boolean esIgual( Bean bean ) {
    Venta ven = (Venta) bean;
    if ( ven.getIdVenta().intValue() == this.getIdVenta().intValue() ) {
        return true;
    }
    return false;
}

public String toString() {
    return (idVenta + ", " + codigo + ", " + precio + ", " + coste );
}
}
```

## Ejemplo de código: los DAOs

Hemos empezado por lo más sencillo, representar las tablas de nuestra base de datos. Dicho de otra forma **proyectar el modelo relacional sobre un modelo de objetos**. Ahora tenemos que implementar los DAOs, los componentes que encapsulan el acceso a la fuente de datos (la base de datos). Empezamos creando un interface (**InterfaceDAO.java**) que representa el comportamiento genérico de cualquier DAO:

```
package dao.accesoDatos;

import java.sql.SQLException;
import java.util.Vector;
import dao.bean.Bean;

public interface InterfaceDAO {
    public int insert( Bean bean ) throws SQLException;
    public int update( Bean bean, String condicion ) throws SQLException;
    public Bean find( String codigo ) throws SQLException;
    public Vector select( String condicion ) throws SQLException;
    public int delete(String condicion) throws SQLException;
}
```

A continuación y a modo de resumen, una parte de **DAOCliente.java**. Se puede ver que implementa el interface anterior y que además hereda de **DAOGeneral**, una clase que contiene servicios comunes, como por ejemplo `getConexion()`, `cerrarConexion(Connection)`, etc:

```
public class DAOCliente extends DAOGeneral implements InterfaceDAO {
```

```
/*
*****
* Inserta cliente (argumento bean)
*****
*/
public int insert( Bean bean ) throws SQLException {
    int numFilas = 0;
    Cliente cli = (Cliente) bean;
    Connection con = getConexion();
    String orden = "INSERT INTO CLIENTE VALUES (" +
        (cli.getCodigo()==null? null: "" + cli.getCodigo() + " ") +
        ", " + (cli.getNombre()==null? null: "" + cli.getNombre() + " ") +
        ", " + (cli.getApe1()==null? null: "" + cli.getApe1() + " ") +
        ", " + (cli.getApe2()==null? null: "" + cli.getApe2() + " ") +
        ", " + cli.getEdad() + ")";
    Statement sentencia = con.createStatement();
    numFilas = sentencia.executeUpdate(orden);
    sentencia.close();
    cerrarConexion( con );
    return numFilas;
}
```

El método **insert()** recibe el Transfer Object (bean) que vamos a insertar, devolviendo el número de registros insertados (uno si ha ido bien, 0 en caso de no inserción). En el método **select()** recibimos la condición (cláusula WHERE) y devuelve un vector cuyos elementos son los clientes que cumplen la condición. Podría también devolver un ArrayList, que resulta más eficiente, pero lo esencial es que este método, al igual que el anterior, **oculta el uso de SQL y JDBC** a la clase que la llama (presentación o BusinessObject):

```
public Vector select( String condicion ) throws SQLException {
    Vector vecClientes = new Vector();
    Cliente cli;
    Connection con = getConexion();

    /// Si la condición es null o vacía, no hay parte WHERE
    String orden = "SELECT * FROM cliente c " +
        (condicion==null || condicion.length()==0 ? "" : "WHERE " + condicion) +
        " ORDER BY c.ape1, c.ape2, c.nombre";

    Statement sentencia = con.createStatement();
    ResultSet rs = sentencia.executeQuery( orden );

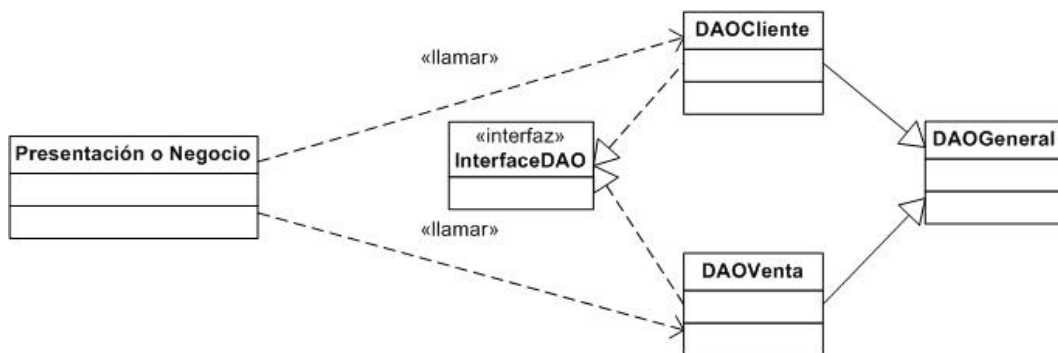
    while (rs.next()) {
        cli = new Cliente( rs.getString("codigo"), rs.getString("nombre"),
```

```

        rs.getString( "ape1" ), rs.getString( "ape2" ),
        (rs.getString( "edad" )==null ? null : new Integer(rs.getInt( "edad" ))));
    vecClientes.add( cli );
}
sentencia.close();
cerrarConexion( con );
return vecClientes;
}

```

En la siguiente figura se muestra la relación de las clases anteriores:



## Ejemplo de código: la factoria de DAOs

Aunque no resulta imprescindible, en proyectos de cierta envergadura necesitaremos una factoria de objetos DAO, que se responsabiliza de instanciar el DAO adecuado (en nuestro ejemplo DAOCliente.java o DAOVenta.java). En nuestro ejemplo hemos implementado una Factoria Simple (FactoriaDAO.java):

```

package dao.accesoDatos;
import java.io.*;

/*****
 *
 * Factoria que crea el DAO en función del argumento del método getDAO()
 *
 *****/
public class FactoriaDAO {

    // true=Carga de Properties desde archivo
    private static boolean propiedadesCargadas = false;

    // Propiedades
    private static java.util.Properties prop = new java.util.Properties();

```



```

/*****
 * Crea y devuelve el DAO
 *****/
public static InterfaceDAO getDAO( String nombre ) {

    try {
        // La clase se consigue leyendo del archivo properties
        Class clase = Class.forName( getClass( nombre ) );
        // Creo una instancia
        return (InterfaceDAO) clase.newInstance();
    } catch (ClassNotFoundException e) {    // No existe la clase
        e.printStackTrace();
        return null;
    }
    catch (Exception e) {                // No puedo instanciar la clase
        e.printStackTrace();
        return null;
    }
}

/*****
 * Lee un archivo properties donde se indica la clase que debe ser instanciada
 *****/
private static String getClass( String nombrePropiedad ) {

    String nombreClase = null;
    try {
        // Carga de propiedades desde archivo
        if ( !propiedadesCargadas ) {
            FileInputStream archivo = new
                FileInputStream( "src/dao/accesoDatos/propiedades.properties" );
            prop.load( archivo );    // Cargo propiedades
            propiedadesCargadas = true;
        }

        // Lectura de propiedad
        nombreClase = prop.getProperty( nombrePropiedad, "" );
        if ( nombreClase.length() == 0 )
            return null;
    } catch ( FileNotFoundException e ) {    // No se puede encontrar archivo
        e.printStackTrace();
    }
    catch ( IOException e ) {                // Falla load()

```

```
        e.printStackTrace();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return nombreClase;
}
}
```

Esta clase lee un archivo properties donde asociamos una clave con una clase DAO. Su contenido es sencillo:

```
cliente=dao.accesoDatos.DAOCliente
venta=dao.accesoDatos.DAOVenta
```

El método **getClase()** recibe un String (la clave: cliente o venta). Si no se ha cargado el archivo properties en el atributo '**Properties prop**', entonces lee archivo y carga las propiedades en **prop**. **getClase()** devuelve el valor de la clave, por ejemplo, si recibe 'cliente' devuelve 'dao.accesoDatos.DAOCliente'. El método **getDAO()** llama a **getClase()** para saber la clase DAO que debe instanciar.

Un aspecto importante es lo que devuelve **getDAO()**: el tipo **InterfaceDAO**. Al devolver el interface, estamos actuando con bastante generalidad: instanciamos objetos concretos, un **DAOCliente.java** o un **DAOVenta.java**, pero lo esencial es que **usamos como tipo de referencia el interface que ambas clases implementan**. La factoria hace su trabajo de modo **genérico o abstracto**, ya que aunque tenga 15 clases de DAOs diferentes que instanciar, devuelve un tipo genérico, es decir, devuelve el interface que implementan todos los DAOs.

### Ejemplo de código: usando lo anterior

Vamos a ver un sencillo ejemplo de capa cliente, es decir, la capa (sea presentación o de negocio) que debe usar lo anterior (factoria, beans y DAOs). Este sencillo ejemplo destaca una de las ventajas de trabajar con los interface: obtenemos abstracción (y resuabilidad). Veamos el ejemplo:

```
// Obtengo condición por teclado
condicion = Teclado.getCadena("Condición:");
// Obtengo el DAO de la factoria
InterfaceDAO dao = FactoriaDAO.getDAO( nombreTabla );
// La Select devuelve Vector
Vector vec = dao.select( condicion );
Iterator it = vec.iterator();
```

```
while (it.hasNext()) {  
    bean = (Bean) it.next();  
    System.out.println( bean.toString());  
}
```

La variable **nombreTabla** es un sencillo String que contiene la clave del archivo properties. De esta forma la factoria sabe la clase DAO que debe instanciar. Lo primero que interesa destacar es que el DAO instanciado y devuelto por la factoria lo manejamos por medio del tipo interface (InterfaceDAO): **sea la que sea la clase instanciada usamos un tipo genérico (el interface que implementan los DAO)**. Esto nos permite actuar con un alto grado de generalidad: **para cualquier tipo de DAO le mandamos el mismo mensaje: *dao.select( condicion )***. No nos vemos obligados a realizar un código para DAOCliente, otro para DAOVenta, etc; sino que **con el mismo código manejamos diversos DAOs**. Este es un ejemplo de patrón estrategia: el código es el mismo (unificado) y la estrategia (el comportamiento específico) depende de los objetos (DAOs) utilizados.

Un ejemplo final de abstracción es que los elementos del vector los manejamos con el tipo **interface Bean** (que implementan las clases 'bean'). Esto permite que les mande a todos el **mismo mensaje, *bean.toString()***, y que cada bean (sea de la clase que sea) implemente su comportamiento específico.

En este ejemplo las clases DAO han cambiado ligeramente para soportar un login y password diferentes en cada conexión.

## Nota final

Evidentemente los DAOs deben implementar los métodos de la interface (InterfaceDAO) que declaran. Pero además pueden implementar otros métodos que no están en el interfaz (lo cual resta generalidad). En nuestro ejemplo hemos introducido un método que nos permite hacer un select de clientes con sus ventas correspondientes. Antes hemos visto que uno de los atributos del 'bean' **Cliente.java** es un **vector de ventas**:

```
public class Cliente implements Bean {  
    ...  
    private Vector ventas = null;  
    ...  
}
```

Este vector nos permite representar en nuestro modelo de objetos la relación 1:N de nuestras tablas. El siguiente método de **DAOCliente.java** nos devuelve un vector de clientes y cada cliente contiene un vector de ventas:

```
/******  
* Consulta de clientes CON sus ventas  
* Si condición es null o vacia, no se aplica en el WHERE
```

```
*****/
public Vector selectConVentas( String condicion ) throws SQLException {
    Vector vecClientes = new Vector();
    Venta ven;
    Cliente cliAnterior = null, cliActual = null;

    Connection con = getConexion();

    // Si la condición es null o vacía, no hay parte WHERE
    String orden = "SELECT c.codigo, c.nombre, c.ape1, c.ape2, c.edad, v.id_venta, " +
        " v.precio, v.coste FROM cliente c, venta v " +
        " WHERE c.codigo = v.codigo " +
        (condicion==null || condicion.length()==0 ? "" : "AND " + condicion) +
        " ORDER BY c.ape1, c.ape2, c.nombre";

    Statement sentencia = con.createStatement();
    ResultSet rs = sentencia.executeQuery( orden );

    // Recorremos el ResultSet y guardamos los clientes en un vector.
    // Cada cliente tiene su vector de ventas
    while (rs.next()) {

        // Obtengo cliente y venta
        cliActual = new Cliente( rs.getString("c.codigo"), rs.getString("c.nombre"),
            rs.getString( "c.ape1" ), rs.getString( "c.ape2" ),
            (rs.getString( "c.edad" )==null ? null : new Integer(rs.getInt( "c.edad" ))));
        ven = new Venta( rs.getInt("v.id_venta"), rs.getString("c.codigo"),
            rs.getFloat("v.precio"), rs.getFloat("v.coste"));

        // SI ES NUEVO: Si es el primer cliente (cliAnterior==null) o es distinto que el anterior ...
        if ( cliAnterior == null || !cliActual.esIgual( cliAnterior ) ) {

            // El anterior es el actual
            cliAnterior = cliActual;

            // Inicializo vector de ventas del cliente actual y añado venta
            cliActual.setVentas( new Vector());
            cliActual.addVenta( ven );

            // Añado cliente actual al vector
            vecClientes.add( cliActual );
        }
        // SI NO ES NUEVO CLIENTE: simplemente añado venta al cliente
    }
}
```

```
else {  
    // Añado venta al cliente actual  
    cliAnterior.addVenta( ven );  
}  
  
}  
sentencia.close();  
cerrarConexion( con );  
return vecClientes;  
}
```

## DAO + FACTORY

### El problema

Hicimos la capa de persistencia de una aplicación con el patrón DAO. Funciona perfectamente, tenemos como motor de base de datos MySQL, y estamos tranquilos porque sabemos que si de pronto nuestro jefe se le ocurre que quiere usar otro medio de persistencia, simplemente vamos a tener que modificar la capa de persistencia, sin tener que tocar para nada la capa de lógica de negocio de la aplicación.

Bueno, como ya lo suponíamos, llega el día en que nuestro querido jefe se le ocurre usar como medio de persistencia la serialización en XML.

A nosotros no nos importa, ya que tenemos una capa de persistencia, y hacer los cambios no nos representa gran esfuerzo.

Lo que tenemos:

- Una interfaz DAO, con los métodos para hacer las operaciones CRUD sobre la base de datos.
- Las clases MySQLEmployerDAO y MySQLProductDAO, que implementan la interfaz DAO, con los métodos para hacer las consultas correspondientes a la base de datos.
- Los DTO: EmployerDTO y ProductDTO para transportar los datos. Que como dijimos en el primer artículo, en una aplicación MVC vendrían a ser los modelos, y representan por ejemplo en una RDB las tablas Employer y Product.

Bueno, entonces los cambios que tendríamos que hacer serían:

- Crear las clases XMLEmployerDAO y XMLProductDAO, implementando la interfaz DAO. Con los métodos CRUD para que en vez de hacer consultas a la base de datos MySQL, serialicen los datos en XML.

En la capa de lógica de negocio está lleno de inicializaciones de este tipo:

```
DAO dao = new MySQLEmployerDAO();  
DAO dao = new MySQLEmployerDAO();
```

Y ahora deberían ser:

```
DAO dao = new XMLEmployerDAO();  
DAO dao = new XMLEmployerDAO();
```

Estamos en un problema, (por si no se dieron cuenta).

Somos el hazme reir de la oficina, nuestra capa de persistencia da asco.

Ahora tenemos que cambiar la capa de lógica de negocio, tooodos los new MySQLEmployerDAO() y new MySQLProductDAO() por new XMLEmployerDAO() y new XMLProductDAO().

## La solución

Veamos... ¿cómo podemos mejorar nuestra capa de persistencia para que esto no vuelva a ocurrir?

Ya sabemos que crear la instancia de la clase **XMLEmployerDAO** desde la capa de lógica de negocio es definitivamente una mala idea. Ya que tendríamos que modificar la capa de lógica de negocio en caso de que la clase **XMLEmployerDAO** cambiase por la clase **OracleEmployerDAO...**

OK, entonces... ¿cómo podemos entonces instanciar la clase desde la capa de lógica de negocio sin hardcodear la clase?

Bueno, con el patrón Factory...

Con el patrón Factory, le delegamos el trabajo de instanciar la clase XMLEmployerDAO a otra clase, a una clase que se dedica a fabricar DAOs (una fábrica de DAOs) llamada por ejemplo: DAOFactory. Entonces, lo que antes era:

```
DAO dao = new XMLEmployerDAO();  
DAO dao = new XMLEmployerDAO();
```

Ahora va a ser:

```
DAO dao = daoFactory.getEmployerDAO();  
DAO dao = daoFactory.getEmployerDAO();
```

El método `getEmployerDAO()` va a devolver un DAO para Employer, que en este caso sería un `XMLEmployerDAO`, pero si en algún momento tuviéramos que cambiarlo por un `OracleEmployerDAO`, no habría problema. Ya que simplemente tendríamos que modificar el método `getEmployerDAO()` para que cree y devuelva un objeto `OracleEmployerDAO`, sin tener que cambiar nada en la capa de lógica de negocio, ya que lo único que hacemos ahí es llamar al método `getEmployerDAO()` de `DAOFactory`.

Además, desde la capa de lógica de negocio, vamos a usar el objeto `MySQLEmployerDAO` a través de una interfaz DAO (que tiene los métodos CRUD), ya que las clases `MySQLEmployerDAO`, `OracleEmployerDAO`, etc. van a implementar toda la interfaz DAO.

Bueno, veamos un ejemplo sencillo:

- Creamos la clase `DAOFactory`, esta clase tiene los métodos `getEmployerDAO()` y `getProductDAO()` que devuelven los DAOs correspondientes:

```
import dao.DAO;  
import dao.XMLEmployerDAO;  
import dao.XMLProductDAO;  
  
public class DAOFactory {  
  
    public XMLEmployerDAO getEmployerDAO() {  
        return new XMLEmployerDAO();  
    }  
  
    public XMLProductDAO getProductDAO() {  
        return new XMLProductDAO();  
    }  
  
}
```

- Y en la capa de lógica de negocio, ahora simplemente tenemos que pedirle a la fábrica que nos de un DAO para Employer o Product, así:

```
DAOFactory daoFactory = new DAOFactory();  
DAO employerDAO = daoFactory.getEmployerDAO();
```

Fíjense que usamos el objeto devuelto por la fábrica, a través de la interfaz DAO.

Como podrán ver, si ahora tuviéramos que cambiar la clase XMLEmployerDAO por OracleEmployerDAO, bastaría con modificar la DAOFactory, sin tener que tocar la capa de lógica de negocio.



# Capítulo 7

## iReport & JasperReport

### INTRODUCCIÓN

---

JasperReports es la mejor herramienta de código libre en Java para generar reportes. Puede entregar ricas presentaciones o diseños en la pantalla, para la impresora o para archivos en formato PDF, HTML, RTF, XLS, CSV y XML.

Está completamente escrita en Java y se puede utilizar en una gran variedad de aplicaciones de Java, incluyendo J2EE o aplicaciones Web, para generar contenido dinámico.

### Requerimientos de JasperReports

Se requiere tener instalado en el equipo el JDK 1.4 (SDK) o posterior. No basta con tener instalado el J2RE (Run Time Environment).

Las siguientes librerías junto con la de JasperReports deben incluirse en el proyecto en que se desee incluir esta herramienta para generar reportes.

- Jakarta Commons Digester Component (versión 1.1 o posterior)  
  
<http://jakarta.apache.org/commons/digester/>  
  
commons-digester.jar
- Jakarta Commons BeanUtils Component (versión 1.1 o posterior)  
  
<http://jakarta.apache.org/commons/beanutils/>  
  
commons-beanutils.jar
- Jakarta Commons Collections Component (versión 1.0 o posterior)  
  
<http://jakarta.apache.org/commons/collections/>  
  
commons-collections.jar
- Jakarta Commons Logging Component (versión 1.0 o posterior)  
  
<http://jakarta.apache.org/commons/logging/>

commonslogging.jar

- Driver JDBC 2.0 (Usualmente incluido en el SDK)
- PDF. Librería libre JavaPDF iText por Bruno Lowagie y Paulo Soares (versión 1.01 o posterior)

<http://www.lowagie.com/iText/>

itext1.02b.jar

- XLS Jakarta POI (versión 2.0 o posterior)

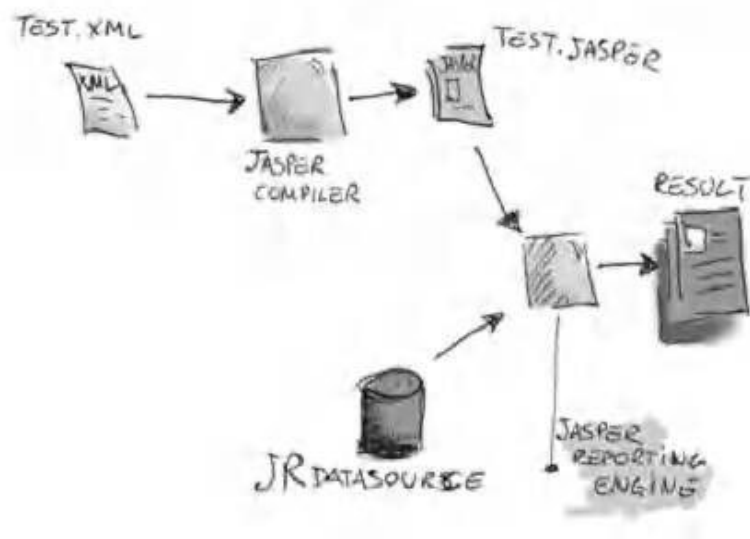
<http://jakarta.apache.org/poi/poi2.0final20040126.jar>

## **FUNCIONAMIENTO DE JASPERREPORTS**

---

JasperReports trabaja en forma similar a un compilador y a un intérprete, ver figura 1. El usuario diseña el reporte codificándolo en XML de acuerdo a las etiquetas y atributos definidos en un archivo llamado jasperreports.dtd (parte de JasperReports). Usando XML el usuario define completamente el reporte, describiendo donde colocar texto, imágenes, líneas, rectángulos, cómo adquirir los datos, como realizar ciertos cálculos para mostrar totales, etc.

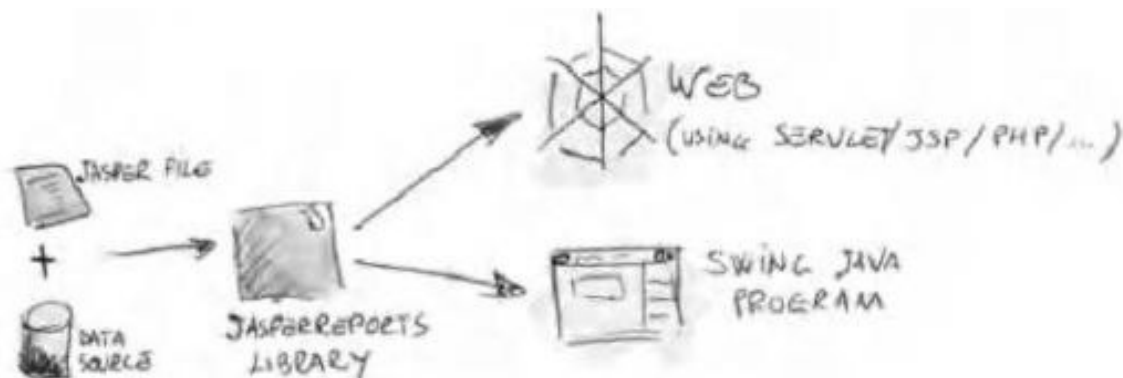
Este archivo fuente XML debe ser compilado para obtener un reporte real. La versión compilada del fuente es nombrada "archivo jasper" (este termina con .jasper). Un Archivo jasper es el compilado de un código fuente. Cuando tenemos un archivo jasper, necesitamos otra cosa para producir un reporte: necesitamos datos. Esto no siempre es cierto. En algunos casos querríamos generar un reporte que no mostrara datos dinámicos, solo texto estático por ejemplo, pero esto puede simplificarse a un reporte que tiene solamente un registro vacío. Para proporcionar estos registros al "jasper engine" necesitamos presentarlos usando una interfaz especial específica llamada JRDataSource. Una fuente de datos + un Archivo jasper = un "archivo print". Un "archivo print" puede exportarse en muchos formatos como PDF, HTML, RTF, XML, XLS, CVS, etc. La exportación se puede realizar utilizando clases especiales para implementar exportadores específicos.



## Compilación, exportación de reportes de JasperReports

Para un novato, diseñar y crear el archivo jasper es la tarea mas dura. Cuando se haya diseñado y compilado el archivo jasper, se puede utilizar la librería JasperReports para llenar dinámicamente el reporte en varios entornos como una aplicación web (Usando un servlet de Java por ejemplo, pero también funciona para generar reportes PDF desde un script PHP).

Jasper tiene disponible un visualizador especial para desplegar la vista previa de un reporte; diseñado para aplicaciones tradicionales de Java basadas en Swing.



## IREPORT

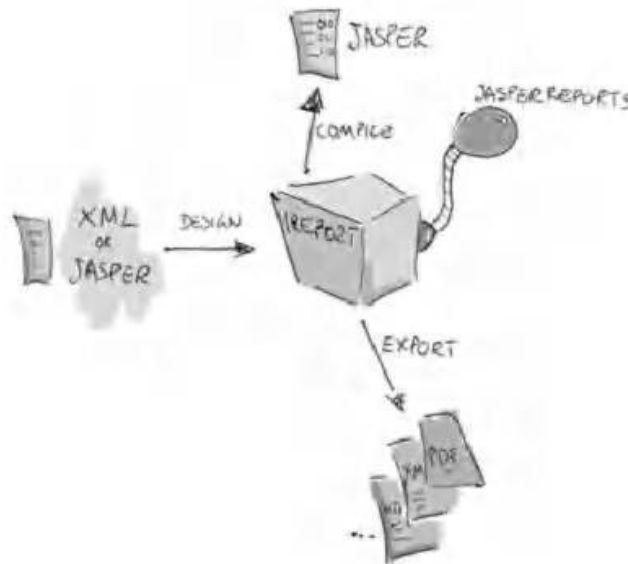
iReport es un diseñador visual de código libre para JasperReports escrito en Java. Es un programa que ayuda a los usuarios y desarrolladores que usan la librería JasperReports para diseñar reportes visualmente. A través de una interfaz rica y simple de usar, iReport provee las funciones más importantes para crear reportes amenos en poco tiempo.

iReport puede ayudar a la gente que no conoce la sintaxis XML para generar reportes de JasperReports.

## Funcionamiento de iReport

iReport provee a los usuarios de JasperReports una interfaz visual para construir reportes, generar archivos "jasper" y "print" de prueba. iReport nació como una herramienta de desarrollo, pero puede utilizarse como una herramienta de oficina para adquirir datos almacenados en una base de datos, sin pasar a través de alguna otra aplicación.

iReport puede leer y modificar ambos tipos de archivo, XML y jasper. A través de JasperReports, es capaz de compilar XML a archivos jasper y "ejecutar reportes" para llenarlos usando varios tipos de fuentes de datos (JRDataSource) y exportar el resultado a PDF, HTML, XLS, CSV,...



## Requerimientos de instalación (Windows 2000, NT, XP)

- Sun JDK 1.4 (SDK) o superior.
- Acrobat 5.0 no es requerido, pero es fuertemente recomendado.
- Si se desea conectar con una base de datos, se debe proporcionar el Driver JDBC correspondiente.

## Instalación y configuración ((Windows 2000, NT, XP))

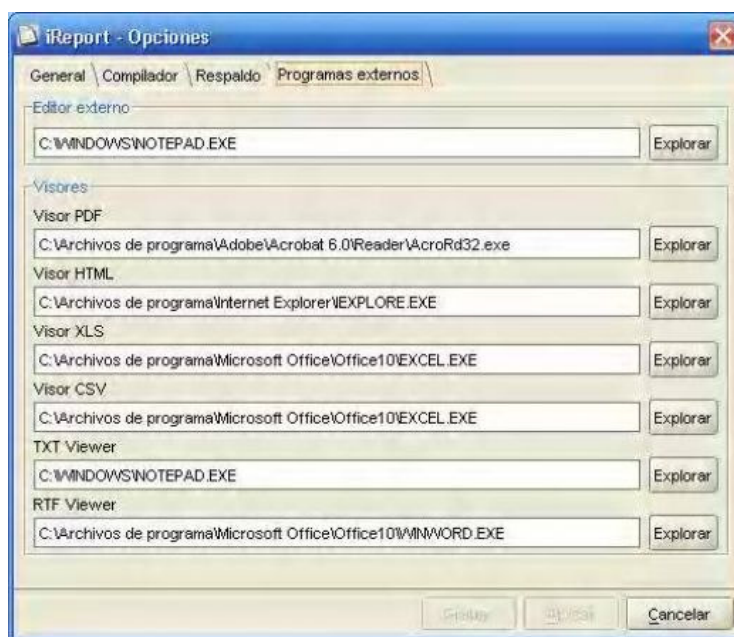
Si tiene instalado en su equipo un jdk (sdk) y no simplemente un j2re, está listo para iniciar la instalación.

1. Descomprima iReportx.x.x.zip y copie el directorio extraído al lugar que desee (C:\iReportx.x.x\ por ejemplo)
2. Busque un archivo llamado tools.jar en su jdk y cópielo en el directorio lib de iReport. (C:\iReport-x.x.x\lib)
3. Ejecute iReport.bat o iReport.sh.

Después de ejecutarse por primera vez, iReport creará un directorio (.ireport) en su directorio principal (home). Aquí se almacenarán todos los archivos de configuración en formato XML.

Proceda a configurar iReport con los siguientes pasos:

1. Vaya a menu>tools>options.
4. Vaya a la pestaña de Programas externos (external programs).
5. Configure los programas visualizadores externos.
6. En la pestaña General puede configurar el idioma



Pruebe si la configuración fue correcta. Cree un nuevo reporte en blanco, haga clic en el botón ejecutar sin conexión (run without connection).

Después de unos segundos aparecerá el reporte con el programa que se haya seleccionado para visualizarse.



## ¿Qué necesito descargar?

Este es el resumen de las librerías que debe descargar:

- • JasperReports
  - jasperreports1.0.1.jar
  - commonsdigester.jar
  - commonsbeanutils.jar
  - commonscollections.jar
  - commonslogging.jar
  - itext1.02b.jar
  - poi2.0final20040126.jar
- iReport
  - ○ iReport0.5.1 (versión 0.5.1)

Recuerde que es necesario tener instalado el Jdk 1.4 o superior, no basta con tener instalado el J2RE.

### NOTA:

En el directorio /lib de iReport se encuentran versiones de las librerías de jasperreports y librerías adicionales para JasperReports (commons-digester.jar, commons-beanutils.jar, ...).

Es muy importante asegurarse que la librería de jasper utilizada tanto en iReport como en su proyecto, sea la misma, de lo contrario obtendrá errores al intentar llenar dinámicamente su reporte desde otra aplicación Java. Para cualquier otra versión de iReport que descargue, debe verificar la versión máxima de JasperReports soportada si es que considera descargarla por separado.

Los sitios desde donde puede descargar JasperReports e iReport son los siguientes

- <http://jasperreports.sourceforge.net/>

- <http://ireport.sourceforge.net/index.php>)

En ambos sitios tiene la opción de descargar la distribución binaria, o bien el código fuente de los proyectos. Es recomendable obtener el fuente de JasperReports, en este se incluye toda la documentación y API's, además de código y ejemplos para realizar reportes. Debes obtener la herramienta ANT si deseas modificar y recompilar los fuentes de los proyectos.

¿Como agregar la librería de JasperReports y las librerías adicionales al proyecto?

Dependiendo del entorno de desarrollo que utilice (NetBeans, JBuilder, Eclipse...), puede agregar estas librerías a su IDE y posteriormente a su proyecto. Acerca del directorio en que deberían colocarse, algunos prefieren colocar las librerías en el directorio `\jre\lib\ext\` de su SDK. Personalmente trabajando en Windows prefiero colocar las librerías en un directorio del proyecto, como el `\lib`. Lo más importante es indicar al compilador el lugar en que estas se localizan.

En JBuilder, las librerías se pueden agregar accediendo al menú `tools -> Configure -> Libraries`. Seleccionando las carpetas `Project` o `User` se oprime el botón `New`, se pone el nombre de la librería y se especifica el directorio en que se encuentran. Para agregarlas al proyecto, se va al menú `Project->Project Properties` y en la pestaña `Required libraries` se selecciona la de JasperReports que acaba de agregar y listo.

Si la compilación se realiza manualmente, deberá utilizar `-classpath` en la línea de comando para especificar la ubicación de las librerías. Un ejemplo de línea de comando es el siguiente:

```
-classpath "C:\JasperReports\jasperreports-1.0.1.jar;C:\JasperReports\commons-digester.jar; ... "
```

Recuerde que debe incluir no solo la librería JasperReports; también sus librerías adicionales, de lo contrario podría obtener errores al compilar su proyecto. La librería `iReport.jar` en el directorio `\lib` de su iReport, debería agregarse al proyecto solo si diseña el reporte con iReport y pretende compilarlo desde su aplicación Java. En nuestro caso, se diseñará y compilará el reporte con iReport para generar un archivo `.jasper`, por lo que no es necesario agregar iReport.jar al proyecto.

En la segunda parte del artículo, se mostrará paso a paso la manera de diseñar un reporte con iReport. De la compilación de este reporte, se obtendrá un archivo `*.jasper`, el cual será llenado y mostrado dinámicamente desde una aplicación Java Swing.

## Configuración de la conexión a una base de datos.

Para establecer una conexión entre iReport y una base de datos, se debe proporcionar el driver JDBC correspondiente. La versión 0.5.1 de iReport ya proporciona drivers JDBC para establecer conexiones con bases de datos como MySQL y Access. Para nuestro ejemplo se

usará una conexión con una base de datos Access, para la cual se ha configurado un origen de datos con nombre DSN. En el artículo “El puente JDBC-ODBC(Ejemplo de conexión entre Access y Java)” se explica detalladamente la manera de configurar un origen de datos.

Suponiendo que se ha configurado un origen de datos nombrado para una base de datos Access con los siguientes valores:

Nombre de driver: PDRV Nombre de inicio de sesión: cvazquez Contraseña: vazquez

Procedemos a configurar iReport para establecer la conexión con la base de datos, para ello debe ir a menú->Fuente de datos->Conexiones/Fuente de datos. En la pantalla Connections/Datasources oprima el botón new para agregar una conexión.

La pantalla de conexión debe llenarse tal como se muestra a continuación:



A continuación oprima el botón Test para probar la conexión. Si la conexión fue exitosa, oprima finalmente el botón Save para guardar esta conexión.





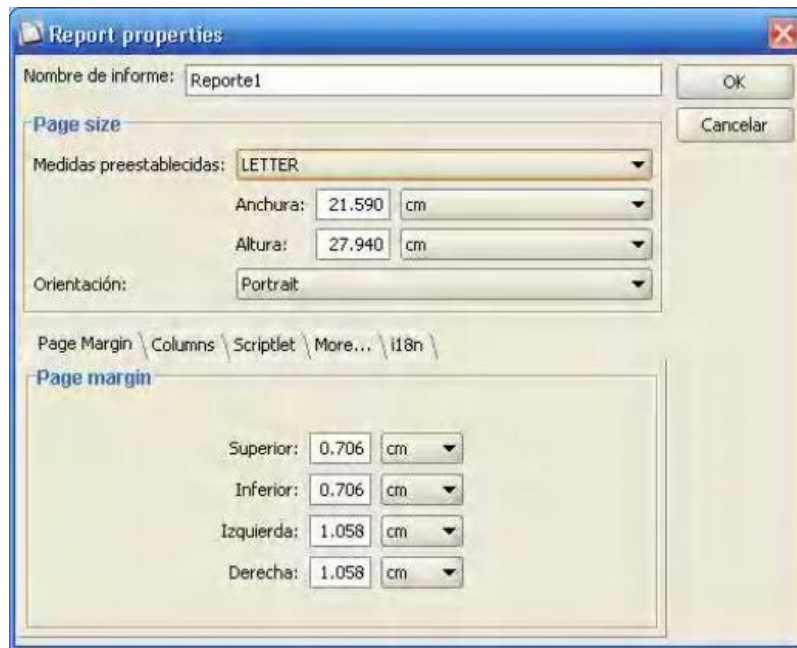
## Creación del Reporte.

En iReport, se tiene la opción para trabajar por proyecto, el cual puede contener varios reportes, en nuestro caso no se creará un proyecto, se creará solo un reporte de la siguiente manera:

Seleccione nuevo documento del menú Fichero o bien oprima el botón new report de la barra de herramientas, aparecerá una pantalla de propiedades del nuevo reporte que queremos crear:

En esta pantalla podemos configurar las propiedades del reporte, en nuestro caso, le llamaremos Reporte1, oprimir el botón OK para crearlo.

Seleccionar la opción Guardar como... del menú fichero o bien el botón Save report de la barra de herramientas, debe seleccionar el nombre y el directorio en que se guardara el reporte. El reporte se guardará con la extensión .xml. Por defecto los archivos de salida de la compilación se crearán en el directorio de instalación de iReport si no especificó uno.



## Secciones de un Reporte en iReport.

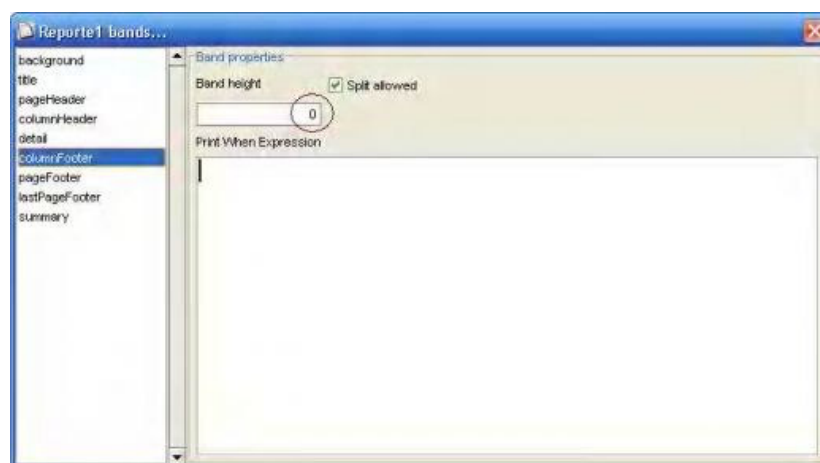
A continuación se explicará de manera breve, las secciones que componen a un reporte en iReport

	title	
	pageHeader	
	columnHeader	
	detail	
	columnFooter	
	pageFooter	
	lastPageFooter	
	summary	

- **title.** El título de nuestro reporte debe escribirse en esta sección. Solo se mostrará en la primera página del reporte.
- **pageHeader.** Aparece en la parte superior de cada página. Puede contener información adicional del reporte, descripciones, etc.
- **columnHeader.** En esta sección se muestran los nombres de los campos que se van a presentar

- **detail.** En esta sección se despliegan los valores correspondientes a los nombres de los campos definidos en la sección anterior. Estos datos pueden obtenerse mediante consultas SQL a una base de datos por ejemplo.
- **columnFooter.** Puede presentar información de totales para algunos de los campos de la sección detail. Por ejemplo "Total de Empleados: 220"
- **pageFooter.** Aparece en la parte inferior de cada página. Este parte puede presentar, la fecha, número de página del reporte.
- **summary.** Esta sección puede presentar totales de campos de la sección detail. Si se desea incluir algún gráfico en el reporte, debe hacerse en esta sección.

En el diseño de su reporte pueden omitirse algunas de las secciones o bandas mencionadas, en nuestro caso solo usaremos las secciones title, PageHeader, ColumHeader, detail, y Pagefooter. Para omitir las secciones del reporte que no se usaran, debe oprimir el botón bands de la barra de herramientas, o bien haciendo click con el botón secundario del ratón sobre el diseño del reporte y seleccionando la opción band properties del menú contextual. En la pantalla de propiedades de las bandas, debe seleccionar las bandas no deseadas y colocar su propiedad band height igual a cero como se muestra en la siguiente figura.



## Diseño del Reporte.

Se muestran a continuación los botones principales para el diseño del reporte de la barra de herramientas:



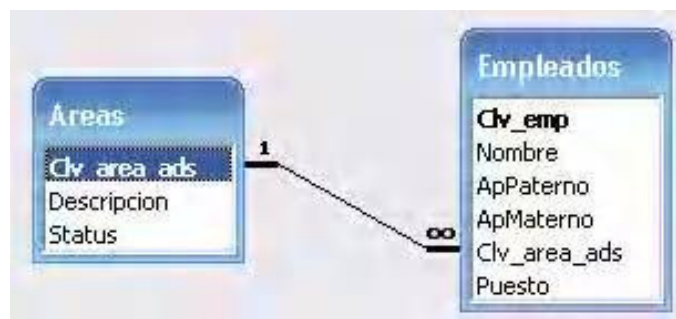


Agreguemos en primer lugar el título de nuestro reporte, para ello seleccione de la barra de herramientas el objeto Static text tool y dibuje una caja sobre la banda title. Haciendo doble click sobre la caja dibujada se mostrará la pantalla de propiedades de la caja de texto estático. Aquí puede configurar el tamaño, ubicación de la caja de texto, tipo de letra para el texto, entre otros; seleccionando la pestaña Static Text puede ingresar el título que desee para el reporte, el resultado debe ser parecido al de la siguiente figura:

En el encabezado de página, pageHeader, podemos colocar una descripción del reporte utilizando también el objeto Static Text tool.

Ahora se agregarán los nombres de los campos que pretendemos mostrar en el reporte, en este caso se recuerda que se configuró una conexión con una base de datos Access a través de un driver u origen de datos con nombre DSN.

Para cuestiones de prueba he agregado las siguientes tablas con los siguientes campos a la base de datos:



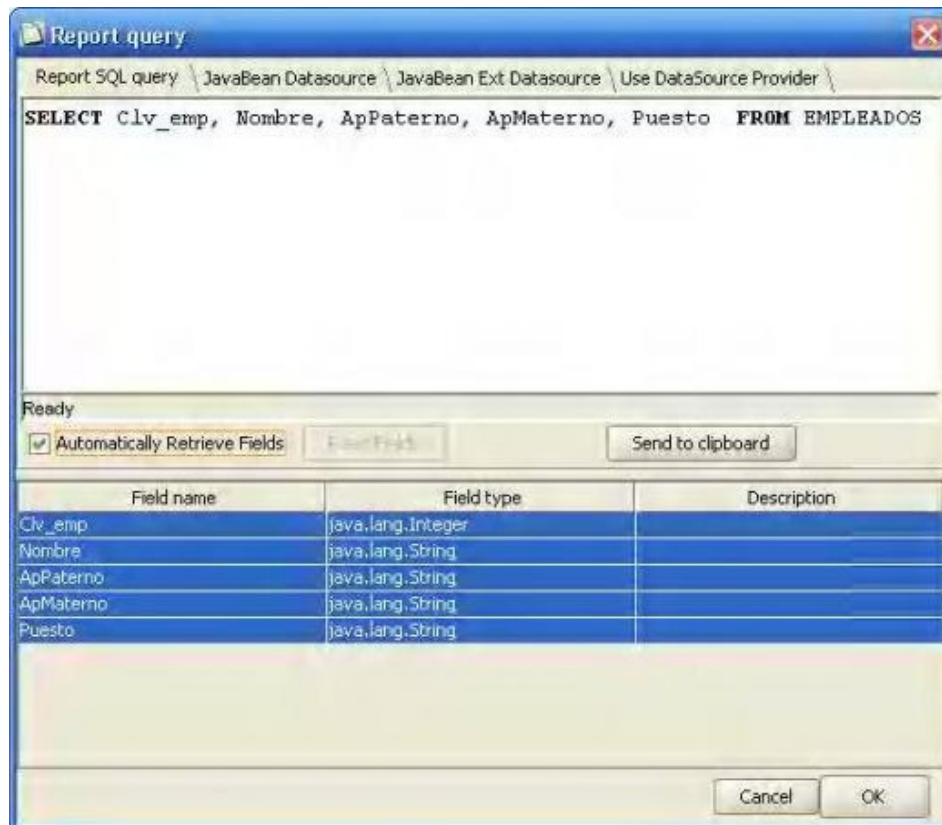
La relación de las tablas como se observa es “uno a muchos”. En este contexto, un empleado puede pertenecer a solo una área de trabajo y una área puede relacionarse con uno o muchos empleados.

Se realiza por ahora una consulta sencilla para el reporte a la tabla de empleados de la siguiente manera:

```
SELECT Clv_emp, Nombre, ApPaterno, ApMaterno, Puesto FROM EMPLEADOS
```

Antes, de agregar los nombres y campos a nuestro reporte, se establecerá la consulta anterior para el reporte. Vaya a la barra de herramientas y seleccione el botón Database, en la pantalla Report Query y pestaña Report SQL query, puede escribirse la sentencia SQL. Si

se encuentra seleccionado el check box Automatically Retrieve Fields, nos mostrará automáticamente los campos que se obtiene la consulta, el tipo y una descripción de estos si es que cuentan con ella. Si la consulta es incorrecta mostrará un mensaje de error. La pantalla debe lucir como sigue:



Debe oprimir el botón OK para guardar esta consulta.

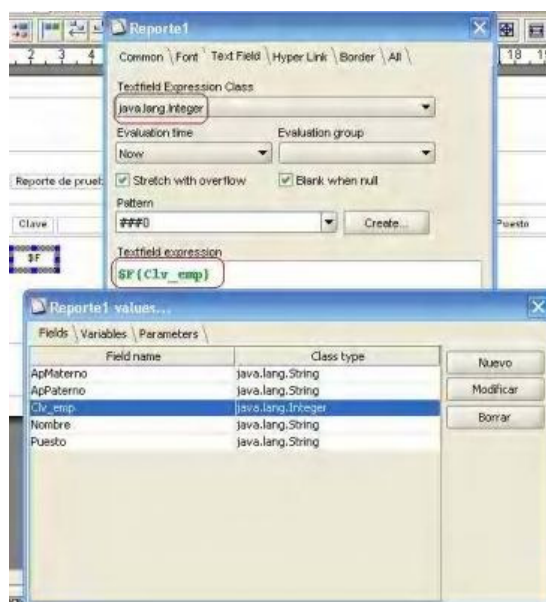
De la misma manera en que se agregó el título al reporte, deberá agregar los nombres de los campos en la banda columnHeader utilizando objetos Static Text tool. El reporte debe lucir hasta ahora como se muestra a continuación:

Mi primer reporte				
Reporte de prueba en iReport, muestra los datos de empleados				
Clave	Nombre	Apellido Paterno	Apellido Materno	Puesto
Detail				
Page Footer				

Ahora solo resta colocar en la sección detail, los campos que se mostrarán en el reporte. Para esto se usará el objeto Text Field, las cajas se pintarán de manera similar a las cajas de texto estático realizadas en la banda columnHeader, sin embargo cada campo debe configurarse de acuerdo al tipo de dato que se quiere mostrar.

A continuación se mostrará la manera de configurar un Text Field. Una vez colocado un campo en la sección detail, haga doble click sobre este para abrir su ventana de propiedades y sitúese en la pestaña Text Field. Vaya enseguida al menú Ver y seleccione el item Campos de informe, esto desplegará la pantalla values con los campos de nuestro reporte, los cuales se generaron al establecer la consulta SQL. Esta pantalla muestra adicionalmente los parámetros y variables del reporte, cada uno se distinguirá con la siguiente notación: Campos:  $\$F\{\text{Campo}\}$  Variables:  $\$V\{\text{valor}\}$  Parámetros:  $\$P\{\text{Parámetro}\}$

Utilice esta pantalla para auxiliarse al configurar un Text Field.



La figura anterior muestra como debe configurarse el campo “clave de empleado”. En la ventana de propiedades ponga especial atención en seleccionar el tipo correcto del campo en el combo Textfield ExpressionClass (Integer en este caso). En la sección Textfield expression de la misma ventana, cambie la expresión  $\$F\{\text{Field}\}$  por el nombre correcto del campo  $\$F\{\text{Clv\_emp}\}$ . Configure así cada uno de los campos restantes.

Adicionalmente, se agregará una línea al inicio de la sección detail para separar los registros con el objeto line tool de la barra de herramientas. Debe reducir el tamaño de la banda detail al alto de las cajas de los campos para evitar demasiado espacio entre los registros y listo, con ligeras adecuaciones a los campos, el reporte final debería lucir de la siguiente manera:



Mi primer reporte				
Reporte de prueba en iReport, muestra los datos de empleados				
Clave	Nombre	Apellido Paterno	Apellido Materno	Puesto
\$F	\$F{Nombre}	\$F{ApPaterno}	\$F{ApMaterno}	\$F{Puesto}
pageFooter				

## Compilación y Ejecución del Reporte.

Las siguientes figuras muestran los botones de la barra de herramientas necesarios para compilar, y ejecutar el reporte con o sin conexión.



Antes que nada, seleccione la vista para el Reporte, vaya al menú Construir y seleccione el item vista previa en JRViewer (Vista previa en el Viewer de Jasper). En este menú, puede seleccionar la vista previa para distintos formatos de archivo, siempre y cuando haya configurado los programas externos como se explicó en la primera parte del artículo.

Compile el reporte, el resultado de la compilación aparecerá en la parte inferior de la pantalla. Los errores más frecuentes de compilación se relacionan con los tipos de los campos que pretenden mostrarse. Si la compilación resultó sin errores, esta listo para ver su reporte, es recomendable probarlo primero sin usar una conexión a una base de datos. Finalmente, ejecute el reporte ocupando la conexión a la base de datos que se configuró. El resultado dependiendo de sus datos en las tablas debe ser parecido al siguiente:



The screenshot shows a window titled "Report JasperViewer". Inside, the report is titled "Mi primer reporte" and "Reporte de prueba en iReport, muestra los datos de empleados". It displays a table with 5 columns: Clave, Nombre, Apellido Paterno, Apellido Materno, and Puesto. The table contains 18 rows of employee data. At the bottom, it says "Page 1 of 99".

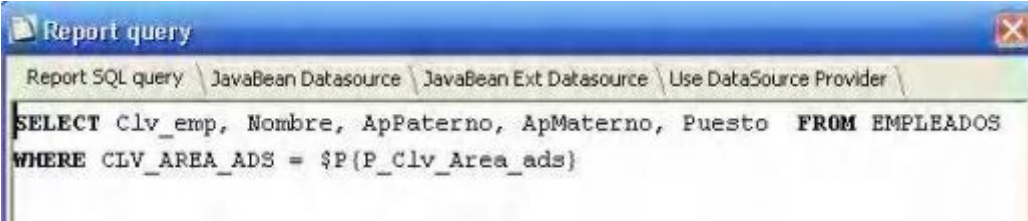
Clave	Nombre	Apellido Paterno	Apellido Materno	Puesto
144	BENITO	BAHENA Y	LOME	CAJERO GENERAL
2210	LUIS	LOPEZ	VARGAS	AUXILIAR ADMIVO. "A"
10877	FELIPE	RAYON	BELTRAN	OPERARIO 1º TALLER
10884	URBANO	RAMIREZ	CORTES	CHOFER NOCTURNO TALLER
10907	GABINO	BENITO	AGUIRRE	OPERADOR DE TROLEBUS
10917	JORGE DE JESUS	HERNANDEZ	ORTIZ	CHECADOR DE TIEMPO
10922	ARTURO	MENDOZA	LORENZO	OPERADOR DE TROLEBUS
10923	J. DOMINGO	PEREZ	MEDINA	OPERARIO 1RA. "B" CABO
10936	JOSE	SANCHEZ	FLORES	OPERARIO 1º TALLER
10967	ADOLFO	GARCIA	LARA	SUPERVISOR "B" DE OPERACION
10969	ALFONSO ARTEMIO	ROSAS	VILLALVA	OPERADOR DE TROLEBUS
10971	ANTONIO	CASILLAS	CORTES	OPERADOR DE TROLEBUS
10972	JOSE LUIS	RAMIREZ	AYALA	OPERADOR DE TROLEBUS
10978	HUMBERTO	CORDOBA	VEGA	OPERADOR DE TROLEBUS
11014	MARIO	OSORIO	ARENAS	SUPERV.DE TALLER AUTOMOTRIZ
11018	GUILLERMO	SOTO	GARCIA	OPERADOR DE TROLEBUS
11023	MANUEL	VEGA	CAZARES	OPERADOR DE TROLEBUS

El Viewer de JasperReports, muestra en su barra de herramientas la posibilidad de enviar el reporte directamente a la impresora o bien de guardar el reporte en algún formato de archivo específico: PDF, HTML, XLS, RTF entre otros. La funcionalidad de iReport en este caso es de oficina, obteniendo los datos almacenados en una base de datos y mostrándolos sin pasar a través de ninguna otra aplicación.

Llenar el reporte dinámicamente desde una aplicación Swing.

Algo que resultaría más interesante, es llenar el reporte dinámicamente desde alguna aplicación Java. En este caso, debe hacerse uso del archivo \*.jasper generado de la compilación del archivo xml.

Oprima el botón Database de la barra de herramientas. En la ventana Report Query, modifique el Query que se muestra en la pestaña Report SQL query, como se muestra a continuación:



The screenshot shows a window titled "Report query". It has a tabbed interface with "Report SQL query" selected. The query text is: `SELECT Clv_emp, Nombre, ApPaterno, ApMaterno, Puesto FROM EMPLEADOS WHERE CLV_AREA_ADS = $P{P_Clv_Area_ads}`.

```
Report query
Report SQL query | JavaBean Datasource | JavaBean Ext Datasource | Use DataSource Provider
SELECT Clv_emp, Nombre, ApPaterno, ApMaterno, Puesto FROM EMPLEADOS
WHERE CLV_AREA_ADS = $P{P_Clv_Area_ads}
```



Se ha modificado el query para que se muestre en el reporte, solo a aquellos empleados que pertenezcan a determinada área, dicha área se pasará como parámetro desde una aplicación Swing para llenar el reporte.

Debe agregarse este parámetro al reporte, para esto, oprima el botón Parameters de la barra de herramientas o bien, desde el menú ver, seleccione Parámetros de informe. En la pantalla valores, asegúrese de estar ubicado en la pestaña Parameters y oprima el botón Nuevo. Agregue el nuevo parámetro del reporte, como se muestra en la siguiente figura.



El tipo de parámetro se estableció como String aún cuando se sabe de las tablas que debería ser entero, en realidad esto funciona bien, pasando desde la aplicación java al reporte un String. El tipo de parámetro del reporte debe ser del mismo tipo al que se vaya a pasar desde sus aplicaciones en Java, de lo contrario obtendrá errores que le darán muchos dolores de cabeza. Recompile el proyecto, si se muestra un mensaje de error de compilación mencionando la ausencia del parámetro, vuelva a agregarlo y abra la ventana Report query para asegurarse que se muestran los campos de la consulta. Por alguna razón, la aplicación algunas veces no detecta el nuevo parámetro. El siguiente hilo es capaz de llenar y exportar el reporte realizado, solo ha de proporcionarse una conexión a la base de datos y la ruta del archivo Jasper.

```
import java.util.*; import java.sql.Connection;  
import java.awt.event.*;
```

```
/*Librerías necesarias para Jasper Reports*/
```

```
import net.sf.jasperreports.engine.*;
import net.sf.jasperreports.view.*;

public class cExport_thread extends Thread {

    cConnection conexion;

    public cExport_thread(String Clv_area) {
    }

    /**
     * Método del hilo */
    public void run(){
        try {
            //Ruta de Archivo Jasper
            String fileName="C:\\proyecto\\Reporte1.jasper";
            //Obtener una conexión a la base de datos
            conexion = new cConnection(); Connection con = conexion.mkConection();
            //Pasamos parametros al reporte Jasper.
            Map parameters = new HashMap(); parameters.put("P_Clv_Area_ads",Clv_area);
            //Preparacion del reporte (en esta etapa llena el diseño de reporte)
            //Reporte diseñado y compilado con iReport
            JasperPrint jasperPrint = JasperFillManager.fillReport(fileName,parameters,con);
            //Se lanza el Viewer de Jasper, no termina aplicación al salir
            JasperViewer jviewer = new JasperViewer(jasperPrint,false);
            jviewer.show();
        } catch (Exception j) {
            System.out.println("Mensaje de Error:"+j.getMessage()) } finally{
            conexion.closeConection();
        }
    }
}
```

La finalidad del hilo, en mi caso particular, fue para liberar de carga al evento de un botón en una aplicación Swing, dado que la obtención de la conexión y el llenado del reporte puede ser tardado. El hilo debe lanzarse de la siguiente manera:

```
cExport_thread thread_exp = new cExport_thread();
thread_exp.start();
```

Hasta esta fecha, han salido bastantes versiones de JasperReports e iReport, si utiliza versiones diferentes a las aquí utilizadas, asegúrese que coincidan las librerías de Jasper tanto en iReport como en su proyecto java con el que pretende llenar el reporte.

---

## CREACIÓN DE GRÁFICOS EN IREPORT

---

JasperReports no maneja directamente gráficos: estos deben crearse independientemente como imágenes, incluso utilizando una de las numerosas librerías de código libre disponibles para la creación de gráficos. La imagen producida será mostrada usando un componente de imagen. La idea es realmente simple, pero la creación de un gráfico en tiempo de ejecución requiere de un buen conocimiento de la programación de JasperReports, y muchas veces es necesario utilizar scriptlets capaces de coleccionar los datos que se mostrarán en el gráfico.

A partir de la versión 0.4.0, iReport proporciona una herramienta para simplificar la construcción de un gráfico. Esta herramienta permite crear un gráfico configurando propiedades y datos principales que serán vistos de manera simple por el usuario final.

La creación de un gráfico se basa en una librería muy conocida de código libre llamada JFreeCharts desarrollada por David Gilbert de Object Refinery Limited. iReport soporta por ahora solamente un pequeño número de tipos de gráficos presentados en JFreeCharts, y pueden modificarse solamente algunas de las propiedades del gráfico, pero es posible aún así, crear gráficos limpios con un gran impacto a la vista.

Fuente: Documentación y Tutoriales de iReport en: <http://ireport.sourceforge.net/>

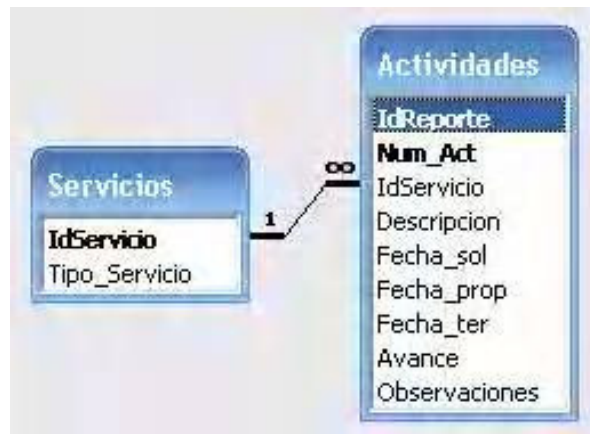
Después de la pequeña introducción, se mostrará la manera de crear un grafico en iReport.

La versión de iReport aquí utilizada será la 0.5.1, para versiones más recientes, la creación de los gráficos puede ser diferente; deben referirse a la documentación correspondiente de su versión.

Supondremos que ya se cuenta con una configuración correcta del iReport y sus librerías, especialmente la librería JFreecharts, la versión que se incluye con iReport 0.5.1, es la jfreechart-1.0.0-rc1.

Además, se necesitará tener configurada una conexión a una base de datos, y por supuesto datos para poder establecer una consulta que alimentará con datos al gráfico.

A partir de los datos de las siguientes tablas, se creará el gráfico en cuestión.



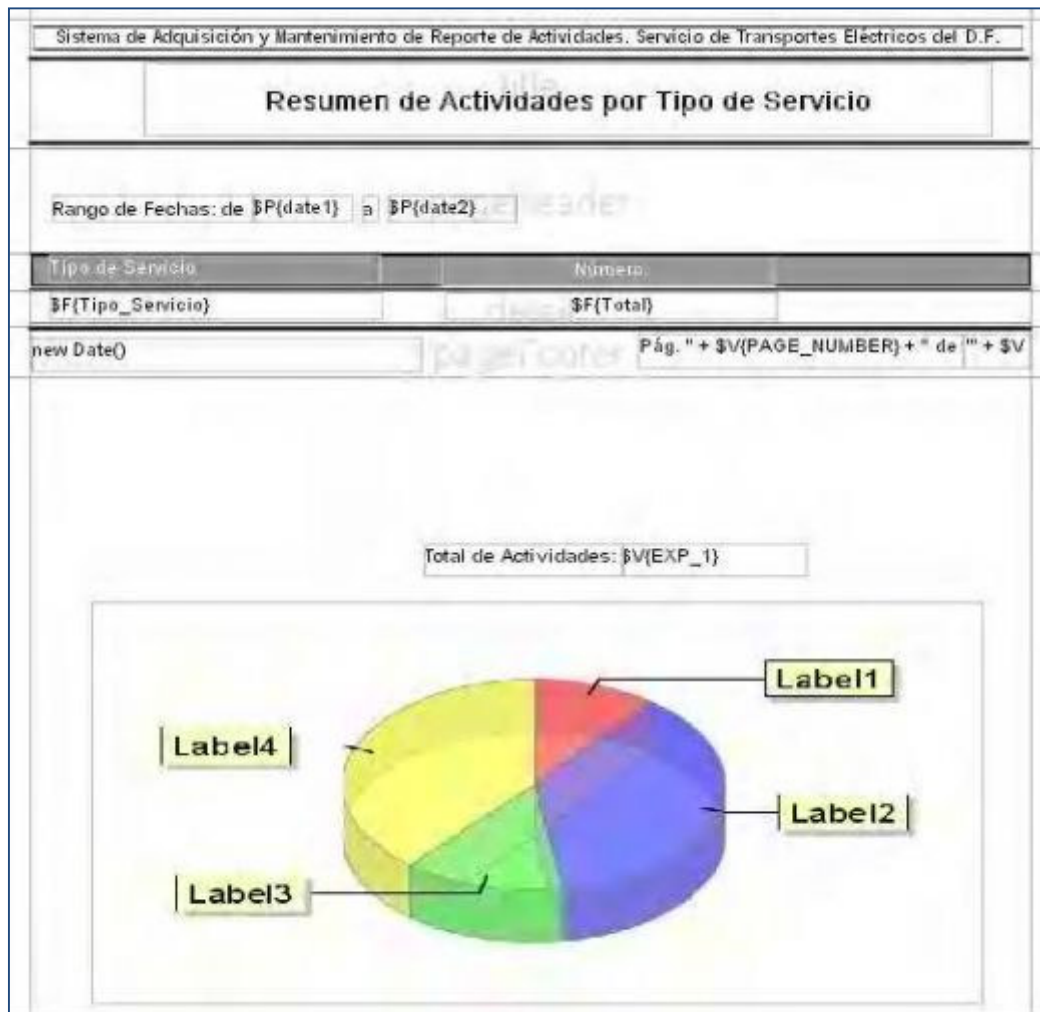
Las tablas anteriores muestran las actividades realizadas en una empresa. Cada una de las actividades se relaciona con un solo servicio. Cada actividad pertenece a un trabajador, que por simplicidad este campo no se muestra en este diseño de tablas.

Supongamos que se desea saber cuantas actividades se realizaron por servicio en un determinado rango de fechas, digamos en un mes. Se debe proceder a crear la consulta SQL correspondiente para obtener estos datos, más aún, podemos reflejarlos gráficamente, la consulta sería como la siguiente:

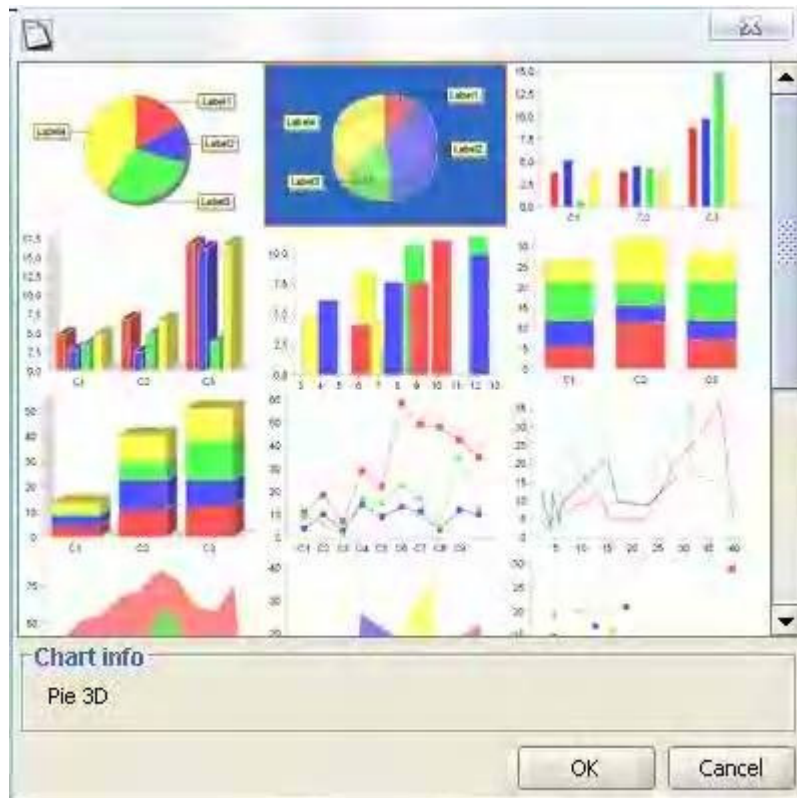
```
Report query
Report SQL query \ JavaBean Datasource \ JavaBean Ext Datasource \ Use DataSource Provider \
select Servicios.Tipo_Servicio,count(*) As Total
from Actividades,Servicios
where Actividades.IdServicio = Servicios.IdServicio
And Actividades.Fecha_ter >= #1/5/2005#
And Actividades.Fecha_ter <= #25/5/2005#
Group By Servicios.Tipo_Servicio
```

La consulta anterior agrupará las actividades realizadas por tipo de Servicio que se realizaron del 1 de mayo al 25 de mayo del 2005.

El diseño de reporte es el siguiente:

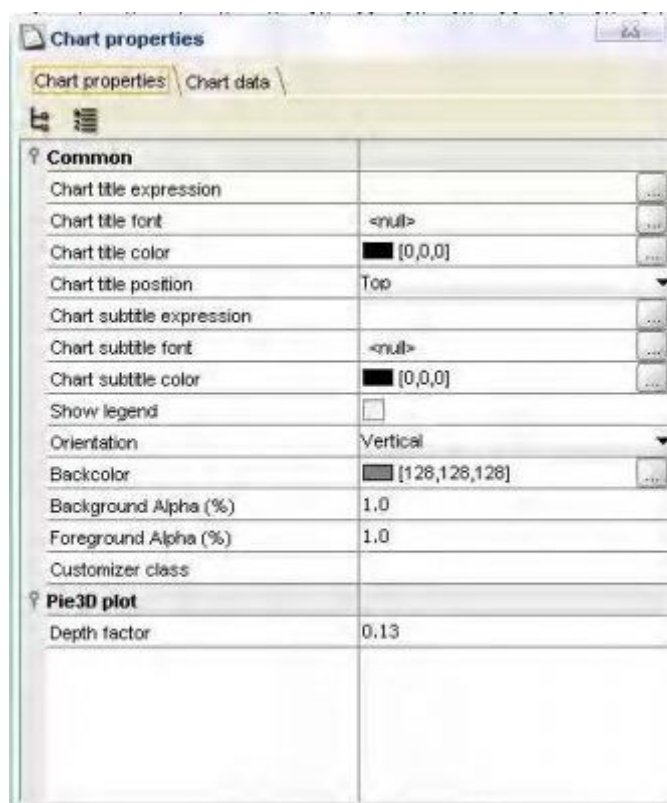


Para insertar un gráfico en un reporte de iReport, seleccione el objeto Chart tool de la barra de herramientas de iReport y dibuje el gráfico sobre la sección summary del reporte:



A continuación deberá seleccionar el tipo de gráfico deseado, para este ejemplo seleccionamos el tipo Pie 3D. Oprimir el botón OK y el componente que contendrá al gráfico ya ha sido creado, lo que resta es configurar las propiedades del mismo y los datos que mostrará.

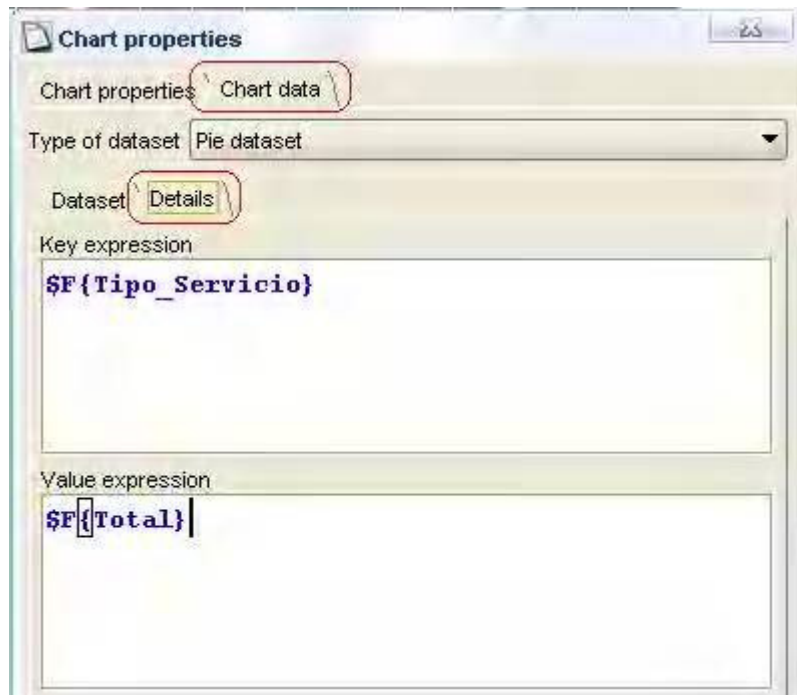
Haciendo doble click sobre el componente del gráfico, se mostrará la pantalla de propiedades de este, sitúese en la pestaña Chart, a continuación oprima el botón Edit chart properties, deberá aparecer una pantalla como la siguiente:



En la pestaña Chart properties de esta pantalla, se encuentran las propiedades que pueden modificarse para el gráfico en turno, como el nombre del gráfico, las fuentes, leyendas, ect. Puede dejar las propiedades actuales por ahora y modificarlas posteriormente. Ahora centraremos la atención en la manera de configurar los datos para el gráfico.

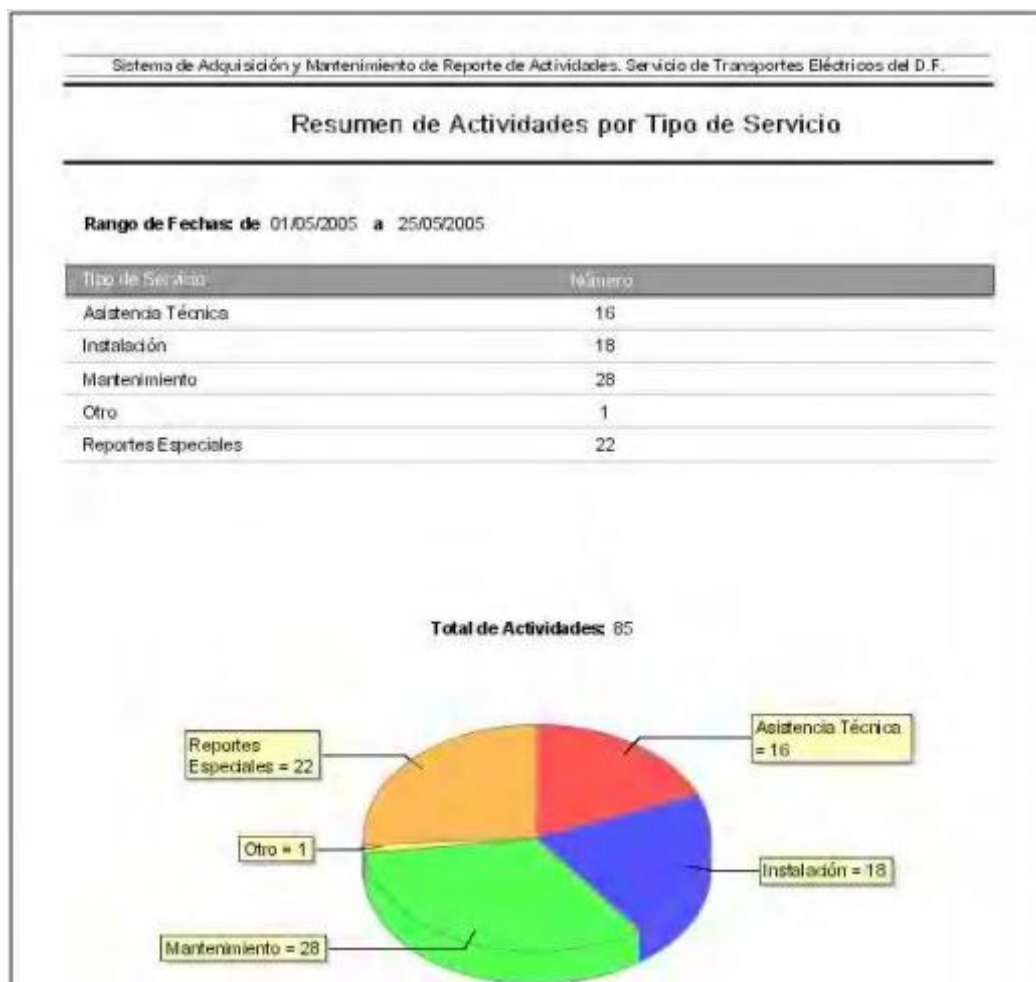
Debe situarse en la pestaña Chart data de la pantalla anterior y enseguida en la pestaña Details. En el apartado Key expression, coloque el campo `$F{Tipo_Servicio}` y en el apartado Value expresión coloque el campo `$F{Total}`. Estos campos son precisamente los que se obtienen de la consulta que se estableció anteriormente, estos a su vez se mostrarán en la sección detail del reporte.





Listo se han configurado los datos que necesita el gráfico. Al compilar y ejecutar el reporte con iReport el resultado puede ser parecido al siguiente:





Si se desea mostrar el gráfico anterior en tiempo de ejecución desde una aplicación Java, se procede de manera semejante a como se muestran otros reportes. En este caso deben determinarse los parámetros que se pasarán desde la aplicación java al reporte, por lo que la consulta se modifica como se muestra a continuación:

```

Report query
Report SQL query \ JavaBean Datasource \ JavaBean Ext Datasource \ Use DataSource Provider \
select Servicios.Tipo_Servicio,count(*) As Total
from Actividades,Servicios
where Actividades.IdServicio = Servicios.IdServicio
And Actividades.Fecha_ter >= $P{date1}
And Actividades.Fecha_ter <= $P{date2}
Group By Servicios.Tipo_Servicio

```

Lo que se pretende con la consulta anterior, es pasar como parámetros las fechas para las que se desea obtener el total de actividades agrupadas por servicio.

Para la versión 0.5.1 de iReport, solo debe compilarse el reporte para obtener el archivo Jasper que será llenado por la aplicación. En versiones anteriores o quizá recientes, es posible que se necesite incluir un scriptlet para coleccionar los datos del gráfico.

Contando con el archivo Jasper del reporte, la manera en que se manda llenar desde una aplicación Java es la misma que para otros reportes, como se muestra en el siguiente fragmento de código: ...

```
//Ruta de Archivo Jasper
String fileName="C:\\proyecto\\Grafico.jasper";
//Obtner una conexión a la base de datos
conexion = new cConnection(); Connection con = conexion.mkConection();
//Pasamos parametros al reporte Jasper.
Map parameters = new HashMap(); parameters.put("date1",p_date1);
parameters.put("date2",p_date2);
//Preparacion del reporte (en esta etapa llena el diseño de reporte) //Reporte diseñado y compilado
con iReport
JasperPrint jasperPrint = JasperFillManager.fillReport(fileName,parameters,con);
//Se lanza el Viewer de Jasper, no termina aplicación al salir
JasperViewer jviewer = new JasperViewer(jasperPrint,false); jviewer.show();
...
```

Puede exportar el reporte a diferentes formatos de archivo directamente desde la aplicación sin pasar por el JasperViewer, para esto puede referirse a las API's de su versión correspondiente.

# Capítulo 8

## Apache POI

### INTRODUCCIÓN

---

El proyecto Apache POI es el proyecto principal para el desarrollo de adaptaciones en Java puro de los archivos basados en Compound Document format (OLE2) de Microsoft. OLE2 lo utilizan los Documentos de Microsoft Office, así como por los programas que utilizan conjuntos de propiedades MFC para serializar sus objetos de tipo documento.

### LEER UN ARCHIVO EXCEL

---

Apache POI es una librería Java que se utiliza para poder interactuar con hojas de cálculos de Microsoft, en sus distintos formatos.

Vamos a ver el código que se necesita para poder leer los datos contenidos en un archivo excel.

Para ello lo que vamos a necesitar es:

- El fichero jar de Apache POI
- JDK 1.5 o superior
- Un editor de JAVA.

Vamos a ver las clases que utilizamos en el ejemplo para leer los datos de un archivo excel.

- **POIFSFileSystem:** Esta clase se utiliza para gestionar el ciclo de vida completo del sistema de archivos.
- **HSSFWorkbook:** Este es el primer objeto construido para escribir o leer un archivo de Excel.
- **HSSFSheet:** Esta clase se utiliza para crear hojas de cálculo.
- **HSSFRow:** Esta clase representa la fila de una hoja de cálculo.
- **HSSFCell:** Esta clase representa la celda en una fila de la hoja de cálculo.

El código que utilizamos es el siguiente:

```
package com.sample.excel;

import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

public class ExcelSheetReader {

    /**
     * This method is used to read the data's from an excel file.
     * @param fileName - Name of the excel file.
     */
    private void readExcelFile(String fileName) {

        /**
         * Create a new instance for cellDataList
         */
        List cellDataList = new ArrayList();
        try{
            /**
             * Create a new instance for FileInputStream class
             */
            FileInputStream fileInputStream = new FileInputStream(fileName);

            /**
             * Create a new instance for POIFSFileSystem class
             */
            POIFSFileSystem fsFileSystem = new POIFSFileSystem(fileInputStream);

            /**
             * Create a new instance for HSSFWorkbook Class
             */
            HSSFWorkbook workBook = new HSSFWorkbook(fsFileSystem);
            HSSFSheet hssfSheet = workBook.getSheetAt(0);

            /**
```

```

    * Iterate the rows and cells of the spreadsheet
    * to get all the datas.
    */
    Iterator rowIterator = hssfSheet.rowIterator();
    while (rowIterator.hasNext()) {
        HSSFRow hssfRow = (HSSFRow) rowIterator.next();
        Iterator iterator = hssfRow.cellIterator();
        List cellTempList = new ArrayList();
        while (iterator.hasNext()) {
            HSSFCell hssfCell = (HSSFCell) iterator.next();
            cellTempList.add(hssfCell);
        }
        cellDataList.add(cellTempList);
    }
} catch (Exception e) {
    e.printStackTrace();
}

/**
 * Call the printToConsole method to print the cell data in the
 * console.
 */
printToConsole(cellDataList);

}

/**
 * This method is used to print the cell data to the console.
 * @param cellDataList - List of the data's in the spreadsheet.
 */
private void printToConsole(List cellDataList) {
    for (int i = 0; i < cellDataList.size(); i++) {
        List cellTempList = (List) cellDataList.get(i);
        for (int j = 0; j < cellTempList.size(); j++) {
            HSSFCell hssfCell = (HSSFCell) cellTempList.get(j);
            String stringCellValue = hssfCell.toString();
            System.out.print(stringCellValue + "\t");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
```

```
String fileName = "C:" + File.separator + "Users" +  
File.separator + "Giftsam" + File.separator + "Desktop" +  
File.separator + "sampleexcel.xls";  
new ExcelSheetReader().readExcelFile(fileName);  
}  
  
}
```

## CREAR UN ARCHIVO EXCEL

Al principio, descargas los jars correspondientes a la última versión desde la web de Apache POI de la página oficial: <http://poi.apache.org>.

Copias los jars en la carpeta de librerías: WEB-INF/lib.

Es importante tener en cuenta que cada archivo de Excel representa un LIBRO, dentro de cada libro tenemos HOJAS, dentro de cada HOJA tenemos FILAS, y, finalmente, en cada FILA tenemos CELDAS. Entender esto nos ayudará a ver cómo se organiza la información en el archivo.

Primero, necesitamos crear un LIBRO haciendo uso de la clase *HSSFWorkbook*:

```
// Procesar información y generar el xls.  
HSSFWorkbook objWB = new HSSFWorkbook();
```

A continuación, creamos la hoja con la clase *HSSFSheet*:

```
// Crear la hoja  
HSSFSheet hoja1 = objWB.createSheet("hoja 1");
```

Luego, creamos la fila con *HSSFRow*:

```
// Crear la fila.  
HSSFRow fila = hoja1.createRow((short)1);
```

Notemos que el valor que se envía al método encargado de crear las filas es de tipo short, el mismo que indica el número correspondiente a la fila que hemos de trabajar. El índice de las filas empieza en "0", aunque ello no nos impide trabajar directamente con otras filas.

Una vez creada la fila, empezamos a trabajar con las celdas.

```
// Aunque no es necesario podemos establecer estilos a las celdas.  
// Primero, establecemos el tipo de fuente  
HSSFFont fuente = objLibro.createFont();  
fuente.setFontHeightInPoints((short)11);
```

```
fuelle.setFontName(fuelle.FONT_ARIAL);
fuelle.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);

// Luego creamos el objeto que se encargará de aplicar el estilo a la celda
HSSFCellStyle estiloCelda = objLibro.createCellStyle();
estiloCelda.setWrapText(true);
estiloCelda.setAlignment(HSSFCellStyle.ALIGN_JUSTIFY);
estiloCelda.setVerticalAlignment(HSSFCellStyle.VERTICAL_TOP);
estiloCelda.setFont(fuelle);

// También, podemos establecer bordes...
estiloCelda.setBorderBottom(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setBottomBorderColor((short)8);
estiloCelda.setBorderLeft(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setLeftBorderColor((short)8);
estiloCelda.setBorderRight(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setRightBorderColor((short)8);
estiloCelda.setBorderTop(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setTopBorderColor((short)8);

// Establecemos el tipo de sombreado de nuestra celda
estiloCelda.setFillForegroundColor((short)22);
estiloCelda.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);

// Creamos la celda, aplicamos el estilo y definimos
// el tipo de dato que contendrá la celda
HSSFCell celda = objFila.createCell((short)0);
celda.setCellStyle(estiloCelda);
celda.setCellType(HSSFCell.CELL_TYPE_STRING);

// Finalmente, establecemos el valor
celda.setCellValue("Un valor");
```

Como podemos apreciar en el código tenemos la posibilidad de establecer estilos mediante las clases *HSSFFont* y *HSSFCellStyle*.

La primera, nos permite establecer el tipo de fuente que se empleará para la celda que hemos de utilizar. Para ello, contamos con los métodos *setPointHeightInPoints* que recibe un valor de tipo *short* que representa el tamaño de la fuente; el método *setFontName* el mismo que recibe una constante de la misma clase que nos permite establecer la fuente que se ha de emplear, y, otros métodos como: *setBoldweight* y *setUnderline*, entre otros, que nos permitirán aplicarle otros estilos y efectos al valor que ocupe nuestra celda.

La segunda, es la clase que, finalmente, nos ayudará a aplicar el estilo a la celda. Podemos acomodar y alinear el texto mediante los métodos *setWrapText*, *setAlignment* y *setVerticalAlignment*; aplicar la fuente trabajada, con el método *setFont*; configurar los bordes mediante los métodos: *setBorderBottom*, *setBorderLeft*, *setBorderRight*, *setBorderTop*, para el tipo; y, *setBottomBorderColor*, *setLeftBorderColor*, *setRightBorderColor*, *setTopBorderColor* para establecer el color de los bordes; y, establecer el sombreado de las celdas mediante los métodos *setFillForegroundColor*, *setFillBackgroundColor* y *setFillPattern*.

Aunque, es un poco engorroso trabajar estos estilos, celda por celda, de esta forma, lo mejor es encapsular todo este proceso en métodos que nos permitan ahorrar líneas de código, preestableciendo, los estilos que se emplearán.

Según la versión de la librería que se esté empleando, podremos contar o no, con algunas constantes para la configuración del color y el establecimiento de los sombreados.

Finalmente, volcamos nuestro libro a un archivo de la siguiente forma:

```
// Volcamos la información a un archivo.
String strNombreArchivo = "C:/libro1.xls";
File objFile = new File(strNombreArchivo);
FileOutputStream archivoSalida = new FileOutputStream(objFile);
objWB.write(archivoSalida);
archivoSalida.close();
```

A continuación les presento el código completo:

```
// Procesar la información y generar el xls
HSSFWorkbook objWB = new HSSFWorkbook();

// Crea la hoja
HSSFSheet hoja1 = objWB.createSheet("hoja 1");

// Procesar la información y genero el xls.
HSSFRow fila = hoja1.createRow((short)1);

// Aunque no es necesario podemos establecer estilos a las celdas.
// Primero, establecemos el tipo de fuente
HSSFFont fuente = objLibro.createFont();
fuente.setFontHeightInPoints((short)11);
fuente.setFontName(fuente.FONT_ARIAL);
fuente.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);

// Luego se crea el objeto que se encargará de aplicar el estilo a la celda
```



```
HSSFCellStyle estiloCelda = objLibro.createCellStyle();
estiloCelda.setWrapText(true);
estiloCelda.setAlignment(HSSFCellStyle.ALIGN_JUSTIFY);
estiloCelda.setVerticalAlignment(HSSFCellStyle.VERTICAL_TOP);
estiloCelda.setFont(fuente);

// También se puede establecer bordes
estiloCelda.setBorderBottom(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setBottomBorderColor((short)8);
estiloCelda.setBorderLeft(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setLeftBorderColor((short)8);
estiloCelda.setBorderRight(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setRightBorderColor((short)8);
estiloCelda.setBorderTop(HSSFCellStyle.BORDER_MEDIUM);
estiloCelda.setTopBorderColor((short)8);

// Establecer el tipo de sombreado de la celda
estiloCelda.setFillForegroundColor((short)22);
estiloCelda.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);

// Crear la celda, se aplica el estilo y se define
// el tipo de dato que contendrá la celda
HSSFCell celda = objFila.createCell((short)0);
celda.setCellStyle(estiloCelda);
celda.setCellType(HSSFCell.CELL_TYPE_STRING);

// Finalmente se establece el valor
celda.setCellValue("Un valor");

// Se vuelca la información al un archivo.
String strNombreArchivo = "C:/libro1.xls";
File objFile = new File(strNombreArchivo);
FileOutputStream archivoSalida = new FileOutputStream(objFile);
objWB.write(archivoSalida);
archivoSalida.close();
```

Aunque el ejemplo es un tanto sencillo, todo lo presentado aquí generalmente se combina con arreglos de beans y bucles, los cuales nos ayudarán a presentar más datos en nuestras hojas de Excel.

El objetivo es presentar un pequeño caso práctico sobre el uso de HSSF (POI) para la generación de archivos en formato Excel.