Escribir Aplicaciones Avanzadas para la Plataforma Java™

Como desarrollador experimentado sobre la plataforma Java ™, indudablemente sabes lo rápido que evoluciona esta plataforma. Sus muchos Interfaces de Programación de Aplicaciones (APIs) proporcionan una gran cantidad de funcionalidades para todos los aspectos de la programación de aplicaciones y nivel de sistema. Los desarrolladores del mundo real nunca usan uno o dos APIs para resolver un problema, sino que juntan la funcionalidad clave de la expansión de varios APIs. Conocer los APIs que necesitas, qué parte de esos APIs necesitas, y como los APIs funcionan juntos para crear la mejor solución puede ser una tarea intimidatoria.

Para ayudarte a navegar por los APIs de Java y acortar el tiempo de desarrollo de tus proyectos, esta sección incluye las fases de diseño, desarrollo, prueba y despliegue para una aplicación de subastas. Durante esta aplicación de ejemplo, no cubriremos cada posible escenario de programación, explora muchas situaciones comunes y la discusión te dejará con una metodología sólida para diseñar y construir tus propias soluciones.

Esta sección es para desarrolladores con un nivel un poco alto de entendimiento de la escritura de programas Java. El ejemplo está escrito con los APIs de la plataforma Java® 2 y explica en términos de funcionalidad el cómo y el por qué, por eso, si necesitas ayuda para instalar la plataforma Java, configurar tu entorno, o conseguir que tu primera aplicación funcione, primero deberías leer las secciones anteriores de este tutor...

Índice de Contenidos

Concordar los Requerimientos del Proyecto con la Tecnología

El desafío de escribir un libro sobre el desarrollo de una aplicación avanzada para la plataforma Java™ es encontrar un proyecto lo suficientemente pequeño, pero al mismo tiempo, los suficientemente completo para garantizar las tecnicas de programación avanzadas.

El proyecto presentado en este libro es una casa de subastas basada en web. La aplicación está inicialmente escrita para la plataforma Enterprise JavaBeans™. En los capítulos posteriores expandieremos el corazón del ejemplo descrito aquí añadiendo funcionalidades avanzadas, mejoras y soluciones alternativas a algunas de las cosas que obtendrás gratis cuando use la plataforma Enterprise JavaBeans.

Para mantener la explicación sencilla, la aplicación de ejemplo sólo tiene un conjunto básico de transaciones para poner y pujar ítems a subasta. Sin embargo, la aplicación escala para manejar múltiples usuarios, proporciona un entorno de tres filas basado en transaciones, controla la seguirad, e integra sistemas basados en la legalidad. Este capítulo cubre cómo determinar los requerimientos del proyecto y el modelo de aplicación -- pasos importantes que siempre deberían realizarse antes de empezar a codificar.

¿Tienes Prisa?

Esta tabla te enlaza directamente con los tópicos específicos.

Tópico	Sección
Demostración de Subasta	La Subasta de Duke
Requerimientos del Proyecto	Entrevista Base Modelar el Proyecto
Modelado	La Casa Identifica Compradores y Vendedores La Casa Determina la Mayor Puja La Casa Notifica a Compradores y Vendedores Alguien Busca un Ítem Alguien Ve un Ítem en Venta Alguien Ve los Detalles de un Ítem El Vendedor Pone un Ítem en Venta El Compador Puja por Ítems Diagrama de Actividad

---

Requerimientos del Proyecto y Modelado

El primer paso para determinar los requerimientos del proyecto es la entrevista con el usuario base para saber que se espera de una subasta on-line. Este es un paso importante, y no puede pasarse por alto porque es una base sólida de información que nos ayudará a definir las capacidades clave de nuestra aplicacion.

El capítulo 2 pasea a través del código de la aplicación, explica como trabaja la plataforma Enterprise JavaBeans, y nos cuenta cómo ejecutar una demostración en vivo. Si nunca has visto o usado una subasta on-line, aquí hay una maqueta de las páginas HTML de la aplicación de ejemplo.

Entrevista al usuario Base

Por la dicusión y por mantener las cosas sencillas, esta explicación asume que en las entrevistas con los usuarios base se encontraron los siguientes requerimientos para la casa de subastas:

Requerimientos de la Casa de Subastas

- Información Necesaria del Comprador y Vendedor
- Notas de Vendedores para postear ítems
- Grabar e Informar de las transaciones diarias

Requerimientos del usuario

- Pujar por o Vender un ítem
- Buscar o ver ítems en venta
- Notificar las ventas al vendedor y al comprador

Modelo de Proyecto

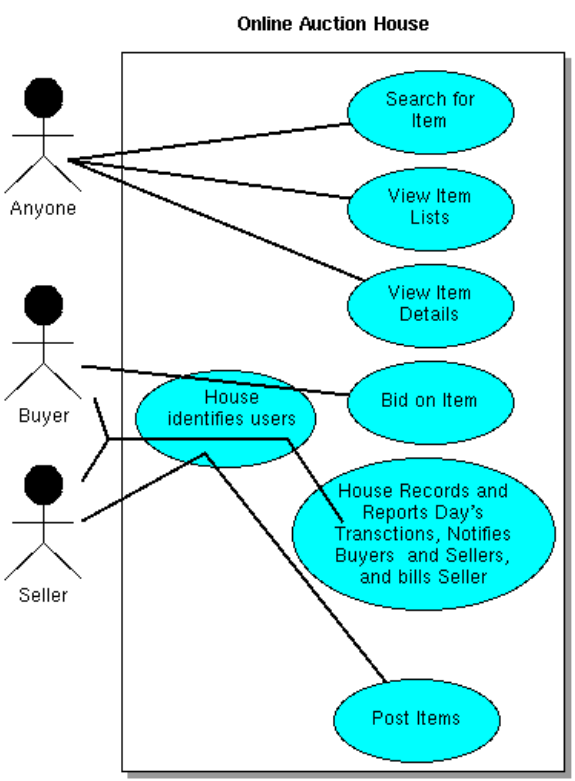
Después de analizar los requerimientos, podemos construir un diagrama de flujo de la aplicación para obtener un mejor entendimiento de los elementos necesarios en la aplicación y cómo interactua.

Un diagrama de flujo muestra la relación entre los actores y procesos dentro del sistema. Un proceso es una función única en un sistema, y un actor es la persona o software que realiza la acción o proceso. Por ejemplo, un comprador es el actor que realiza la función (proceso) de pujar por un ítem de la subasta, y el vendedor es el actor que realiza el proceso de postear un ítem para su subasta.

Aunque, no todos los actores son personas. Por ejemplo, el software es el actor que determina cuando un ítem se ha cerrado, encuentra la puja más alta, y notifica la venta al comprador y al vendedor.

El Unified Modeling Language (UML) es la herramienta que se usa para los diagramas de procesos. El siguiente diagrama usa UML para describir los procesos del comprador y del vendedor para una aplicación de subasta on-line.

En UML, los sistemas se agrupan en cuadrados, los actores se representas por figuras humanas, los procesos se denotan mediante óvalos, y las líneas muestran como los actores usan el sistema.


Online Auction House
Anyone
Buyer
Seller
Search for Item
View Item Lists
View Item Details
Bid on Item
House identifies users
House Records and Reports Day's Transctions, Notifies Buyers and Sellers, and bills Seller
Post Items

La siguiente descripción define el proyecto. Estas descripciones no son parte del UML, pero son una herramienta útil para la definición de proyectos.

La Casa Identifica a Compradores y Vendedores

Una aplicación de subastas es usada por compradores y vendedores. Un comprador necesita saber quíen es el vendedor a quien tiene que pagarle, y el vendedor necesita conocer a los compradores para responder a sus preguntas sobre el producto y para finalizar la venta. Por eso, para postear o pujar por un ítem de la subasta, los compradores y vendedores necesitan estar registrados. El registro necesita obtener la siguiente información sobre los compradores y vendedores:

- User ID y password para comprar y vender.
- Dirección de E-mail para que pueda comunicarsele la puja más alta cuando se cierre la subasta.
- Información de la tarjeta de crédito para que la casa de subastas pueda cobrar al vendedor por listar sus ítems.

Una vez registrado, el usuario puede postear o pujar por un ítem en venta.

La Casa Determina la Puja más alta

La aplicación de subastas hace consultas a la base de datos y graba e informa de las transaciones diarias. La aplicación busca ítems que se han cerrado y determina la puja más alta.

La Casa Notitica a los Compradores y Vendedores

La aplicación subasta usa el e-mail para notificar al que ha pujado más alto y al vendedor, y cobrarle al vendedor por los servicios.

Alguien Busca un Ítem

Los compradores y vendedores introducen un string de búsqueda para localizar todos los ítems en subasta de la base de datos.

Alguien Ve los Ítems en Venta

Para popularizar la subasta y conseguir nuevos vendedores y compradores, la aplicación pemite que cualquiera vea los ítems de la subasta sin requerir que esté registrado. Para hacer esto sencillo, la subasta permite que cualquiera vea una lista de los ítems de alguna de estas tres formas:

- Todos los ítems en subasta.
- Nuevos ítems listados hoy.
- Ítems que se cierran hoy.

Alguien Ve los Detalles de un Ítem

La lista sumarizada enlaza con la siguiente información detallada de cada ítem. Esta información está disponible para cualquiera sin necesidad de identificación.

- Sumario del Ítem.
- Número del ítem en la subasta.
- Precio Actual
- Número de pujas
- Fecha de puesta en subasta
- Fecha de cierre del ítem
- ID del vendedor
- Puja más alta
- Descripción del ítem

El Vendedor Postea Ítems para su Venta

Para postear un ítem para su venta, un vendedor necesita identificarse a sí mismo y describir el ítem, de esta forma:

- User ID y password para la identificación del vendedor
- Descripción sumaria de ítem
- Precio de puja inicial
- Descripción detallada del ítem
- Número de días que el ítem estará en la subasta

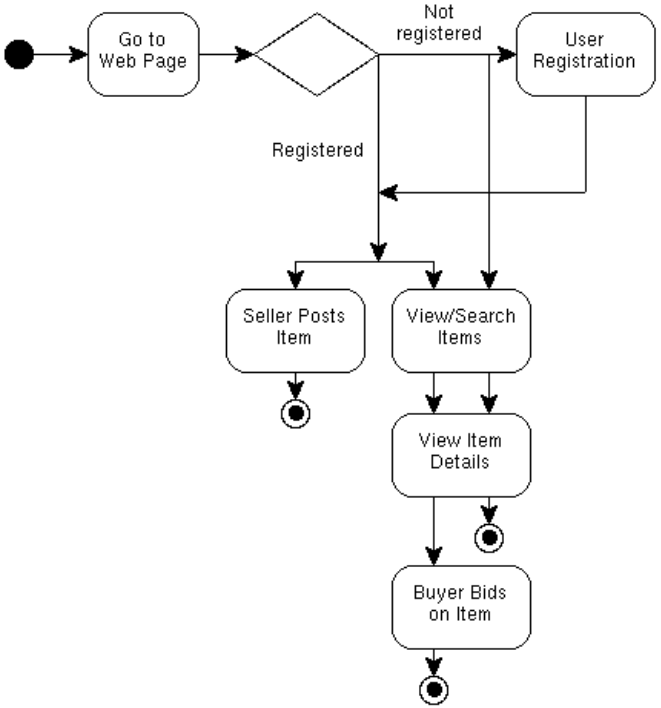
El Comprador Puja por Items

Una página de sumario detallado por cada ítem permite a los usuarios registrados identificarse a sí mismos y pujar por el ítem proporcionando la siguiente información:

- User ID
- Password
- Cantidad de la Puja

Diagrama de Actividad

El diagrama de actividad muestra el flujo de tareas dentro de la casa de subastas como una totalidad. Este diagrama muestra la aplicación subasta. El círculo negro de la izquierda muestra el principio de las actividades, y el círculo blanco punteado en el centro denota donde terminan las actividades.



Ozito

Elegir el Software

Con la aplicación modelada y los requerimientos del proyecto definidos, es hora de pensar en los APIs de Java™ que vamos a usar. La aplicación está claramente basada en cliente y servidor porque queremos acomodar desde 1 hasta n compradores, vendedores y mirones al mismo tiempo. Como el registro de los datos de los ítems en subasta deben almacenarse y recuperarse de alguna manejar, necesitares el API para acceder a bases de datos.

Los APIs de Java™

El corazón de la aplicación se puede crear de muchas formas usando uno de los siguientes aPIs:

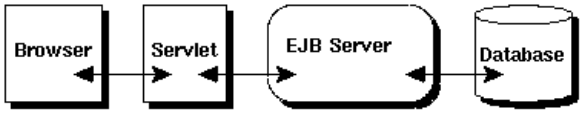
1. APIs de Sockets, multithreads y JDBC™.
2. APIs de Remote Method Invocation (RMI) y JDBC.
3. Plataforma Enterprise JavaBeans™.

Enterprise JavaBeans proporciona una forma sencilla de crear aplicaciones de clientes multi hilos porque maneja transaciones y control de estado, multithreads, recursos y otros complejos detalles de bajo nivel. La forma más sencilla para codificar la aplicación de subastas es con la plataforma Enterprise JavaBeans.

El capítulo 2 explica el código del corazón de la aplicación y como configurar y ejecutar el ejemplo.

Código de la Aplicación de la Casa de Subastas

La aplicación de ejemplo es una casa de subastas basada en el Web y escrita para la plataforma Enterprise JavaBeans™. El interface de usuario es un conjunto de páginas **HTML** que obtienen la entrada del usuario y le muestran la información. Detrás de las páginas **HTML** hay un servelt que pasa datos entre el navegador y el servidor Enterprise JavaBeans. Este servidor maneja la lectura y escritura de la base de datos.



Este capítulo describe el código de la aplicación, cómo funciona con el servidor Enterprise JavaBeans, y dónde obtener este servidor para ejcutar el ejemplo. O, si lo prefieres, aquí hay una maqueta de la aplicación subasta.

¿Tienes Prisa?

Esta tabla contiene enlaces directos a los tópicos específicos.

Tópico	Sección
Una Aplicación Multi-Hilo con Enterprise Beans	El Enterprise Beans Definido Beans de Entidad y de Sesión La Casa de Subastas Funciona Desarrollar y Ejecutar Aplicaciones ¿Cómo funcionan las aplicaciones Multi-Hilo?
Beans de Entidad y de Sesión	El servelt Auction Beans Entity Beans Session Clases Contenedor
Examinar un Bean de Contenedor Controlador	Variables Miembro Método Create Métodos de Contexto de Entidad Método Load Método Store Guardar la Conexión Descriptor de Desarrollo
Métodos del Buscador de Contenedo Controlador	AuctionServlet.searchItems BidderBean.getMatchingItemsList AuctionItemHome.findAllMatchingItems AuctionItemBean Deployment Descriptor

---

Un Aplicación Multi-Fila con Beans de Enterprise

La proliferación de aplicaciones basadas en internet - e intranet - ha creado una gran necesidad de aplicaciones transacionales distribuidas que aumente la velocidad, seguridad y rendimiento de la tecnología del lado del servidor. Una forma de conseguir estas necesidades es usar un modelo multi-fila donde una pequeña aplicación cliente invoca lógica de negocio que se ejecuta en el servidor.

Normalmente, las pequeñas aplicaciones clientes multi-hilo son dificiles de escribir porque se involucran muchas líneas de código intrincado para manejar la transación, el control de estados, multithreads, solape de recursos y otros detalles complejos de bajo nivel. Y para rematar estas dificultades, tenemos que retrabajar este código cada vez que escribamos una aplicación porque es tan de bajo nivel que no es reutilizable.

Si pudieramos usar un código de manejo de transaciones preconstruido por alguien o incluso si puedieramos reutilizar algo de nuestro propio código, ahorrariamos mucho tiempo y energía que podríamos utilizar para resolver otros problemas. Bien, la tecnología Enterprise JavaBeans™ puede darnos la ayuda necesaria. Esta tecnología hace sencillas de escribir las aplicaciones transacionales distribuidas porque separa los detalles de bajo nivel de la lógica del negocio. Nos concentramos en crear la mejor solución para nuestro negocio y dejamos el resto a la arquitectura oculta.

Este capítulo describe cómo crear la aplicación de subastas del ejemplo usando los servicios proporcionados por la plataforma Enterprise JavaBeans. En los siguientes capítulos veremos como podemos personalizar estos servicios e integrar estas características en aplicaciones existentes no EJB.

Enterprise Beans Definidos

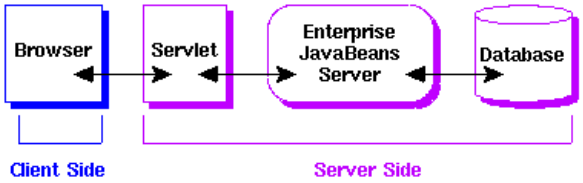
Un Bean Enterprise es una clase que proporciona dos tipos de métodos: lógica de negocio y ciclo de vida. Un programa cliente llama a los métodos de la lógica de negocio para interactuar con los datos contenidos en el servidor. El contenedor llama a los métodos de ciclo de vida para manejar el Bean en el servidor. Además de estos dos tipos de métodos, un Bean Enterprise tiene un fichero de configuración asociado, llamado un descriptor de desarrollo, se usa para configurar el Bean en el momento del desarrollo.

Así como es el responsable de la creacción y borrado de Beans, el servidor de JavaBeans de Enterprise también maneja las transaciones, la concurrencia, la seguridad y la persistencia de datos. Incluso las conexiones entre el cliente y el servidor se proporcionan usando los APIs de RMI y JNDI y opcionalmente los servidores pueden proporcionar escalabilidad a través del manejo de threads.

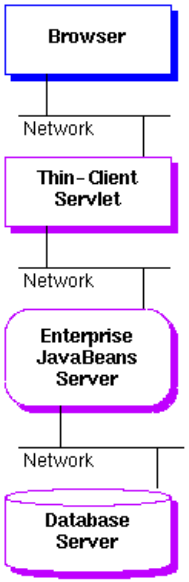
El ejemplo de la casa de subastas implementa una completa solución de JavaBeans de Enterprise que sólo proporcionan la lógica de negocio y usa los servicios ocultos proporcionados por la arquitectura. Sin embargo, podríamos encontrar que el servicio de contenedores controladores, aunque proporcionando una máxima portabilidad, no consigue todos los requerimientos de nuestra aplicación. En los próximos capítulos veremos cómo proporcionar estos servicios a nuestro Bean y también como usar estos servicios en aplicaciones que no usen Beans de Enterprise.

Pequeño Programas Cliente

Un pequeño cliente es un programa cliente que invoca a la lógica de negocio que se ejecuta en el servidor. Se llama "pequeño" porque la mayoría del proceso sucede en el servidor. En la siguiente figura, el servlet es el cliente. Invoca a los Beans Enterprise que se ejecutan sobre un servidor de JavaBeans Enterprise. También ejecuta la lógica que crea las páginas web que aparecen en el navegador.


Browser
Servlet
Enterprise
JavaBeans
Server
Database
Client Side
Server Side

Arquitectura Multi-Fila



arquitectura multi-fila o arquitectura de tres filas desciende del modelo estándard de dos filas de cliente y servidor situando una aplicación multi-fila entre el cliente y la base de datos.

Los programas clientes se comunican con la base de datos a través de la aplicación del servidor usando llamadas de alto nivel e independientes de la plataforma. La aplicación servidor responde a las peticiones del cliente, hace las llamadas necesarias a la base de datos dentro de la base de datos oculta, y responde al programa cliente de la forma apropiada.

El ejemplo de casa de subastas basado en web de tres filas consiste en el servlet cliente, el servidor Enterprise JavaBeans (la aplicación servidor), y el servidor de la base de datos como se ve en la figura.

Beans de Entidad y de Sesión

Existen dos tipos de Beans Enterprise: Beans de entidad y de sesión. Un Bean Enterprise que implementa una entidad de negocio es un **Bean de Entidad**, y un Bean Enterprise que implementa una tarea de negocio es un **Bean de Sesión**.

Típicamente, un Bean de entidad representa una fila de datos persistentes almacenados en una tabla de la base de datos. En el ejemplo de la casa de subastas, **RegistrationBean** es un Bean de entidad que representa los datos de un usuario registrado, y **AuctionItemBean** es un Bean de entidad que represena los datos de un ítem de la subasta. Los Beans de entidad son transacionales y de larga vida. Mientras que los datos permanezcan, el Bean de entidad puede acceder y actualizarlos. Esto no significa que tengamos un Bean ejecutándose por cada fila de la tabla. Si no que los Beans Enterprise se cargan y graban cuando es necesario.

Un Bean de sesión podría ejecutar una lectura o escritura en la base de datos, pero no es necesario. Un Bean de sesión podría invocar llamadas al JDBC por sí mismo o podría usar un Bean de entidad para hacer la llamada, en cuyo caso el Bean de sesión es un cliente del Bean de entidad. Un campo de Bean contiene el estado de la conversación y son temporales. Si el servidor o el cliente se bloquean, el Bean de sesión se vá. Frecuentemente se usan los Beans de sesión con uno o más Beans de entidad y para operaciones complejas con datos.

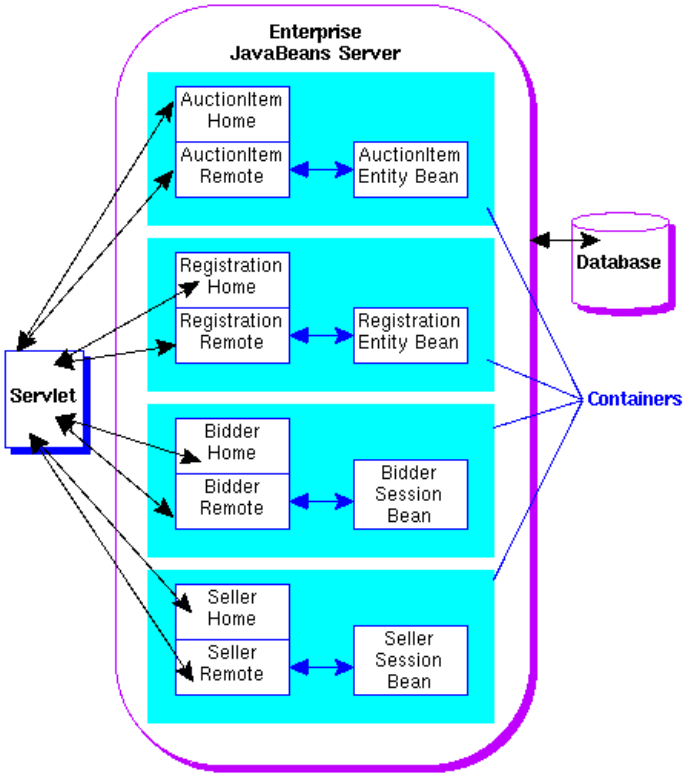
Beans de Sesión	Beans de Entidad
Campos que contienen el estado de la conversación	Representan datos de la base de datos
Manejan accesos a la base de datos por parte del cliente	Comparten accesos entre múltiples usuarios
La vida del cliente es la vida del Bean	Pesiste mientras existan los datos
Pueden perderse con la transación	Transacional
No sobrevive a las caídas del servidor	Sobrevive a las caídas del servidor
No maneja los datos de forma fina	Manejo de datos de forma delicada

La Casa de Subastas Funciona

El diagrama muestra los Beans de Enterprise para la aplicación de la casa de subastas y su relación con el servidor de JavaBeans de Enterprise. El cliente invoca la lógica de negocio en cuatro Beans de Enterprise a través de sus interfaces home y remoto. El servidor JavaBeans de este ejemplo maneja los detalles de bajo nivel incluyendo las operaciones de lectura y escritura en la base de datos.

Los cuatro Beans del ejemplo son:

- **AuctionItemBean** un Bean de entidad que mantiene información sobre el ítem de la subasta.
- **RegistrationBean** un Bean de entidad que almacena información de registro de los usuarios.
- **BidderBean** un Bean de sesión que usa **AuctionItemBean** para recuperar una listra de los ítems de la subastas, sólo los nuevos ítems, ítems cerca del cierre, e ítems cuyo sumario corresponde con una cadena de busqueda en la base de datos. También comprueba la identidad del usuario y la password cuando alguien hace una puja, y almacena las nuevas pujas en la base de datos.
- **SellerBean** es un Bean de sesión que usa **RegistrationBean** para comprobar la identidad del usuario y la password cuando alguien postea un ítem para su subasta, y **AuctionItemBean** para añadir nuevos ítems a la base de datos de la subasta.


Enterprise
JavaBeans Server
AuctionItem
Home
AuctionItem
Remote
AuctionItem
Entity Bean
Registration
Home
Registration
Remote
Registration
Entity Bean
Bidder
Home
Bidder
Remote
Bidder
Session
Bean
Seller
Home
Seller
Remote
Seller
Session
Bean
Database
Containers
Servlet

Como se ve en la figura superior, un Bean de entidad o de sesión realmente es una colección de clases e interfaces. Todos los Beans de entidad y de sesión consisten en un interfae remoto, un interface home, y la clase del Bean. El servidor busca el interface home del Bean que está ejecutándose en el servidor JavaBeans de Enterprise, lo usa para crear el interface remoto, e invoca a los métodos del Bean a través del interface remoto.

- Un Interface remoto de un Bean Enterprise decribe los métodos del Bean, o qué hace el Bean. Un programa cliente u otro Bean Enterprise llama a los métodos definidos en el interface remoto para invocar la lógica de negocios implementada por el Bean.
- Un interface home de un Bean de Enterprise describe cómo un programa cliente u otro Bean Enterprise crea, encuentra (sólo los Beans de entidad), y elimina ese Bean de Enterpise de su contenedor.
- El contenedor, mostrado en cyan en el gráfico, proporciona el interface entre el Bean Interface y las funcionalidades de bajo nivel específicas de la plataforma que soporta el Bean.

Desarrollar y Ejecutar Aplicaciones

Las herramientas de desarrollo y un servidor de JavaBeans Enterprise es esencial para ejecutar aplicaciones con JavaBeans Enterprise. Las herramientas de desarrollo generan contenedores, que son clases que proporcionan un interface de implementaciones de bajo nivel en un servidor JavaBeans Enterprise dado. El servidor proporcionado puede incluir contenedores y herramientas de desarrollo para sus servidores y normalmente publicará los interfaces de bajo nivel para que otros vendedores pueden desarrollar contenedores y herramientas de desarrollo para sus servidores.

El ejemplo de casa de subastas usa el servidor JavaBeans y las herramientas de desarrollo creadas por BEA Weblogic. Visita su site para obtener una demo de 30 días.

Como todo está sujeto a las especificaciones, todos los Beans Enterprise son intercambiables con contenedores, herramientas de desarrollo, y servidores creados por otros vendedores. De hecho, podriamos escribir nuestro propio Bean Enterprise porque es posible, y algunas veces deseable, usar Beans Enterprise escritos por uno o más proveedores que ensamblaremos dentro de una aplicación de JavaBeans Enterprise.

Cómo Funcionan las Aplicaciones Multi-Fila

El objetivo de una aplicación multi-fila es que el cliente pueda trabajar con los datos de una aplicación sin conocer en el momento de la construcción dónde se encuentran los datos. Para hacer posible este nivel de transparencia, los servicios ocultos en una arquitectura multi-fila usan servicios de búsqueda para localizar los objetos del servidor remoto (el objeto interface del Bean remoto), y los servicios de comunicación de datos para mover los datos desde el cliente, a través del objeto servidor remoto, hasta su destino final en el medio de almacenaje.

Servicio de Búsqueda

Para encontrar los objetos del servidor remoto en el momento de la ejecución, el programa cliente necesita una forma de buscarlos. Una de estas formas es usar el API Java Naming y Directory Interface™ (JNDI). JNDI es un interface común para interfaces existentes de nombres y directorios. Los contenedores de los JavaBeans de Enterprise usan JNDI como interface para el servicio de nombres del Remote Method Invocation (RMI).

Durante el desarrollo, el servicio JNDI registra el interface remoto con un nombre. Siempre que el programa cliente use el mismo servicio de nombres y pregunte por el interface remoto con su nombre registrado, podrá encontrarlo. El programa cliente llama al método **lookup** sobre un objeto **javax.naming.Context** para preguntar por el interface remoto con su nombre registrado. El objeto **javax.naming.Context** es donde se almacenan las uniones y es un objeto diferente del contexto del JavaBean de Enterprise, que se cubre más adelante..

Comunicación de Datos

Una vez que el programa cliente obtiene una referencia al objeto servidor remoto, hace llamadas a los métodos de este objeto. Como el programa cliente tiene una referencia al objeto servidor remoto, se usa una técnica llamada "envolver datos" para hacer que parezca que el objeto servidor remoto es local para el programa cliente.

La "ordenación de datos" es donde las llamadas a métodos del objeto servidor remoto se empaquetan con sus datos y se envían al objeto servidor remoto. El objeto servidor remoto desempaqueta (desordena) los métodos y los datos, y llama al Bean Enterprise. El resultado de la llamda al Bean es empaquetado de nuevo y pasado de vuelta al cliente a través del objeto servidor remoto, y son desempaquetados.

Los contenedores de JavaBeans Enterprise usan servicios RMI para ordenar los datos. Cuando se compila un Bean, se crean unos ficheros **stub** (talón) y **skeleton** (esqueleto). El fichero **talón** proporciona la configuración del empaquetado y desempaquetado de datos en el cliente, y el **esqueleto** proporciona la misma información para el servidor.

Los datos se pasan entre el programa cliente y el servidor usando serialización. La serialización es una forma de representar objetos Java™ como bytes que pueden ser enviados a través de la red como un stream y pueden ser reconstuidos en el mismo estado en el que fueron enviados originalmente.

Beans de Entidad y de Sesión

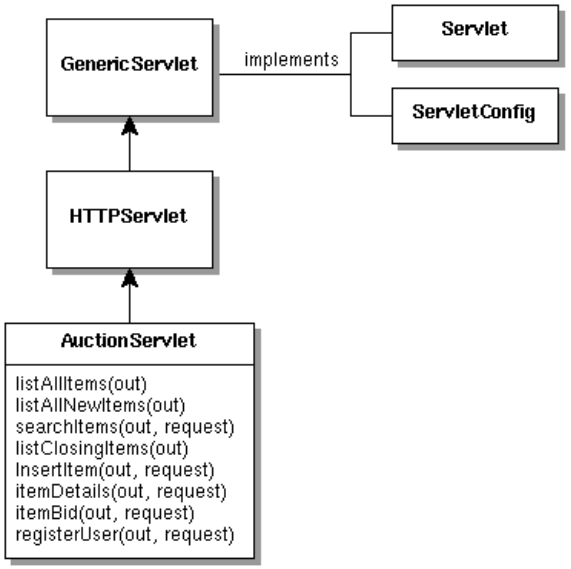
El ejemplo usa dos Beans de entidad y dos de sesión. Los Beans de entidad, **AuctionItemBean** y **RegistrationBean**, representan ítems persistentes que podrían estar almacenados en un base de datos, y los Beans de sesión, **SellerBean** y **BidderBean**, representan operaciones de vida corta con el cliente y los datos.

Los Beans de sesión son el interface del cliente hacia los beans de entidad. El SellerBean procesa peticiones para añadir nuevos ítems para la subasta. El BidderBean procesa peticiones para recuperar ítems de la subasta y situar las pujas por esos ítems. El cambio o adición de datos a la base de datos en un Bean controlado por contenedor se le deja a los Beans de entidad:

AuctionServlet

El AuctionServlet es esencialmente la segunda fila en la aplicación y el punto focal para las actividades de la subasta. Acepta entradas finales del usuario desde el navegador mediante el protocolo de transferencia de hypertexto (**HTTP**), pasa la entrada al Bean Enterprise apropiado para su proceso, y muestra el resultado del proceso al usuario final en el navegador.

Aquí hay un diagrama del tipo Unified Modeling Language (UML) para la clase **AuctionServlet**.



Los métodos de **AuctionServlet** mostrados arriba invocan a la lógica del negocio que se ejecuta en el servidor buscando un Bean Enterprise y llamando a uno o más de sus métodos. Cuando el servelt añade código **HTML** a una página para mostrarsela al usuario, la lógica se ejecuta en el cliente.

Por ejemplo, el método **listAllItems(out)** ejecuta código en el cliente para generar dinámicamente una página **HTML** para que la vea el cliente en un navegador. La página **HTML** se rellena con los resultados de una llamada a **BidderBean** que ejecuta la lógica en el servidor para generar una lista de todos los ítems de la subasta.

```
private void listAllItems(ServletOutputStream out)
                                throws IOException{

//Put text on HTML page
  setTitle(out, "Auction results");
  String text = "Click Item number for description
                and to place bid.";
  try{
     addLine("<BR>"+text, out);
//Look up Bidder bean home interface.
     BidderHome bhome=(BidderHome) ctx.lookup("bidder");
//Create Bidder bean remote interface.
     Bidder bid=bhome.create();
//Call Bidder bean method through remote interface.
     Enumeration enum=(Enumeration)bid.getItemList();

     if(enum != null) {
//Put retrieved items on servlet page.
       displayitems(enum, out);
       addLine("", out);
     }
  } catch (Exception e) {
//Pring error on servlet page.
     addLine("AuctionServlet List All Items error",out);
     System.out.println("AuctionServlet <list>:"+e);
  }
     out.flush();
}
```

Beans de Entidad

AuctionItemBean y **RegistrationBean** son Beans de entidad. **AuctionItemBean** añade nuevos ítems de subasta a la base de datos y actualiza la cantidad pujada por los usuarios cuando éstos pujan por el ítem. **RegistrationBean** añade información a la base de datos sobre usuarios registrados. Ambos Beans consisten en las clases descritas aquí.

AuctionItem Entity Bean

Aquí están las clase de **AuctionItemBean**. Recuerda que estos Beans de Enterprise son objetos distribuidos que usan el API RMI (Invocación Remota de Métodos), por eso, cuando ocurre un error se lanza una excepción RMI remota.

- AuctionItem.java
- AuctionItemHome.java
- AuctionItemBean.java
- AuctionItemPk.java

AuctionItem es un interface remoto. Describe qué hace el Bean declarando los métodos definidos por el usuario que proporcionan la lógica de negocio para este Bean. Estos métodos son usados por el cliente para interactuar con el Bean sobre la conexión remota. Su nombre se mapea a la tabla **AUCTIONITEMS** que puedes ver abajo.

AuctionItemHome es el interface home. Describe cómo se crea el Bean, como encontrarlo, y eliminarlo de su contenedor. Las herramientas de desarrollo del servidor de Beans de Enterprise proporcionarán la implementación para este interface.

AuctionItemBean es el Bean de Enterprise. Implementa **EntityBean**, proporciona la lógica de negocio para los métodos definidos por el desarrollador, e implementa los métodos de **EntityBean** para crear el Bean y seleccionar el contexto de sesión. Esta es una clase que necesita implementar el desarrollador del Bean. Sus campos variables mapean a los campos de la tabla **AUCTIONITEMS** que puedes ver abajo.

AuctionItemPK es la clase clave primaria. El servidor de Beans Enterprise requiere que un Bean de Entidad Manejado por Contenedor tenga una clase clave primaria con un campo público primario (o campos, si se usan claves primarias compuestas). El desarrollador del Bean implementa esta clase. El campo **ID** es la clave primaria en la tabla **AUCTIONITEMS** que puedes ver más abajo, por eso el campo **id** es un campo público de esta clase. Al campo **id** se le asigna un valor cuando se construye la clase de la clave primaria.

Podemos pedirle al contenedor que maneje la persistencia de la base de datos de un Bean Enterprise o escribir el código para manejar la persistencia por nosotros mismos. En este capítulo, todos los beans son manejados por el contenedor. Con esto nosotros sólo decimos qué campos son manejados por el contenedor y le dejamos al servidor de JavaBeans de Enterprise que haga el resto. Esto es fenomenal para las aplicaciones sencillas, pero si tuvieramos que codificar algo más complejo, necesitaríamos más control.

Cómo escribir los servicios ocultos de los JavaBeans Enterprise para ganar más control o proporcionar servicios similares para las aplicaciones que no usen JavaBeans de Enterprise se cubre en el capítulo 3.

Tabla Auction Items

Aquí está la tabla **AUCTIONITEMS**.

```
create table AUCTIONITEMS (SUMMARY VARCHAR(80) ,
ID INT ,
COUNTER INT ,
DESCRIPTION VARCHAR(1000) ,
STARTDATE DATE ,
ENDDATE DATE ,
STARTPRICE DOUBLE PRECISION ,
INCREMENT DOUBLE PRECISION ,
SELLER VARCHAR(30) ,
MAXBID DOUBLE PRECISION,
BIDCOUNT INT,
HIGHBIDDER VARCHAR(30) )
```

Registration Entity Bean

RegistrationBean consta de las mismas clases y tablas de base de datos que el Bean **AuctionItem**, excepto que la lógica de negocio real, los campos de la tabla de la base de datos, y la clave primaria son de alguna forma diferentes. En vez de describir las clases, podemos navegar por ellas y luego volver a la descripción de las clases de **AuctionItem** si tenemos alguna pregunta.

- Registration.java
- RegistrationHome.java
- RegistrationBean.java
- RegistrationPK.java

Tabla Registration

Aquí está la tabla **REGISTRATION**.

```
create table REGISTRATION (THEUSER VARCHAR(40) ,
PASSWORD VARCHAR(40) ,
EMAILADDRESS VARCHAR(80) ,
CREDITCARD VARCHAR(40) ,
BALANCE DOUBLE PRECISION )
```

Beans de Sesión

BidderBean y **SellerBean** son los Beans de sesión. **BidderBean** recupera una lista de los ítems de la subasta, busca ítems, chuequea el ID y la password del usuario cuando alguien hace una puja, y almacena las nuevas pujas en la base de datos. **SellerBean** chequea el ID y la password del usuario cuando alguien postea un ítem para su subasta, y añade nuevos ítems para subasta a la base de datos.

Ambos Beans de sesión están desarrollados inicialmente como Beans sin estado. Un Bean sin estado no mantiene un registro de lo que hizo el cliente en una llamada anterior; mientras que un Bean con estado completo si lo hace. Los Beans con estado completo son muy útiles si la operación es algo más que una simple búsqueda y la operación del cliente depende de algo que ha sucedido en una llamada anterior.

Bean de sesión Bidder

Aquí están las clase de **BidderBean**. Recuerda que estos Beans de Enterprise son objetos distribuidos que usan el API RMI (Invocación Remota de Métodos), por eso, cuando ocurre un error se lanza una excepción RMI remota.

No exiten claves primarias porque estos Beans son temporales y no hay accesos a la base de datos. Para recuperar ítems de la base de datos, **BidderBean** crea un ejemplar de **AuctionItemBean**, y para procesar las pujas, crea un ejemplar de **RegistrationBean**.

- Bidder.java
- BidderHome.java
- BidderBean.java

Bidder es un interface remoto. Describe lo que hace el Bean declarando los métodos definidos por el desarrollador que proporcionan la lógica de negocio para este Bean. Esto son los que que el cliente llama de forma remota.

BidderHome es el interface home. Descibe cómo se crear el Bean, como se busca y como se elimina de su contenedor.

BidderBean es el Bean de Enterprise. Implementa **SessionBean**, proporciona la lógica de negocio para los métodos definidos por el desarrollador, e implementa los métodos de **SessionBean** para crear el Bean y seleccionar el contexto de sesión.

Bean de sesion Seller

SellerBean consta de los mismos tipos de clase que un **BidderBean**, excepto que la lógica de negocio es diferente. En vez de describir las clases, puedes navegar por ellas y luego volver a la explicación de **BidderBean** si tienes alguna duda.

- Seller.java
- SellerHome.java
- SellerBean.java

Clases Contenedor

Las clases que necesita el contenedor para desarrollar un Bean Enterprise dentro de un servidor de JavaBeans Enterprise particular se generan con una herramienta de desarrollo. Las clases incluyen **_Stub.class** y **_Skel.class** que proporcionan el RMI en el cliente y el servidor respectivamente.

Estas clases se utilizan para mover datos entre el programa cliente y el servidor de JavaBeans de Enterprise. Además, la implementación de las clases se crea para los interfaces y las reglas de desarrollo definidas para nuestro Bean.

El objeto **Stub** se instala o se descarga en el sistema cliente y proporciona un objeto proxy local para el cliente. Implementa los interfaces remotos y delega de forma transparente todas las llamadas a métodos a través de la red al objeto remoto.

El objeto **Skel** se instala o se descarga en el sistema servidor y proporciona un objeto proxy local para el servidor. Despempaqueta los datos recibidos a través de la red desde el objeto **Stub** para procesarlos en el servidor.

Examinar un Bean Controlado por Contenedor

Esta sección pasea a través del código de RegistrationBean.java para ver lo fácil que es hacer que el contenedor maneje la persistencia del almacenamiento de datos en un medio oculto como una base de datos (por defecto).

Variables Miembro

Un entorno de contenedor controlador necesita saber qué variables son para almacenamiento persistente y cuales no. En el lenguaje Java™, la palabra clave **transient** indica variables que no son incluidas cuando los datos de un objeto se serializan y escriben en un almacenamiento permanente. En la clase **RegistrationBean.java**, la variable **EntityContext** está marcada como **transient** para indicar que su dato no será escrito en ningún medio de almacenamiento.

El dato de **EntityContext** no se escribe en el almacenamiento permanente porque su propósito es proporcionar información sobre el contexto en el momento de ejecución del contenedor. Por lo tanto, no contiene datos sobre el usuario registrado y no debería grabarse en un medio de almacenamiento. Las otras variables están declaradas como **public**, por lo que el contenedor de este ejemplo puede descubrirlas usando el API Reflection.

```
protected transient EntityContext ctx;
public String theuser, password, creditcard,
         emailaddress;
public double balance;
```

Método Create

El método **ejbCreate** del Bean es llamado por el contenedor después de que el programa cliente llame al método **create** sobre el interface **remoto** y pase los datos de registro. Este método asigna los valores de entrada a las variables miembro que representan los datos del usuario. El contenedor maneja el almacenamiento y carga de los datos, y crea nuevas entradas en el medio de

almacenamiento oculto.

```
public RegistrationPK ejbCreate(String theuser,
                                String password,
                                String emailaddress,
                                String creditcard)
        throws CreateException, RemoteException {

  this.theuser=theuser;
  this.password=password;
  this.emailaddress=emailaddress;
  this.creditcard=creditcard;
  this.balance=0;
```

Métodos de Contexto de Entidad

Un Bean de entidad tiene un ejemplar de **EntityContext** asociado que ofrece al Bean acceso a la información del contenedor controlador en el momento de la ejecución, como el contexto de la transación.

```
  public void setEntityContext(
               javax.ejb.EntityContext ctx)
               throws RemoteException {
    this.ctx = ctx;
  }

  public void unsetEntityContext()
               throws RemoteException{
    ctx = null;
  }
```

Método Load

El método **ejbLoad** del Bean es llamado por el contenedor para cargar los datos desde el medio de almacenamiento oculto. Esto sería necesario cuando **BidderBean** o **SellerBean** necesiten chequear la ID y password del usuario.

Nota: No todos los objetos Beans están vivos en un momento dato. El servidor de JavaBeans™ de Enterprise podría tener un número configurable de Beans que puede mantener en memoria.

Este método no está implementado porque el contenedor de los JavaBeans de Enterprise carga los datos por nosotros.

```
  public void ejbLoad() throws RemoteException {}
```

Método Store

El método **ejbStore** del Bean es llamado por el contenedor para grabar los datos del usuario. Este método no está implementado porque el contenedor de los JavaBeans de Enterprise graba los datos por nosotros.

```
public void ejbStore() throws RemoteException {}
```

Connection Pooling

La carga y almacenamiento de datos en la base de datos puede tardar mucho tiempo y reducir el rendimiento general de la aplicación. Para reducir el tiempo de conexión, el servidor de Weblogic BEA usa una cola de conexiones JDBC™ para hacer un cache con las conexiones con la base de datos, por eso las conexiones están siempre disponibles cuando la aplicación las necesita.

Sin embargo, no estamos limitados a la cola de conexiones JDBC. Podemos sobreescribir el comportamiento de la cola de conexiones del Bean y sustituirla nosotros mismos.

Descriptor de Desarrollo

La configuración restante para un Brans persistente controlado por contenedor ocurre en el momento del desarrollo. Lo que ves abajo es un Descriptor de Desarrollo basado en texto usado en un servidor de BEA Weblogic Enterprise JavaBeans.

Texto del Descriptor de Desarrollo

```
(environmentProperties

  (persistentStoreProperties
    persistentStoreType          jdbc

    (jdbc
      tableName                  registration
      dbIsShared                 false
      poolName                   ejbPool
      (attributeMap
        creditcard               creditcard
        emailaddress             emailaddress
        balance                  balance
        password                 password
        theuser                  theuser
```

```
        ); end attributeMap
      ); end jdbc
    ); end persistentStoreProperties
  ); end environmentProperties
```

El descriptor de desarrollo indica que el almacenamiento es una base de datos cuya conexión está contenida en una cola de conexiones JDBC™ llamada **ejbPool**. El **attributeMap** contiene la variable del Bean Enterprise a la izquierda y su campo asociado de la base de datos a la derecha.

Descriptor de Desarrollo XML

En Enterprise JavaBeans 1.1, el descriptor de desarrollo usa **XML**. Aquí está la configuración equivalente en **XML**:

```
<persistence-type>Container</persistence-type>
<cmp-field><field-name>creditcard
    </field-name></cmp-field>
<cmp-field><field-name>emailaddress
    </field-name></cmp-field>
<cmp-field><field-name>balance
    </field-name></cmp-field>
<cmp-field><field-name>password
    </field-name></cmp-field>
<cmp-field><field-name>theuser
    </field-name></cmp-field>
<resource-ref>
<res-ref-name>registration</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>
```

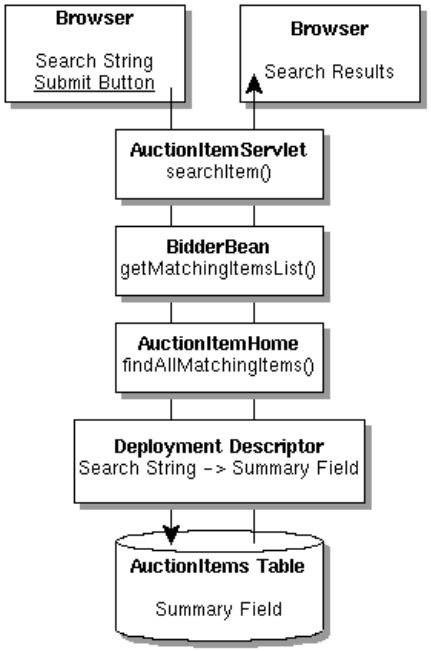
Los campos del contenedor controlador se mapean directamente a su nombre contraparte en la tabla de la base de datos. El recurso de autorización del contenedor (**res-auth**) significa que el contenedor maneja el login a la tabla **REGISTRATION**.

Métodos de Búsqueda del Contenedor Controlador

La facilidad de búsqueda de la casa de subastas está implementada como un método **finder** del contenedor. Arranca cuando el usuario escribe una cadena de búsqueda y pulsa el botón **Submit** en la página principal para localizar un ítem de la subasta. Como se muestra en el diagrama, el navegador pasa la cadena de búsqueda al método **AuctionServlet.searchItem**, que luego la pasa al método **BidderBean.getMatchingItemsList**.

En este punto, **BidderBean.getMatchingItemsList** pasa la cadena de búsqueda al método **findAllMatchingItems** declarado en el interface **AuctionItemHome**. Este método es un método **buscador**, y la implementación del contenedor varía la forma en que maneja las llamadas a los métodos **finder**. Los contenedores BEA Weblogic buscan en el descriptor de desarrollo del Bean la información sobre los métodos **finder**.

En el caso de la busqueda, el descriptor de desarrollo mapea la cadena de búsqueda pasada a **AuctionItemHome.findAllMatchingItems** al campo **summary** en la tabla **AuctionItems** de la base de datos. Este le dice al servidor Enterprise JavaBeans™ que recupere datos de todos los campos que en el campo **summary** contengan el texto de la cadena de búsqueda.



Esta sección pasea a través de las diferentes partes del código de búsqueda **finder**.

- AuctionServlet.searchItems
- BidderBean.getMatchingItemsList
- AuctionItemHome.findAllMatchingItems
- Descriptor de Desarrollo de AuctionItemBean

AuctionServlet.searchItems

El método **searchItems** recupera el texto de la cadena del navegador, crea una página HTML para mostar el resultado de la búsqueda, y le pasa la cadena de búsqueda al método **BidderBean.getMatchingItemsList**. **BidderBean** es un Bean de sesión que recupera una lista de ítems de la subasta y chequea la ID y la password del usuario para los usuarios que quieren pujar por algún articulo.

Los resultados de la búsqueda se devuelven a este método en una variable **Enumeration**.

```
    private void searchItems(ServletOutputStream out,
            HttpServletRequest request)
            throws IOException {

    //Retrieve search string
      String searchString=request.getParameter(
            "searchString");

    //Create HTML page
      String text = "Click Item number for description
            and to place bid.";
      setTitle(out, "Search Results");
      try {
            addLine("<BR>"+text, out);

    //Look up home interface for BidderBean
            BidderHome bhome=(BidderHome) ctx.lookup(
                    "bidder");

    //Create remote interface for BidderBean
            Bidder bid=bhome.create();

    //Pass search string to BidderBean method
            Enumeration enum=(Enumeration)
              bid.getMatchingItemsList(searchString);

            if(enum != null) {
              displayitems(enum, out);
              addLine("", out);
            }
      } catch (Exception e) {
        addLine("AuctionServlet Search Items error",
```

```
            out);
        System.out.println("AuctionServlet <newlist>:
            "+e);
      }
        out.flush();
    }
```

BidderBean.getMatchingItemsList

El método **BidderBean.getMatchingItemsList** llama al método **AuctionItemHome.findAllMatchingItems** y le pasa la cadena de búsqueda. **AuctionItemBean** es un bean de entidad que maneja actualizaciones y recuperaciones de ítems de la subasta.

El resultado de la búsqueda es devuelto a este método en una variable**Enumeration**.

```
public Enumeration getMatchingItemsList(
                String searchString)
        throws RemoteException {

  Enumeration enum=null;
  try{
//Create Home interface for AuctionItemBean
    AuctionItemHome home = (AuctionItemHome)
        ctx.lookup("auctionitems");

//Pass search string to Home interface method
    enum=(Enumeration)home.findAllMatchingItems(
        searchString);
  }catch (Exception e) {
    System.out.println("getMatchingItemList: "+e);
    return null;
  }
  return enum;
}
```

AuctionItemHome.findAllMatchingItems

El método **AuctionItemHome.findAllMatchingItems** no está implementado por **AuctionItemBean**. Las implementaciones del método **AuctionItemBean finder** están definidas en el descriptor de desarrollo de **AuctionItemBean** cuando se usan contenedores de BEA Weblogic.

Cuando se usan estos contenedores, incluso si el Bean tiene implementaciones del método **finder**, son ignorados y en su lugar se consultan las selecciones en el descriptor de desarrollo.

```
//Declare method in Home interface
  public Enumeration findAllMatchingItems(
                String searchString)
        throws FinderException, RemoteException;
```

Descriptor de Desarrollo de AuctionItemBean

Cuando se llama a un método **finder** de un Bean, el contenedor consulta el descriptor de desarrollo para ese Bean para encontrar qué datos necesita recuperar el método **finder** de la tabla de la base de datos. El contenedor pasa esta información al servidor Enterprise JavaBeans, que hace la recuperación real.

El descriptor de desarrollo para **AuctionItemBean** proporciona **finderDescriptors** para todos los métodos **finder** declarados en el interface **AuctionItemHome**. El **finderDescriptor** para el método **findAllMatchingItems** mapea la cadena de búsqueda al campo **summary** de la tabla **AuctionItems** de la base de datos. Esto le dice al servidor Enterprise JavaBeans que recupere los datos de todas las filas de la tabla en las que el contenido del campo summary corresponda con el texto de la cadena de búsqueda.

```
(finderDescriptors
 "findAllItems()"          "(= 1 1)"
 "findAllNewItems(java.sql.Date newtoday)"
        "(= startdate $newtoday)"
 "findAllClosedItems(java.sql.Date closedtoday)"
        "(= enddate $closedtoday)"
 "findAllMatchingItems(String searchString)"
```

```
            "(like summary $searchString)"
    ); end finderDescriptors
```

Ozito

Manejo de Datos y Transaciones

Cuando usamos la arquitectura Enterprise JavaBeans™, los datos se leen y escriben en la base de datos sin tener que escribir ningún código SQL. Pero ¿qué pasa si no quereos almacenar los datos en una base de datos, o si queremos escribir nuestros propios comandos SQL, o manejar transaciones?
Podemos sobreescribir el contenedor controlador interno de persistencia e implementar un Bean controlador de persistencia usando nuestro propio almacenamiento de datos y nuestro código de manejo de transaciones.

La persistencia del Bean controlador se convierte en útil cuando queremos más control del que proporciona el contenedor controlador. Por ejemplo podríamos sobreescribir la mayoría de los contenedores para que mapeen un Bean en una fila de la tabla, implementar nuestros propios métodos **finder**, o personalizar el caché.

Este capítulo presenta dos versiones de la clase **RegistrationBean** del capítulo anterior. Una versión lee y escribe los datos del usuario en un fichero usando streams de entrada y salida serializados. La otra versión proporciona nuestros propios comandos SQL para leer y escribir en la base de datos. También explica cómo podemos escribir nuestro propio código de manejo de transaciones.

¿Tienes Prisa?

Esta tabla te lleva directamente a los tópicos específicos

Tópico	Sección
Bean-Controlador de Persistencia y la Plataformna JDBC	• Conectar con la base de datos • Método Create • Método Load • Método Refresh • Método Store • Método Find
Manejo de Transaciones	• ¿Por qué Manejar Transaciones? • Sincronización de Sesión • Transaction Commit Mode
Métodos de Búsqueda del Bean-Controlador	• AuctionServlet.searchItems • SearchBean

Ozito

Bean-Controlador de Persistencia y la Plataforma JDBC

Puede que haya algunas veces que querramos sobreescribir la persistencia del contenedor controlador e implementar métodos de Beans de entidad o de sesión para usar nuestros propios comandos SQL. Este tipo de persistencia controlada por el Bean puede ser útil si necesitamos aumentar el redimiento o mapear datos de múltiples Beans en una sóla fila de la tabla de la base de datos.

Esta sección nos muestra cómo convertir la clase RegistrationBean.java para acceder a la base de datos con la clase **PreparedStatement** del JDBC.

Conectar con la Base de Datos

Esta versión de la clase RegistrationBean.java establece la conexión con la base de datos ejemplarizando una clase estática **Driver** y proporcionando el método **getConnection**.

El método **getConnection** necesita la clase estática **DriverManager** para un motor de la base datos registrada que corresponda con la URL. En este caso, la URL es **weblogic.jdbc.jts.Driver**.

```
//Create static instance of database driver
static {
  new weblogic.jdbc.jts.Driver();
}

//Get registered driver from static instance
public Connection getConnection() throws SQLException{
  return DriverManager.getConnection(
                        "jdbc:weblogic:jts:ejbPool");
  }
```

Método Create

El método **ejbCreate** asigna valores a las variables miembro, obtiene una conexión con la base de datos, y crea un ejemplar de la clase **java.sql.PreparedStatement** para ejecutar la sentencia SQL que escribe los datos en la tabla **registration** de la base de datos.

Un objeto **PreparedStatement** se crea desde una sentenica SQL que se envía a la base de datos y se precompila antes de enviar cualquier dato. Podemos llamar a las sentencias **setXXX** apropiadas sobre el objeto **PreparedStatement** para enviar datos. Manteniendo los objetos **PreparedStatement** y **Connection** como variables de ejemplar privadas reducimos la sobrecarga porque las sentencias SQL no tienen que compilarse cada vez que se envían.

Lo último que hace el método **ejbCreate** es crear una clase de clave primaria con el ID del usuario, y devolverlo al contenedor.

```
public RegistrationPK ejbCreate(String theuser,
                                String password,
                                String emailaddress,
                                String creditcard)
        throws CreateException, RemoteException {

  this.theuser=theuser;
  this.password=password;
  this.emailaddress=emailaddress;
  this.creditcard=creditcard;
  this.balance=0;

  try {
    con=getConnection();
    ps=con.prepareStatement("insert into registration (
                            theuser, password,
                            emailaddress, creditcard,
                            balance) values (
                            ?, ?, ?, ?, ?)");
    ps.setString(1, theuser);
    ps.setString(2, password);
    ps.setString(3, emailaddress);
    ps.setString(4, creditcard);
    ps.setDouble(5, balance);
    if (ps.executeUpdate() != 1) {
      throw new CreateException (
                  "JDBC did not create a row");
    }
    RegistrationPK primaryKey = new RegistrationPK();
```

```
      primaryKey.theuser = theuser;
      return primaryKey;
      } catch (CreateException ce) {
        throw ce;
      } catch (SQLException sqe) {
        throw new CreateException (sqe.getMessage());
      } finally {
        try {
           ps.close();
        } catch (Exception ignore) {}
        try {
           con.close();
        } catch (Exception ignore) {}
      }
  }
```

Método Load

Este método obtiene la clave primaria desde el contexto de entidad y lo pasa al método **refresh** que carga los datos.

```
    public void ejbLoad() throws RemoteException {
      try {
        refresh((RegistrationPK) ctx.getPrimaryKey());
      }
      catch (FinderException fe) {
        throw new RemoteException (fe.getMessage());
      }
    }
```

Método Refresh

El método **refresh** es el código suministrado por el programador para cargar los datos desde la base de datos. Chequea la clave primaria, obtiene la conexión con la base de datos, y crea un objeto **PreparedStatement** para consultar en la base de datos la clave primaria especificada.

Los datos se leen desde la base de datos en un **ResultSet** y se asignan a las variables miembro globales para que **RegistrationBean** tenga la información más actualizada del usuario.

```
  private void refresh(RegistrationPK pk)
               throws FinderException, RemoteException {

    if (pk == null) {
```

```
    throw new RemoteException ("primary key
                                cannot be null");
  }
  Connection con = null;
  PreparedStatement ps = null;
  try {
     con=getConnection();
     ps=con.prepareStatement("select password,
                emailaddress, creditcard,
                balance from registration
                where theuser = ?");
     ps.setString(1, pk.theuser);
     ps.executeQuery();
     ResultSet rs = ps.getResultSet();
     if (rs.next()) {
       theuser = pk.theuser;
       password = rs.getString(1);
       emailaddress = rs.getString(2);
       creditcard = rs.getString(3);
       balance = rs.getDouble(4);
     }
     else {
        throw new FinderException (
                    "Refresh: Registration ("
                    + pk.theuser + ") not found");
     }
  }
  catch (SQLException sqe) {
     throw new RemoteException (sqe.getMessage());
  }
  finally {
     try {
        ps.close();
     }
     catch (Exception ignore) {}
     try {
        con.close();
     }
     catch (Exception ignore) {}
  }
}
```

Método Store

Este método obtiene una conexión con la base de datos y crea un **PreparedStatement** para actualizarla.

```
public void ejbStore() throws RemoteException {
  Connection con = null;
  PreparedStatement ps = null;
  try {
     con = getConnection();
     ps = con.prepareStatement("update registration
                                set password = ?,
                                emailaddress = ?,
                                creditcard = ?,
                                balance = ?
                                where theuser = ?");
     ps.setString(1, password);
     ps.setString(2, emailaddress);
     ps.setString(3, creditcard);
     ps.setDouble(4, balance);
     ps.setString(5, theuser);
     int i = ps.executeUpdate();
     if (i == 0) {
       throw new RemoteException (
                   "ejbStore: Registration (
                   " + theuser + ") not updated");
     }
  } catch (RemoteException re) {
     throw re;
  } catch (SQLException sqe) {
     throw new RemoteException (sqe.getMessage());
  } finally {
    try {
       ps.close();
    } catch (Exception ignore) {}
     try {
        con.close();
     }
     catch (Exception ignore) {}
  }
}
```

Método Find

El método **ejbFindByPrimaryKey** corresponde con la firma del método **FindByPrimaryKey** del interface RegistrationHome. Este llama al método **refresh** para obtner o refrescar los datos de usuario para el usuario específicado en la clave primaria.

La versión de persistencia del contenedor controlador de **RegistrationBean** no implementa este método porque el contenedor maneja la obtención y refresco de los datos de usuario.

```
public RegistrationPK ejbFindByPrimaryKey(
                RegistrationPK pk)
                throws FinderException,
                RemoteException {

  if ((pk == null) || (pk.theuser == null)) {
     throw new FinderException ("primary key
                                  cannot be null");
  }
  refresh(pk);
  return pk;
}
```

Control de Transaciones

¿No sería maravilloso si cada operación que intentara nuestra aplicación tuviera éxito? Desafortunadamente, en el mundo multi-thread de las aplicaciones distribuidas y recursos compartidos, esto no es siempre posible.

¿Por qué? Primero de todo, los recursos compartidos deben mantener una vista consistente de los datos de todos los usuarios. Esto significa que leer y escribir tiene que ser controlado para que los usuarios no se sobreescriban los datos unos a los otros, o los errores de transación no corrompan la integridad de los datos. También, si trabajamos en una red con retardos intermitenes o caídas de conexiones, el potencial para que las operaciones fallen en una aplicación basada en web se incrementa con el número de usuarios.

Los fallos de operaciones son inevitables, lo mejor es recuperar luego la seguridad, y aquí es donde entra el control de transaciones. Las bases de datos modernas y los controladores de transaciones nos permiten deshacer y repetir el estado de una secuencia de operaciones fallidas para asegurar que los datos son consistentes para el acceso desde múltiples threads.

Esta sección añade código al **SellerBean** del ejemplo de la casa de subastas para que pueda manejar la inserción de itéms en la subasta más allá del controlador de transaciones por defecto proporcionado por su contenedor.

¿Por qué Controlar las Transaciones?

Cuando accedemos a una base de datos usando el API JDBC, todas las aplicaciones se ejecutan con una entrega automática explícita por defecto. Esto significa que cualquier aplicación que esté viendo los datos verá los datos actualizados después de cada llamada a JDBC.

Para aplicaciones sencillas, esto podría ser aceptable, pero consideremos la aplicación de la casa de subastas y las secuencias que ocurren cuando **SellerBean**

inserta un ítem de subasta. Primero se carga la cuenta del usuario para listar el ítem, y se añade el ítem a la lista de ítems de la subasta. Estas operaciones ivolucran a **RegistrationBean** para cobrar la cuenta y **AuctionItemBean** para añadir el ítem a la lista de subasta.

En el modo de entrega automático, si falla la inserción del ítem de subasta, sólo se se puede deshacer el listado, y tenemos que ajustar manualmente la cuenta del usuario para descontarle la lista de cargos. Mientras tanto, otro thread podría estar intentando deducir de la misma cuenta de usuario, sin encontrar crédito, y abortando cuando quizás unos milisegundos después se podría haber completado.

Hay dos formas para segurarnos de que el débito se ha devuelto a su valor cuando falla la inserción de un ítem en la subasta:

- Añadir código de sincronización de sesión al Bean de sesión del contenedor controlador para obtener el control sobre las entregas de la transación y volver atrás.
- Configurar JDBC para los servicios de modo de entrega transación y añadir código para arrancar, parar, entregar y deshacer la transación. Esto es una transación controlada por Bean y puede ser usada por Beans de entidad y de sesión.

Sincronización de Sesisón

Un Bean de sesión controlado por contenedor puede opcionalmente incluir sincronización de sesión para controlar la entrega automática por defecto propocionada por el contenedor. El código de sincronización permite al contenedor notificar al Bean cuando se alcanzan los puntos importantes en la transación. Además de recibir la notificación, el Bean puede tomar cualquier acción necesaria antes de que la transación proceda con el siguiente punto.

Nota: Un Bean de Sesión que usa transaciones controladas por Bean no necesita sincronización de sesión porque tiene la entrega totalmente controlada.

Ejemplo de Control por Contenedor

SellerBean es un Bean de sesión que usa **RegistrationBean** para comprobar la ID y la password del usuario cuando alguien postea un ítem para la subasta y apunta en la cuenta del vendedor un listado, y **AuctionItemBean** añade los nuevos ítems a la base de datos.

La transación empieza en el método **insertItem** con el apunte del débito y termina cuando se entrega la transación completa o se deshace. La transación completa incluye deshacer el apunte de 50 centavos si el ítem de subasta es **null** (la inserción falla), o si se captura una excepción. Si el ítem de subasta no es **null** y la inserción se realiza con éxito, se entrega la transación completa, incluyendo el

cobro de los 50 centavos.

Código

Para usar sincronización de sesión, un Bean de sesión implementa el interface **SessionSynchronzation** y sus tres métodos, **afterBegin**, **beforeCompletion**, y **afterCompletion**. Este ejemplo adapta el código de SellerBean.java para usar sincronización de sesión.

```
public class SellerBean implements SessionBean,
                        SessionSynchronization {

  private transient SessionContext ctx;
  private transient Properties p = new Properties();
  private transient boolean success = true;

  public void afterBegin() {}

  public void beforeCompletion() {
    if (!success ) {
      ctx.setRollbackOnly();
    }
  }

  public void afterCompletion(boolean state) {}
```

afterBegin: El contenedor llama a este método antes del débito para notificar al Bean de sesión de que una nueva transación va a comenzar. Podemos implementar este método que haga cualquier trabajo prévio en la base de datos que pudiera ser necesario para la transación. En este caso no son necesarios trabajos prévios, por eso este método no está implementado.

beforeCompletion: El contenedor llama a este método cuando está listo para escribir el ítem de subasta y el débito en la base de datos, pero antes de hacerlo realmente (entregarlo). Podemos implementar este método para escribir cualquier actualización caché de la base de datos o deshacer la transación. En este ejemplo, el método llama al método **setRollbackOnly** sobre el contexto de la sesión en el caso de que la variable **success** sea **false** durante la transación.

afterCompletion: El contenedor llama a este método cuando la transación se entrega. Un valor booleano de **true** significa que el dato ha sido enviado y **false** significa que la transación se ha deshecho. El método usa el valor **boolean** para determinar si necesita resetear el estado del Bean en el caso de que se haya deshecho. En este ejemplo, no es necesario resetear el estado en el caso de un fallo.

Aquí está el método **insertItem** con comentarios que muestran dónde están los

puntos donde se llama a los métodos de **SessionSynchronization**.

```
    public int insertItem(String seller,
                          String password,
                          String description,
                          int auctiondays,
                          double startprice,
                          String summary)
                    throws RemoteException {

      try{
        Context jndiCtx = new InitialContext(p);

        RegistrationHome rhome =
            (RegistrationHome) sCtx.lookup("registration");
        RegistrationPK rpk=new RegistrationPK();
        rpk.theuser=seller;
        Registration newseller=rhome.findByPrimaryKey(rpk);

        if((newseller == null) ||
            (!newseller.verifyPassword(password))) {
          return(Auction.INVALID_USER);
        }

    //Call to afterBegin
        newseller.adjustAccount(-0.50);

        AuctionItemHome home = (AuctionItemHome)
            jndiCtx.lookup("auctionitems");
        AuctionItem ai= home.create(seller,
                                    description,
                                    auctiondays,
                                    startprice,
                                    summary);
        if(ai == null) {
          success=false;
          return Auction.INVALID_ITEM;
        }
        else {
          return(ai.getId());
        }

        }catch(Exception e){
          System.out.println("insert problem="+e);
          success=false;
```

```
        return Auction.INVALID_ITEM;
      }
//Call to beforeCompletion
//Call to afterCompletion

 }
```

Modo de Entrega de la Transación

Si configuramos los servicos JDBC para modo de entrega de transación, podemos hacer que el Bean controle la transación. Para configurar los servicios de JDBC para la entrega, llamamos a **con.setAutoCommit(false)** sobre nuestra conexión JDBC. No todos los drivers JDBC soportan el modo de entrega, pero para hacer que el Bean controle y maneje las transaciones, necesitamos un driver que lo haga.

El modo de entrega de la transación nos permite añadir código que crea una red de seguridad alrededor de una secuencia de operaciones dependientes. El API de Transaction de Java, proporciona las ayudas que necesitamos para crear esa red de seguridad. Pero si estamos usando la arquitectura JavaBeans de Enterprise, podemos hacerlo con mucho menos código. Sólo tenemos que configurar el servidor de JavaBeans de Entrprise, y especificar en nuestro código donde empieza la transación, donde para, donde se deshace y se entrega.

Configuración del servidor

Configurar el servidor de JavaBeans Enterprise implica especificar las siguientes selecciones en un fichero de configuración para cada Bean:

- Un nivel de aislamiento para especificar cómo de exclusivo es el acceso de una transasción a los datos compartidos.
- Un atributo de transación para especificar cómo controlar las transaciones mediante el Bean o el contenedor que continúa en otro Bean.
- Un tipo de transación para especificar si la transación es manejada por el contenedor o por el Bean.

Por ejemplo, podríamos especificar estas selecciones para el servidor BEA Weblogic en un fichero **DeploymentDescriptor.txt** para cada Bean.

Aquí está la parte de **DeploymentDescriptor.txt** para **SellerBean** que especifica el nivel de aislamiento y el atributo de transación.

```
(controlDescriptors
  (DEFAULT
     isolationLevel              TRANSACTION_SERIALIZABLE
     transactionAttribute        REQUIRED
     runAsMode                   CLIENT_IDENTITY
     runAsIdentity               guest
```

```
  ); end DEFAULT
); end controlDescriptors
```

Aquí está el equivalente en lenguaje XML.

```
<container-transaction>
    <method>
        <ejb-name>SellerBean<ejb-name>
        <method-name>*<method-name>
    <method>
    <transaction-type>Container<transaction-type>
    <trans-attribute>Required<trans-attribute>
<container-transaction>
```

En este ejemplo, **SellerBean** está controlado por el Bean.

```
<container-transaction>
    <method>
        <ejb-name>SellerBean<ejb-name>
        <method-name>*<method-name>
    <method>
    <transaction-type>Bean<transaction-type>
    <trans-attribute>Required<trans-attribute>
<container-transaction>
```

Descripción de Atributo de Transación: Un Bean Enterprise usa un *transaction attribute* para especificar si una transación de Bean es manejada por el propio Bean o por el contenedor, y cómo manejar las transaciones que empezaron en otro Bean.

El servidor de JavaBeans de Enterprise sólo puede controlar una transación a la vez. Este modelo sigue el ejemplo configurado por el Object Transaction Service (OTS) de la OMG, y significa que la especificación actual de los JavaBeans Enterpirse no proporcionan una forma para transaciones anidadas. Una transación anidada es un nueva transación que arranca dentro de otra transación existente. Mientras que las transaciones anidadas no están permitidas, continuar una transación existente en otro Bean es correcto.

Cuando se entra en un Bean, el servidor crea un contexto de transación para controlar la transación. Cuando la transación es manejada por le Bean, accedemos para comenzar, entregar o deshacer la transación cuando sea necesario.

Aquí están los atributos de transación con una breve descripción de cada uno de ellos. Los nombres de los atributos cambiaron entre las especificaciones 1.0 y 1.1 de los JavaBeans Enterprise.

Especificación 1.	**Especificación 1.0**
REQUIRED	**TX_REQUIRED**

Transación controlada por el contenedor. El servidor arranca y maneja una nueva transación a petición del usuario o continúa usando la transación que se arrancó en el código que llamó a este Bean.

REQUIRESNEW **TX_REQUIRED_NEW**

Transación controlada por contenedor. El servidor arranca y maneja una nueva transación. Si una transación existente arranca esta transación, la suspende hasta que la transación se complete.

Especificado como tipo de transación de Bean en el Descriptor de desarrollo **TX_BEAN_MANAGED**

<Transación controlada por el Bean. Tenemos acceso al contexto de la transación para empezar, entregar o deshacer la transación cuando sea necesario.

SUPPORTS **TX_SUPPORTS**

Si el código que llama a este Bean tiene un transación en ejecución, incluye este Bean en esa transación.

NEVER **TX_NOT_SUPPORTED**

Si el código que llama a un método en este Bean tiene una transación ejecuntándose, suspende esa transación hasta que la llamada al método de este Bean se complete. No se crea contexto de transación para este Bean.

MANDATORY **TX_MANDATORY**

El atributo de transación para este Bean se configura cuando otro bean llama a uno de sus métodos. En este caso, este bean obtiene el atributo de transación del Bean llamante. Si el Bean llamante no tiene atributo de transación, el método llamado en este Bean lanza una excepcioón **TransactionRequired**.

Descripción del Nivel de Aislamiento: Un Bean de Enterprise usa un **nivel de aislamiento** para negociar su propia interacción con los datos compartidos y la interacción de otros threads con los mismos datos compartidos. Como el nombre indica, existen varios niveles de aislamiento con **TRANSACTION_SERIALIZABLE** siendo el nivel más alto de integridad de los datos.

Nota: Debemos asegurarnos de que nuestra base de datos puede soportar el nivel elegido. En la especificación 1.1 de los JavaBeans de Enterprise, sólo los Beans de sesión con persistencia controlada por el Bean pueden seleccionar el nivel de aislamiento.

Si la base de datos no puede controlar el nivel de aislamiento, el servidor de JavaBeans Enterprise dará un fallo cuando intente acceder al método **setTransactionIsolation** de JDBC.

TRANSACTION_SERIALIZABLE: Este nivel proporciona la máxima integridad de los datos. El Bean decide la cantidad de accesos exclusivos. Ninguna otra transación puede leer o escribir estos datos hasta que la transación serializable se complete.

En este contexto, serializable significa *proceso como una operación serial*, y no debería confundirse con la serialización de objetos para preservar y restaurar sus estados. Ejecutar transaciones como una sóla operación serial es la selección más lenta. Si el rendimiento es un problema, debemos usar otro nivel de aislamiento que cumpla con los requerimientos de nuestra aplicación, pero mejore el rendimiento.

TRANSACTION_REPEATABLE_READ: En este nivel, los datos leidos por una transación pueden ser leidos, pero no modificados, por otra transación. Se garantiza que el dato tenga el mismo valor que cuando fue leído por primera vez, a menos que la primera transación lo cambie y escriba de nuevo el valor cambiado.

TRANSACTION_READ_COMMITTED: En este nivel, los datos leídos por una transación no pueden ser leídos por otra transación hasta que la primera transación los haya entregado o deshecho

TRANSACTION_READ_UNCOMMITTED:En este nivel, los datos involucrados en una transación pueden ser leídos por otros threads antes de que la primera transación se haya completado o se haya deshecho. Las otras transaciones no pueden saber si los datos fueron finalmente entregados o deshechos.

Ejemplo de Bean Controlador

SellerBean es un Bean de sesión que usa **RegistrationBean** para chequear la ID y la password del usuario cuando alguien postea un ítem para la subasta, apunta el débito en la cuenta del usuario, y **AuctionItemBean** añade un nuevo ítem a la base de datos de la subasta.

La transación empieza en el método **insertItem** con el débito de la cuenta y termina cuando la transación completa se entrega o se deshace. La transación completa incluye deshacer el apunte de 50 centavos si el ítem de subasta es **null** (la inserción falla), o si se captura una excepción. Si el ítem de subasta no es **null** y la inserción se realiza con éxito, se entrega la transación completa, incluyendo el cobro de los 50 centavos.

Para este ejemplo, el nivel de aislamiento es **TRANSACTION_SERIALIZABLE**, y el atributo de transación es **TX_BEAN_MANAGED**. Los otros Beans en la transación, **RegistrationBean** y **AuctionItemBean**, tienen un nivel de aislamiento de **TRANSACTION_SERIALIZABLE** y un atributo de transación de **REQUIRED**.

Los cambios en esta versión de **SellerBean** sobre la versión del contenedor controlador se marcan con comentarios:

```
    public int insertItem(String seller,
            String password,
            String description,
            int auctiondays,
```

```
        double startprice,
        String summary)
        throws RemoteException {

//Declare transaction context variable using the
//javax.transaction.UserTransaction class
  UserTransaction uts= null;

  try{
    Context ectx = new InitialContext(p);

//Get the transaction context
    uts=(UserTransaction)ctx.getUserTransaction();

    RegistrationHome rhome = (
        RegistrationHome)ectx.lookup("registration");
    RegistrationPK rpk=new RegistrationPK();
    rpk.theuser=seller;
    Registration newseller=
        rhome.findByPrimaryKey(rpk);

    if((newseller == null)||
        (!newseller.verifyPassword(password))) {
        return(Auction.INVALID_USER);
    }

//Start the transaction
    uts.begin();

//Deduct 50 cents from seller's account
    newseller.adjustAccount(-0.50);

    AuctionItemHome home = (
        AuctionItemHome) ectx.lookup("auctionitems");
    AuctionItem ai= home.create(seller,
        description,
        auctiondays,
        startprice,
        summary);

    if(ai == null) {
//Roll transaction back
        uts.rollback();
        return Auction.INVALID_ITEM;
    }
```

```
    else {
//Commit transaction
        uts.commit();
        return(ai.getId());
    }

    }catch(Exception e){
        System.out.println("insert problem="+e);
//Roll transaction back if insert fails
        uts.rollback();
        return Auction.INVALID_ITEM;
    }
}
```

Ozito

Métodos de Búsqueda de Bean Controlador

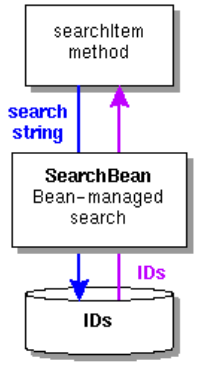
La búsqueda en el contenedor controlador descrita en el capítulo 2 está basada en el mécanismo del método **finder** donde el descriptor de desarrollo, en lugar del Bean, especifica el comportamiento del método **finder**. Mientras el mecanismo del método **finder** funciona bien para consultas sencillas, no puede manejar operaciones complejas que impliquen más de un tipo de Bean o tablas de bases de datos. También, la especificación 1.1 de los JavaBeans de Enterprise actualmente no proporciona para poner las reglas del método **finder** en el descriptor de desarrollo.

Por eso, para consultas y búsquedas más complejas, tenemos que escribir búsquedas y consultas controladas por el Bean. Esta sección explica cómo escribir una versión de la facilidad de búsqueda de la casa de subastas controlada por el Bean. La búsqueda controlada por el Bean inplica cambios en el método **AuctionServlet.searchItems** y un nuevo Bean de sesión, **SearchBean**.

AuctionServlet.searchItems

La búsqueda empieza cuando el usuario final envía una cadena de búsqueda a la facilidad de búsqueda de la página principal de la casa de subastas, y pulsa el boton **Submit**. Esto llama a **AuctionServlet**, que recupera la cadena de búsqueda desde la cabecera **HTTP** y la pasa al método **searchItem**.

> **Nota:** La lógica de búsqueda para este ejemplo es bastante simple. El propósito es mostrar cómo mover la lógica de búsqueda a otro Bean Enterprise separado para que podamos mover búsquedas más complejas nosotros solos.

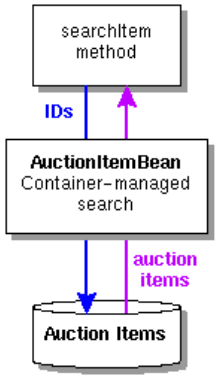


operación **searchItem** se hace en dos partes: 1) usar la cadena de búsqueda para recuperar las claves primarias, y 2) usar las claves primarias para recuperar los ítems de la subasta

Parte 1: Lo primero que hace el método searchItems es pasar la cadena de búsqueda enviada por el usuario final al Bean de sesión **SearchBean**.

SearchBean (descrito en la siguiente sección) implementa una búsqueda controlada por Bean que recupera una lista de claves primarias para todos los ítems de la subasta cuyo campo **Summary** contenga caracteres que correspondan con los de la cadena de búsqueda. Esta lista es devuelta al método **searchItems** en una variable **Enumeration**.

```
Enumeration enum=(Enumeration)
  search.getMatchingItemsList(searchString);
```



2: El método **searchItems** usa la lista **Enumeration** devuelta en la parte 1 y usa AuctionItemBean para recuperar cada Bean por turno llamando a **findByPrimaryKey** sobre cada clave primaria de la lista. Esta es una búsqueda controlada por contenedor basada en el mecanismo del método **finder** descrito en el capítulo 2.

```
//Iterate through search results
while ((enum != null) &&
    enum.hasMoreElements())) {
 while(enum.hasMoreElements(in)) {

//Locate auction items
 AuctionItem ai=ahome.findByPrimaryKey((
    AuctionItemPK)enum.nextElement());
 displayLineItem(ai, out);
 }
}
```

SearchBean

La clase SearchBean.java define una búsqueda controlada por el Bean para claves primarias de ítems de subasta con los campos **summary** que contienen caracteres que corresponden con la cadena de búsqueda. Este Bean establece una conexión con la base de datos, y proporciona los métodos **getMatchingItemsList** y **EJBCreate**.

Conexión con la Base de Datos

Como este Bean controla su propio acceso a la base de datos, tiene que establecer su propia conexión con ella. No puede delegar esto al contenedor..

La conexión con la base de datos se establece ejemplarizando una clase **Driver** estática y proporcionando el método **getConnection**. Este método requiere una clase estática **DriverManager** para registrar un driver con la base de datos que corresponda con la URL. En este caso la URL es **weblogic.jdbc.jts.Driver**.

```
//Establish database connection
  static {
    new weblogic.jdbc.jts.Driver();
  }

  public Connection getConnection()
        throws SQLException {
    return DriverManager.getConnection(
```

```
            "jdbc:weblogic:jts:ejbPool");
    }
```

Obtener la Lista de Ítems Encontrados

El método **getMatchingItemsList** busca **AuctionItemsBean** y crea un objeto **PreparedStatement** para hacer una consulta a la base de datos por los campos **summary** que contengan la cadena de búsqueda. Los datos se leen desde la base de datos dentro de un **ResultSet**, almacenado en un **Vector**, y devuelto a **AuctionServlet**.

```
  public Enumeration getMatchingItemsList(
        String searchString)
        throws RemoteException {

    ResultSet rs = null;
    PreparedStatement ps = null;
    Vector v = new Vector();
    Connection con = null;

    try{
//Get database connection
      con=getConnection();
//Create a prepared statement for database query
      ps=con.prepareStatement("select id from
        auctionitems where summary like ?");
      ps.setString(1, "%"+searchString+"%");
//Execute database query
      ps.executeQuery();
//Get results set
      rs = ps.getResultSet();
//Get information from results set
      AuctionItemPK pk;
      while (rs.next()) {
        pk = new AuctionItemPK();
        pk.id = (int)rs.getInt(1);
//Store retrieved data in vector
        v.addElement(pk);
      }
      rs.close();
      return v.elements();

    }catch (Exception e) {
        System.out.println("getMatchingItemsList:
                           "+e);
        return null;
    }finally {
      try {
         if(rs != null) {
           rs.close();
         }
         if(ps != null) {
           ps.close();
         }
         if(con != null) {
           con.close();
         }
      } catch (Exception ignore) {}
    }
  }
```

Método Create

El método **ejbCreate** crea un objeto **javax.naming.InitialContext**. Esta es una clase JNDI (Java Nanimg and Directory) que permite a **SearchBean** acceder a la base de datos sin relacionarse con el contenedor:

```
  public void ejbCreate() throws CreateException,
    RemoteException {
```

```
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.TengahInitialContextFactory");
        try{
            ctx = new InitialContext(p);
        }catch(Exception e) {
            System.out.println("create exception: "+e);
        }
    }
```

Ozito

Cálculo Distribuido

Tan recientemente como hace diez años, el cálculo distribuido generalmente significaba que teniamos clientes PCs en una habitación con un servidor en otra. El problema con esta arquitectura es que si se pierde la conexión con el servidor, los clientes no pueden actualizar las bases de datos de la compañía.

Para evitar esta pérdida de tiempo, se crearon los diferentes modelos de red. Un ejemplo es el modelo de servidor maestro y esclavo donde si el maestro falla, el esclavo toma el relevo. El problema con los distintos modelos de red es que todos requieren alguna forma de intervención manual y se unieron con un sistema operativo o un lenguaje. Y aunque estas aproximaciones consiguieron reducir el tiempo de parada, no cumplen con los sistemas de distribución heterogénea que consiste en una mezcla de protocolos de red y máquinas.

La plataforma Java™ combinada con otros avances como Common Object Request Broker Architecture (CORBA), servidores multi-fila, y redes sin cables han llevado un paso mas allá la realización de la computación totalmentedistribuida, de la tradicional aproximación cliente y servidor.

Ahora podemos construir aplicaciones que incluyan servicios de redundancia por defecto. Si una conexión de servidor falla, podemos usar un servicio de otro servidor. CORBA y los puentes "Distributed Component Object Model" (DCOM) significan que los objetos pueden ser transferidos entre casi todas las máquinas y lenguajes. Y con el nuevo sistema de software Jini™, el entorno de cálculo distribuido pude estar pronto en todos los hogares, oficinas o escuelas.

¿Tienes Prisa?

Esta tabla te llevará directamente a los tópicos especificados.

Tópico	Sección

Servicios de Búsqueda	• Java Naming and Directory Interface (JNDI) • Common Object Request Broker Architecture (CORBA) Naming Service • Interoperable Object References (IOR) • Remote Method Invocation (RMI) • RMI Over Internet Inter-ORB Protocol (IIOP) • JINI Lookup Services • Aumentar el Rendimiento de la Búsqueda

Remote Method Invocation (RMI)	• Sobre RMI
	• RMI en la aplicación Subasta
	  ○ Introducción a la Clase
	  ○ Sumario de Ficheros
	  ○ Compilar el Ejemplo
	  ○ Arrancar el Registro RMI
	  ○ Arrancar el Servidor Remoto
	• Establecer Comunicaciones Remotas
	• La clase RegistrationServer
	  ○ Exportar un Objeto Remoto
	  ○ Pasar por Valor y por Referencia
	  ○ Recolección de Basura Distribuida
	• Interface Registration
	• Interface ReturnResults
	• La Clase SellerBean

Common Object Request Broker Architecture (CORBA)	• Esquema de Mapeo IDL
	  ○ Referencia Rápida
	  ○ Configurar el Mapeo IDL
	  ○ Otros Tipos IDL
	• CORBA en la Aplicación de Subasta
	  ○ CORBA RegistrationServer
	  ○ Fichero de Mapeos IDL
	  ○ Compilar el Fichero de Mapeos IDL
	  ○ Ficheros Stub y Skeleton
	• Object Request Broker (ORB)

<table>
<tr><td></td><td>

Hacer Accesible el Servidor CORBA
Añadir un nuevo ORB
Accesos al Servicio de Nombres por clientes CORBA

Clases Helper y Holder
Recolección de Basura
Retrollamadas CORBA
Uso de Cualquier Tipo
Conclusión

</td></tr>
<tr><td>Tecnología JDBC</td><td>

Drivers JDBC
Conexiones a Bases de Datos
Sentencias

Sentencias Callable
Sentencias
Sentencias Preparadas

Guardar los Resultados de la Base de Datos
Hojas de Resultados
Hojas de Resultados Scrollables
Controlar Transaciones
Caracteres de Escape
Mapeo de Tipos de Bases de Datos
Mapeo de Tipos de Datos

</td></tr>
</table>

Servicios de Búsqueda

Los servicios de búsqueda permiten las comunicaciones a través de la red. Un programa cliente puede usar un protocolo de búsqueda para obtener información sobre programas remotos o máquinas que usen esa información para establecer una comunicación.

- Un servicio de búsqueda común con el que podríamos estar familiarizados es el Directory Name Service (DNS). Mapea direcciones de Internet Protocol (IP) a nombres de máquinas. Los programas usan el mapeo DNS para buscar direcciones IP asociadas con un nombre de máquina y usar la dirección IP para establecer una comunicación.
- De la misma forma, el AuctionServlet presentado en Chapter 2 usa el servicio de nombres interno de la arquitectura de JavaBeans Enterprise para buscar unas referencias a Beans Enterprise registrados con el servidor de JavaBeans Enterprise.

Además de los servicios de nombres, algunos protocolos de búsqueda proporcionan servicios de directorio. Este servicios como el Lightweight Directory Access Protocol (LDAP) y el NIS+ de Sun proporcionan otra información y servicios más allá de los disponibles con el servicio de nombres. Por ejemplo, NIS+ asocia un atributo **workgroup** con una cuenta de usuario. Este atributo puede usarse para restringir el acceso a una máquína, por lo que sólo los usuarios especificados en el **workgroup** tienen acceso.

Este capítulo describe como se usa el "Naming and Directory Interface (JNDI)" de Java en la aplicación de subastas para buscar los Beans de Enterprise. También explica como usar algunos de los otros muchos servicios de búsqueda que tenemos disponibles. El código para usar estos servicios no es tan sencillo como el código de la búsqueda en la aplicación de la subasta del capítulo 2, pero las ventajas que ofrecen estos otros servicios hacen que algunas veces merezca la pena ese código más complejo.

Java Naming and Directory Interface (JNDI)

El API de JNDI hace sencillo conectar servicios de búsqueda de varios proveedores en un programa escrito en lenguaje Java. Siempre que el cliente y el servidor usen el mismo servicio de búsqueda, el cliente puede fácilmente buscar información registrada en el servidor y establecer una comunicación.

Los Beans de sesión de la aplicación de subasta usan JNDI y una fábrica de nombres JNDI especial de BEA Weblogic para buscar Beans de entidad. Los servicios JNDI normalmente inicializan la fábrica de nombres como una propiedad de la línea de comandos o como un valor de inicialización.

Primero, la fábrica de nombres **weblogic.jndi.TengahInitialContextFactory** se pone dentro de un objeto **java.util.Property**, luego este objeto se pasa como parámetro al constructor de **InitialContextT**. Aquí tenemos un ejemplo del método **ejbCreate**:.

```
Context ctx; //JNDI context

public void ejbCreate()
      throws CreateException, RemoteException {
  Hashtable env = new Hashtable();
  env.put(Context.INITIAL_CONTEXT_FACTORY,
   "weblogic.jndi.TengahInitialContextFactory");
  try{
   ctx = new InitialContext(env);
  }catch(Exception e) {
   System.out.println("create exception: "+e);
  }
}
```

Una vez creado, el contexto JNDI se usa para buscar los interfaces principales de los Beans Enterprise. En este ejemplo, se recupera una referencia a un Bean Enterprise uinda al nombre **registration** y se usa para operaciones posteriores:

```
RegistrationHome rhome =
      (RegistrationHome) ctx.lookup("registration");
RegistrationPK rpk=new RegistrationPK();
rpk.theuser=buyer;
Registration newbidder =
      rhome.findByPrimaryKey(rpk);
```

En el lado del servidor, el descriptor de desarrollo para el **RegistrationBean** tiene su valor **beanhomename** como **registration**. Las herramientas de JavaBeans de Enterprise generan el resto del código de nombres para el servidor.

El servidor llama a **ctx.bind** para unir el nombre **registration** al contexto JNDI. El parámetro **this** referencia a la clase **_stub** que representa el **RegistrationBean**.

```
ctx.bind("registration", this);
```

JNDI no es la única forma de localizar objetos remotos. Los servicios de búsqueda también están disponibles en las plataformas RMI, JNI y CORBA. Podemos usar directamente los servicios de búsqueda de estas plataformas directamente desde el API del JNDI. JNDI permite a las aplicaciones cambiar el servicio de nombres con poco esfuerzo. Por ejemplo, aquí está el código que hace que el método **BidderBean.ejbCreate** use el servicio de búsqueda de **org.omb.CORBA** en vez del servicio de búsqueda por defecto de BEA Weblogic.

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
      "com.sun.jndi.cosnaming.CNCtxFactory");
Context ic = new InitialContext(env);
```

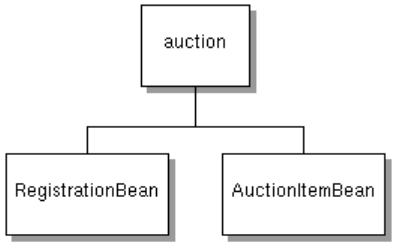
Servico de Nombres CORBA

El "Common Object Request Broker Architecture" (CORBA) define una especificación para que los objetos de un sistema distribuido se comuniquen unos con otros. Los objetos que usan la especificación CORBA para comunicarse se llaman objetos CORBA, y consisten en objetos cliente y servidor.

Los objetos CORBA puede estar escritos en cualquier lenguaje con el mapeo "Interface Definition Language" (IDL). Estos lenguajes incluyen lenguajes de programación como Java, C++, y muchos otros lenguajes tradicionales no orientados a objetos.

El servicio de búsqueda de nombres, al igual que otras especificaciones CORBA, está definido en términos de IDL. El módulo IDL para el servicio de búsqueda CORBA se llama **CosNaming**. Cualquier plataforma con un mapeo IDL, como la herramienta **idltojava**, puede usar este servicio para descubrir objetos CORBA. El módulo IDL para este servicio de búsqueda CORBA está disponible en la plataforma Java 2 en el paquete **org.omg.CosNaming**.

El interface clave en el módulo **CosNaming** es **NamingContext**. Este interface define métodos para unir objetos a un nombre, listar estas uniones, y recuperar referencias a dichos objetos.



Además de estos interfaces públicos hay clases de ayuda. La clase **NameComponent** se usa en programas cliente y servidor CORBA para construir el nombre completo para el nombre del objeto referencia. El nombre completo es un array de uno o más **NameComponents** que indica donde encontrar los objetos. El esquema de nombrado puede ser específico de la aplicación.

Por ejmplo en la aplicación de subastas, el nombre completo puede ser definido para usar **auction** como la raíz del contexto de nombres y **RegistrationBean** y **AuctionItemBean** como hijos del contexto raíz. Esto en efecto utiliza un esquema de nombres similar al usado a los paquetes de clases.

En este ejemplo, la aplicación de subastas a adaptado **SellerBean** a un servicio de nombres CORBA para buscar el **RegistrationBean** CORBA. El siguiente código se ha extraído de **SellerBean**, y actúa como un cliente CORBA, y el servidor CORBA **RegistrationServer**.

CORBA RegistrationServer

Este código del programa RegistrationServer crea un objeto **NameComponent** que indica dónde está localizado el **RegistrationBean** usando **auction** y **RegistrationBean** como el nombre completo:

```
NameComponent[] fullname = new NameComponent[2];
fullname[0] = new NameComponent("auction", "");
fullname[1] = new NameComponent(
                    "RegistrationBean", "");
```

El siguiente códido une el **fullname** como un nuevo contexto. Los primeros elementos en el nombre completo (**auction** en este ejemplo) son huecos para construir el árbol del contexto de nombrado. El último elemento del nombre completo (**RegistrationBean** en este ejemplo) es el nombre enviado para unirlo al objeto:

```
String[] orbargs = { "-ORBInitialPort 1050"};
ORB orb = ORB.init(orbargs, null) ;

RegistrationServer rs= new RegistrationServer();
orb.connect(rs);

try{
  org.omg.CORBA.Object nameServiceObj =
      orb.resolve_initial_references("NameService");
  NamingContext nctx =
      NamingContextHelper.narrow(nameServiceObj);
  NameComponent[] fullname = new NameComponent[2];
  fullname[0] = new NameComponent("auction", "");
  fullname[1] = new NameComponent(
                      "RegistrationBean", "");

  NameComponent[] tempComponent =
                    new NameComponent[1];
```

```
for(int i=0; i < fullname.length-1; i++ ) {
   tempComponent[0]= fullname[i];
   try{
      nctx=nctx.bind_new_context(tempComponent);
   }catch (Exception e){}
}
tempComponent[0]=fullname[fullname.length-1];

// finally bind the object to the full context path
nctx.bind(tempComponent, rs);
```

Una vez que se ha unido el objeto **RegistrationServer**, puede ser localizado con una búsqueda JNDI usando el proveedor de servicio **CosNaming** como se describe al final de la sección JNDI, o usando el servicio de búsquedas de nombres CORBA. De cualquier forma, el servidor de nombres CORBA debe arrancarse antes de que pueda suceder cualquier búsqueda. En la plataforma Java 2, el **nameserver** CORBA se arranca con este comando:

```
tnameserv
```

Esto arranca el **RegistrationServer** CORBA en el puerto TCP por defecto 900. Si necesitamos usar otro puerto diferente, podemos arrancar el servidor de esta forma. En sistemas Unix, sólo el administrador puede acceder a los números de puerto inferiores a 1025,

```
tnameserv -ORBInitialPort 1091
```

CORBA SellerBean

En el lado del cliente, la búsqueda CORBA usa el objeto **NameComponent** para construir el nombre. Arrancamos el servidor de objetos de esta forma:

```
java registration.RegistrationServer
```

La diferencia en el cliente es que este nombre se pasa al método **resolve** que devuelve el objeto CORBA. El siguiente código del objeto SellerBean ilustra este punto:

```
String[] args = { "-ORBInitialPort 1050"};
orb = ORB.init(args, null) ;
org.omg.CORBA.Object nameServiceObj =
      orb.resolve_initial_references("NameService") ;
nctx= NamingContextHelper.narrow(nameServiceObj);

NameComponent[] fullname = new NameComponent[2];
fullname[0] = new NameComponent("auction", "");
fullname[1] = new NameComponent(
                    "RegistrationBean", "");

org.omg.CORBA.Object cobject= nctx.resolve(fullname);
```

El método **narrow**, desde el método **Helper**, es generado por el compilador IDL, que proporciona una mapeo detallado para traducir cada campo CORBA en su respectivo campo del lenguaje Java. Por ejemplo, el método **SellerBean.insertItem** busca un objeto CORBA registrado usando el nombre **RegistrationBean**, y devuelve un objeto **RegistrationHome**. Con el objeto **RegistrationHome** podemos devolver un registro Registration llamando a su método **findByPrimaryKey**.

```
org.omg.CORBA.Object cobject= nctx.resolve(fullname);
RegistrationHome regHome=
RegistrationHomeHelper.narrow(cobject);
RegistrationHome regRef =
      RegistrationHomeHelper.narrow(
        nctx.resolve(fullname));
RegistrationPKImpl rpk= new RegistrationPKImpl();
rpk.theuser(seller);
Registration newseller =
      RegistrationHelper.narrow(
        regRef.findByPrimaryKey(rpk));
if((newseller == null)||
      (!newseller.verifyPassword(password))) {
  return(Auction.INVALID_USER);
}
```

Interoperable Object References (IOR)

Usar un servicio de nombres CORBA funciona para la mayoría de las aplicaciones CORBA, especialmente cuando el (ORB) está suministrado por un vendedor. Sin embargo, podríamos encontrar que el servicio de nombres no es totalmente compatible con todos los ORBs, y podríamos obtener el frustante mensaje **COMM_FAILURE** cuando el cliente CORBA intente conectarse con el servidor CORBA.

La solución es usar un "Interoperable Object Reference" (IOR) en su lugar. Este está disponible en los ORBs que soportan el protocolo "Internet Inter-ORB Protocol" (IIOP). Conteine la información que el servicio de nombres podría mantener para cada objeto como el host y el puerto donde reside el objeto, una única clave de búsqueda para el objeto en ese host, y qué version de IIOP soporta.

Servidor IOR

Para crear un IOR todo lo que tenemos que hacer es llamar al método **object_to_string** desde la clase **ORB** y pasarle un ejemplar del objeto. Por ejemplo, para convertir el objeto **RegistrationServer** en un IOR, necesitamos añadir la línea **String ref = orb.object_to_string(rs);** del siguiente código en el programa principal:

```
  String[] orbargs=  {"-ORBInitialPort 1050"};
  ORB orb = ORB.init(orbargs, null);
  RegistrationServer rs = new RegistrationServer();
//Add this line
  String ref = orb.object_to_string(rs);
```

Por eso, en lugar de recuperar la información de este objeto desde un servicio de nombres, hay otra forma para que el servidor envíe esta información a un cliente. Podemos registrar el **string** devuelto con un nombre sustitutivo del servidor, que puede ser un sencillo servidor web HTTP porque el objeto ya está en un formato transmitible.

Cliente IOR

Este ejemplo usa una conexión HTTP para convertir el string IOR de nuevo en un objeto. Podemos llamar al método **string_to_object** desde la clase **ORB**. Este método llama al IOR desde el **RegistrationServer** y devuelve el string ORB. Este string se pasa al ORB usando el método **ORB.string_to_object**, y el ORB devuelve la referencia al objeto remoto:

```
URL iorserver = new URL(
      "http://server.com/servlet?object=registration");
URLConnection con = ioserver.openConnection();
BufferedReader br = new BufferReader(
      new InputStreamReader(con.getInputStream));
String ref = br.readLine();
org.omg.CORBA.Object cobj = orb.string_to_object(ref);
RegistrationHome regHome =
      RegistrationHomeHelper.narrow(cobj);
```

El nombre sustituto del servidor puede mantener registros persistentes IOR que pueden sobrevivir a paradas si es necesario.

Remote Method Invocation (RMI)

El API "Remote Method Invocation" (RMI) originalmente usaba su propio protocolo de comunicación llamado "Java Remote Method Protocol" (JRMP), que resultaba en tener su propio servicio de búsqueda. Las nuevas versiones de RMI pueden usar el protocolo IIOP, además de JRMP, RMI-IIOP se cubre en la siguiente sección.

El servicio de nombrado del JRMP RMI es similar a otros servicios de búsqueda y nombrado. La búsqueda real se consigue llamando a **Naming.lookup** y pasándole un parámetro URL a este método. La URL especifica el nombre de la máquina, y opcionalmente el puerto donde está el servidor de nombres, **rmiregistry**, que sabe que objeto se está ejecutando, y el objeto remoto que queremos referenciar para llamar a sus métodos.

Por ejemplo:

```
SellerHome shome =
  (SellerHome)Naming.lookup(
  "rmi://appserver:1090/seller");
```

Este código devuelve la referencia remota de **SellerHome** desde el objeto unido al nombre **seller** en la máquina llamada **appserver**. La parte **rmi** de la URL es opcional y podríamos haber visto URLs RMI sin ella, pero si estámos usando JNDI o RMI-IIOP, incluir **rmi** en la URL nos ahorra confusiones posteriores. Una vez que tenemos la referencia a **SellerHome**, podemos llamar a sus métodos.

En contraste con la búsqueda JNDI realizada por **AuctionServlet.java**, que requiere una búsqueda de dos estados para crear un contexto y luego la búsqueda real, RMI inicializa la conexión con servidor de nombres RMI, **rmiregistry**, y también obtiene la referencia remota con una llamada.

Esta referencia remota será el cliente inquilino de **rmiregistry**. Inquilino significa que a menos que el cliente informe al servidor de que todavía necesita una referencia al objeto, el alquiler expira y la memoria es liberada. Esta operación de alquiler es transparente para el usuario, pero puede ser ajustada seleccionando el valor de la propiedad **java.rmi.dgc.leaseValue** en el servidor, en milisegundos cuando se arranca el servidor de esta forma:

```
java -Djava.rmi.dgc.leaseValue=120000 myAppServer
```

RMI sobre Internet Inter-ORB Protocol (IIOP)

La ventaja de RMI sobre "Internet Inter-ORB Protocol " (IIOP), significa que el código RMI existente puede referenciar y buscar un objeto con el servicio **CosNaming** de CORBA. Esto nos ofrece una gran interoperatividad entre arquitecturas con un pequeño cambio en nuestro código RMI existente.

Nota: El compilador **rmic** proporciona la opción **-iiop** para generar el stub y las clases tie necesarias para RMI-IIOP.

Servidor IIOP

El protocolo RMI-IIOP se implementa como un plug-in JNDI, por lo que como antes, necesitamos crear un **InitialContext**:

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
      "com.sun.jndi.cosnaming.CNCtxFactory");
env.put("java.naming.provider.url",
      "iiop://localhost:1091");
Context ic = new InitialContext(env);
```

La factoría de nombres debería parecer familiar como el mismo servicio de nombres usado en la sección CORBA. La principal diferencia es la adicción de un valor URL especificando el servicio de nombres al que conectarse. El servicio de nombres usado aquí es el programa **tnameserv** arrancado en el puerto 1091:

```
tnameserv -ORBInitialPort 1091
```

El otro cambio principal en el lado del servidor es reemplazar las llamadas a **Naming.rebind** para usar el método **rebind** de JNDI en el ejemplar **InitialContext**. Por ejemplo:

Viejo código RMI:

SellerHome shome=(SellerHome)Naming.lookup("rmi://appserver:1090/seller");

Nuevo código RMI:

Hashtable env = new Hashtable(); env.put("java.naming.factory.initial", "com.sun.jndi.cosnaming.CNCtxFactory"); env.put("java.naming.provider.url", "iiop://localhost:1091"); Context ic = new InitialContext(env); SellerHome shome= (SellerHome)PortableRemoteObject.narrow(ic.lookup("seller"), SellerHome)

Ciente IIOP

En el lado del cliente, la búsqueda RMI se cambia para usar un ejemplar del **InitialContext** en lugar del **Naming.lookup** de RMI. El objeto devuelto es mapeado al objeto requerido usando el método **narrow** de la clase **javax.rmi.PortableRemoteObject**. **PortableRemoteObject** reemplaza **UnicastRemoteObject** que estaba disponible anteriormente en código de servidor RMI.

Viejo código de búsqueda RMI:

```
SellerHome shome= new SellerHome("seller");
Naming.rebind("seller", shome);
```

Nuevo código RMI:

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
     "com.sun.jndi.cosnaming.CNCtxFactory");
env.put("java.naming.provider.url",
     "iiop://localhost:1091");
Context ic = new InitialContext(env);

SellerHome shome= new SellerHome("seller");
ic.rebind("seller", shome);
```

El **PortableRemoteObject** reemplaza al **UnicastRemoteObject** disponible anteriormente en el código del servidor RMI. El código RMI debería extender **UnicastRemoteObject** o llamar al método **exportObject** de la clase **UnicastRemoteObject**. **PortableRemoteObject** También contiene un método **exportObject** equivalente. En la implementación actual, es mejor eliminar explícitamente los objetos no utilizados mediante llamadas a **PortableRemoteObject.unexportObject()**.

Servicios de Búsqueda JINI

(Para hacerlo más tarde)

Aumentar el Rendimiento de la Búsqueda

Cuando ejecutemos nuestra aplicación, si encontramos que llevar el objeto a otro ordenador a través de un diskette será más rápido, es que tenemos un problema de configuración de la red. La fuente del problema es cómo se resuelven los nombres de host y las direcciones IP, y aquí tenemos un atajo.

RMI y otros servicios de nombres usan la clase **InetAddress** para resolver los nombres de host y direcciones IP. **InetAddress** almacena los resultados para mejorar las llamadas subsecuentes, pero cuando se le pasa una nueva dirección IP o un nombre de servidor, realiza una referencia cruzada entre la dirección IP y el nombre del host. Si suministramos el nombre del host como una dirección IP, **InetAddress** todavía intentará verificar el nombre del host.

Para evitar este problema, incluimos el nombre del host y la dirección IP en un fichero host en el cliente.

Sistemas Unix: En Unix, el fichero host normalmente es **/etc/hosts**.

Windows: En windows 95 ó 98, el fichero host es **c:\windows\hosts**, (el fichero **hosts.sam** es un fichero de ejemplo). En windows NT, el fichero host es **c:\winnt\system32\drivers\etc\hosts**.

Todo lo que necesitamos hacer es poner estas líneas en nuestro ficheo host. Las entradas **myserver1** y **myserver2** son los host donde se ejecutan el servidor remoto y **rmiregistry**

```
127.0.0.1  localhost
129.1.1.1  myserver1
129.1.1.2  myserver2
```

Invocación Remota de Métodos

El API de Invocación Remota de Métodos (RMI) permite las comunicaciones entre cliente y servidor a través de la red entre programas escritos en Java. El servidor de JavaBeans Enterprise implementa de forma transparente el código RMI necesario para que el programa cliente pueda referenciar a los Beans Enterprise que se ejecutan en el servidor y acceder a ellos como si se estuvieran ejecutando localmente en el programa cliente.

El tener el RMi incluido internamente el servidor JavaBeans de Enterprise es muy conveniente y nos ahorra tiempo de codificación, pero si necesitamos usar características avanzadas de RMI o integrar RMI con una aplicación existente, necesitamos sobreescribir la implementación por defecto RMI y escribir nuestro propio código RMI.

El capítulo reemplaza el **RegistrationBean** manejado por contenedor del Capítulo 2: Beans de Entidad y de Sesión con un servidor de registro basado en RMI. El Bean **SellerBean** del capítulo 2, también se modifica para llamar al nuevo servidor de registro RMI usando una llamada a **lookup** de Java 2 RMI.

Sobre RMI

El API RMI nos permite accede a un servidor de objetos remoto desde un programa cliente haciendo sencillas llamadas a métodos del servidor de objetos. Mientras que otras arquitecturas distribuidas para acceder a servidores de objetos remotos como "Distributed Component Object Model" (DCOM) y "Common Object Request Broker Architecture" (CORBA) devuelven referencias al objeto remoto, el API RMI no sólo devuelve referencias, si no que proporciona beneficios adicionales.

- El API RMI maneja referencias a objetos remotos (llamadas por referencia) y también devuelve una copia del objeto (llamada por valor).
- Si el programa cliente no tiene acceso local a la clase para la que se ejemplarizó un objeto remoto, los servicios RMI pueden descargar el fichero class.

Serialización y colocación de Datos

Para transferir ojbjetos, el API RMI usa el API Serialization para empaquetar (colocar) y desempaquetar (descolocar) los objetos. Para colocar un objeto, el API Serialization convierte el objeto a un Stream de bytes, y para descolocar el objeto, el API Serialization convierte un stream de bytes en un objeto.

RMI sobre IIOP

Una de las desventajas iniciales del RMI era que la única relación con la plataforma Java para escribir interfaces hacen díficil la intregración con sistemas legales existentes. Sin embargo, RMI sobre "Internet Inter-ORB Protocol" (IIOP) explicado en el Capítulo 4: Servicios de Búsqueda permite a RMI comunicarse con cualquier sistema o lenguaje que soporte CORBA.

Si combinamos la integración mejorada con la habilidad de RMI para trabajar a través de firewalls usando proxies HTTP, podríamos encontrar distribuciones para la lógica de nuestro negocio usando RMI más fáciles que una solución basada en sockets.

> **Nota:** La transferencia de código y datos son partes clave de la especificación JINI. De hecho, si añadieramos un servicio de uniones a los servicios RMI crearíamos algo muy similar a los que obtenemos con la arquitectura JINI.

RMI en la aplicación de Subastas

El RegistrationServer basado en RMI tiene los siguientes métodos nuevos:

- Un nuevo método **create** para crear un nuevo usuario.
- Un nuevo método **find** para buscar un usuario.

- Un nuevo método **search** para la búsqueda personalizada de usuarios en la base de datos.

La nueva búsqueda personalizada sevuelve los resultados al cliente llamante mediante una llamada a un Callbak RMI. Está búsqueda es similar a los métodos **finder** usados en los Beans de ejemplos usados en los capítulos 2 y 3, excepto en que la versión RMI, puede tardar más tiempo en generar los resultados porque el sevidor de registros remoto llama al método remoto exportado por el cliente SellerBean basado en RMI.

Si el cliente llamante está escrito en Java, y no es, por ejemplo, una página web, el servidor puede actualizar el cliente tan pronto como los resultados estuvieran listos. Pero, el protocolo HTTP usado en la mayoría de los navegadores no permite que los resultados sean enviados sin que haya una petición. Esto significa que el resultado de una página web no se crea hasta que los resultados estén listos, lo que añade un pequeño retraso.

Introducción a las Clases

Las dos clases principales en la implementación de la subasta basada en RMI son SellerBean y el remoto RegistrationServer. **SellerBean** es llamado desde AuctionServlet para insertar un ítem para la subasta en la base de datos, y chequear balances negativos en las cuentas.

Los modelos de ejemplo de la arquitectura JavaBeans Enterprise en los que los detalles de registro del usuario se han separado del código para crear y encontrar detalles de registro. Es decir, los detalles de registro de usuario proporcionados por la clase Registration.java se separan del código para crear y encontrar un objeto **Registration**, que está en la clase RegistrationHome.java.

La implementación del interface remoto de RegistrationHome.java está unida al **rmiregistry**. Cuando un programa cliente quiere manipular detalles del registro del usuario, primero tiene que buscar la referencia al objeto RegistrationHome.java en el **rmiregistry**.

Sumario de Ficheros

Todo los fciheros de código fuente para el ejemplo basado en RMI se describen en la siguiente lista.

- SellerBean.java: Programa cliente que llama a los métodos remotos **RegistrationServer.verifypasswd** y **RegistrationServer.findLowCreditAccounts**. **SellerBean** también exporta su método **updateResults** que llama a **RegistrationServer** cuando completa su búsqueda **RegistrationServer.findLowCreditAccounts**.
- RegistrationServer.java: Servidor de objetos remotos que implementa los interfaces remotos **RegistrationHome** y **Registration**.
- Registration.java: Interface remoto que declara los métodos remotos **getUser**, **verifypasswd**, y otros métodos para el manejo de los detalles de registro del usuario.
- RegistrationHome.java: Interface remoto que declara los métodos remotos **create**, **findByPrimaryKey**, y **findLowCreditAccounts** que crean o devuelven ejemplares de detalles de registro.
- RegistrationImpl.java: El fichero fuente **RegistrationServer.java** incluye la implementación para el interface remoto **Registration** como la clase **RegistrationImpl**.
- RegistrationPK.java: Clase que representa los detalles de registro de usuario usando sólo la clave primaria del registro de la base de datos.
- ReturnResults.java: Interface remoto que declara el método **updateResults** la clase **SellerBean** lo implementa como callback.
- AuctionServlet.java: Versión modificada de la clase original **AuctionServlet** conde las cuentas de registro se crean mediante llamadas directas al **RegistrationServer** de RMI. El servelt de subasta también llama al método **SellerBean.auditAccounts**, que devuelve una lista de los usuarios con un bajo balance en la cuenta.

  El método **auditAccounts** es llamado con la siguiente URL, donde hace un simple chequeo para verificar que la petición viene del host local.

```
http://phoenix.eng.sun.com:7001/
        AuctionServlet?action=auditAccounts
```

También necesitaremos un fichero de policía java.policy para conceder los permisos necesarios para ejecutar el ejemplo en plataformas Java 2.

La mayoría de las aplicaciones RMI necesitan dos permisos socket, para accesos a los socket y a HTTP para especificar los puertos. Los dos permisos de threads fueron listados en una pila cuando sea necesario por la clase **RegistrationImpl** para crear un thread interno.

En la plataforma Java 2, cuando un programa no tiene todos los permisos que necesita, la "Máquina Virtual Java" genera una pila de seguimiento que lista los permisos que necesitan ser añadidos al fichero de policía de seguridad.

```
grant {
  permission java.net.SocketPermission
     "*:1024-65535", "connect,accept,resolve";
  permission java.net.SocketPermission "*:80",
                        "connect";
  permission java.lang.RuntimePermission
                         "modifyThreadGroup";
  permission java.lang.RuntimePermission
                         "modifyThread";
};
```

Compilar el Ejemplo

Antes de describir el código basado en RMI de las clases anteriores, aquí está la secuencia de comandos para compilar el ejemplo en las plataformas Unix y Win32:

```
Unix:
javac registration/Registration.java
javac registration/RegistrationPK.java
javac registration/RegistrationServer.java
javac registration/ReturnResults.java
javac seller/SellerBean.java
rmic -d . registration.RegistrationServer
rmic -d . registration.RegistrationImpl
rmic -d . seller.SellerBean

Win32:
javac registration\Registration.java
javac registration\RegistrationPK.java
javac registration\RegistrationServer.java
javac registration\ReturnResults.java
javac seller\SellerBean.java
rmic -d . registration.RegistrationServer
rmic -d . registration.RegistrationImpl
rmic -d . seller.SellerBean
```

Arrancar el Registro RMI

Como estamos usando nuestro propio código RMI, tenemos que arrancar explícitamente el RMI Registry para que el objeto **SellerBean** pueda encontrar los Beans remotos de Enterprise. El **RegistrationServer** usa el registro RMI para registrar o unir los Beans enterprise que pueden ser llamados de forma remota. El cliente **SellerBean** contacta con el registro para buscar y obtener las referencias a los Beans **AuctionItem** y **Registration**.

Como RMI permite que el código y los datos sean transferidos, debemos asegurarnos que el sistema **classloader** no carga clases extras que puedan ser enviadas erróneamente al cliente. En este ejemplo, las clases extras podrían ser las clases Stub y Skel, y las clases **RegistrationSever** y **RegistrationImpl**, y para evitar que lo sean cuando arrancamos el registro RMI. Como el path actual podría ser incluido automáticamente, necesitamos arrancar el RMI Registry desde fuera del espacio de trabajo.

Los sigueintes comandos evitan el envío de clases extras, desconfigurando la variable **CLASSPATH** antes de arrancar el Registro RMI en el puerto 1099. Podemos especificar un puerto diferente añadiendo el número de puerto de esta forma: **rmiregistry 4321 &**. Si cambiamos el número de puerto debemos poner el mismo número en las llamadas al cliente <**lookup** y al servidor **rebind**.

```
Unix:
export CLASSPATH=""
rmiregistry &

Win32:
unset CLASSPATH
start rmiregistry
```

Arrancar el Servidor Remoto

Una vez que **rmiregistry** se está ejecutando, podemos arrancar el servidor remoto, **RegistrationServer**. El programa **RegistrationServer** registra el nombre **registration2** con el servidor de nombres **rmiregistry**, y cualquier cliente puede usar este nombre para recuperar una referencia al objeto remoto, **RegistrationHome**.

Para ejecutar el ejemplo, copiamos las clases**RegistrationServer** y **RegistrationImpl** y las clases stub asociadas a un área accesible de forma remota y arrancamos el programa servidor.

```
Unix:
cp *_Stub.class
        /home/zelda/public_html/registration
cp RegistrationImpl.class
/home/zelda/public_html/registration
cd /home/zelda/public_html/registration
java -Djava.server.hostname=
       phoenix.eng.sun.com RegistrationServer

Windows:
copy *_Stub.class
        \home\zelda\public_html\registration
copy RegistrationImpl.class
        \home\zelda\public_html\registration
cd \home\zelda\public_html\registration
```

```
java -Djava.server.hostname=
        phoenix.eng.sun.com RegistrationServer
```

Las siguientes propiedades se usan para configurar los clientes y servidores RMI. Estas propiedades pueden seleccionarse dentro del programa o suministrarlas como propiedades en la línea de comandos para la JVM.

- La propiedad **java.rmi.server.codebase** especifica dónde se localizan las clases accesibles públicamente. En el servidor esto puede ser un simple fichero URL para apuntar al directorio o fichero JAR que contiene las clases. Si el URL apunta a un directorio, debe terminar con un carácter separador de ficheros , "/".

  Si no usamos un fichero URL, tampoco necesitaremos un servidor HTTP para descargar las clases remotas o tener que enviar manualmente el stub del cliente y las clases de interfaces remotos, por ejemplo, un fichero JAR.
- La propiedad **java.rmi.server.hostname** es el nombre completo del host del servidor donde residen las clases con acceso público. Esto es sólo necesario si el servidor tiene problemas para generar por sí mismo un nombre totalmente cualificado.
- La propiedad **java.rmi.security.policy** especifica el policy file con los permisos necesarios para ejecutar el objeto servidor remoto y para acceder a la descarga de las clases del servidor remoto.

Establecer Comunicaciones Remotas

Los programas clientes se comunican unos con otros a través del servidor. El programa servidor consiste en tres ficheros. Los ficheros de interfaces remotos **Registration.java** y **RegistrationHome.java** definen los métodos que pueden ser llamados de forma remota, y el fichero **RegistrationServer.java** de clase define las clases **RegistrationServer** y **RegistrationImpl** que implementan los métodos.

Para establecer comunicaciones remotas, tanto el programa cliente como el servidor necesitan acceder a las clases del interface remoto. El servidor necesita las clases del interface para generar la implementación del interface, y el cliente usa el interface remoto para llamar a las implementaciones de los métodos del servidor remoto.

Por ejemplo, **SellerBean** crea una referencia a el interface **RegistrationHome**, y no **RegistrationServer**, la implementación, cuando necesita crear un regisro de usuario.

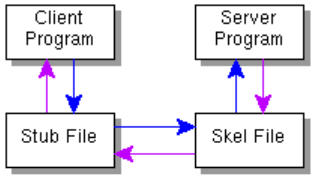
Junto con los interfaces del servidor y las clases, necesitamos las clases Stub y Skel para establecer comunicaciones remotas. Estas clases se generan cuando ejecutamos el comando del compilador **rmic** sobre las clases **RegistrationServer** y **SellerBean**.

Las clases **SellerBean**, **SellerBean_Stub.class** y **SellerBean_Skel.class** generadas son necesarias para la llamada desde el servidor hasta el cliente **SellerBean**. Es el fichero **_Stub.class** en el cliente que coloca y descoloca los datos desde el servidor, mientras que la clase **_Skel.class** hace los mismo en el servidor.

Nota: En la plataforma Java 2, el fichero del lado delservidor, **_Skel.class** ya no es necesario porque sus funciones han sido reemplazadas por las clases de la "Java Virtual Machine".

Colocar Datos

Colocar y descolocar los datos significa que cuando llamamos al método **RegistrationHome.create** desde **SellerBean**, esta llamada es reenviada al método **RegistrationServer_Stub.create**. El método **RegistrationServer_Stub.create** envuelve los argumentos del método y los envía a un stream serializado de bytes para el método **RegistrationServer_Skel.create**.


Client Program
Server Program
Stub File
Skel File

método **RegistrationServer_Skel.create** desenvuelve el stream de bytes serializado, re-crea los argumentos de la llamada original a **RegistrationHome.create**, y devuelve el resultado de la llamada real **RegistraionServer.create** de vuelta, junto con la misma ruta, pero esta vez, se empaquetan los datos en el lado del servidor.

Colocar y descolocar los datos tiene sus complicaciones. El primer problema son los objetos serializados que podrían ser incompatiles entre versiones del JDK. Un objeto serializado tiene un identificador almacenado con el objeto que enlaza el objeto serializado con su versión. Si el cliente RMI y el servidor son incompativles con su ID de serie, podríamos necesitar generar Stubs y Skels compatibles usando la opción **-vcompat** del compilador **rmic**.

Otro problema es que no todos los objetos son serializables por defecto. El objeto inicial **RegistrationBean** está basado en la devolución de un objeto **Enumeration** que contiene elementos **Registration** en un **Vector**. Devolver la lista desde el método remoto, funciona bien, pero cuando intentamos envíar un vector como un parámetro a un objeto remoto, obtendremos una excepción en tiempo de ejecución en la plataforma Java 2.

Afortunadamente, en el API Collections, la plataforma Java ofrece alternativas a la descolocación de objetos anterior. En este ejemplo, un **ArrayList** del API Collections reemplaza el **Vector**. Si el API Collections no es una opción, podemos crear una clase envoltura que extienda **Serializable** y proporcione implementaciones para los métodos **readObject** y **writeObject** para convertir el objeto en un stream de bytes.

La clase RegistrationServer

La clase RegistrationServer extiende **java.rmi.server.UnicastRemoteObject** e implementa los métodos **create**, **findByPrimaryKey** y **findLowCreditAccounts** declarados en el interface **RegistrationHome**. El fichero fuente RegistrationServer.java también incluye la implementación del interface remoto **Registration** como la clase **RegistrationImpl**. **RegistrationImpl** también extiende **UnicastRemoteObject**.

Exportar un Objeto Remoto

Cualquier objeto que querramos que se accesible remotamente necesita extender el interface **java.rmi.server.UnicastRemoteObject** o usar el método **exportObject** de la clase **UnicastRemoteObject**. Si extendemos **UnicastRemoteObject**, también obtendremos los métodos **equals**, **toString** y **hashCode** para el objeto exportado.

Pasar por Valor y por Referencia

Aunque la clase **RegistrationImpl** no está unida al registro, todavía está referenciada remotamente porque está asociada con los resultados devueltos por **RegistrationHome**. **RegistrationImpl** extiende **UnicastRemoteObject**, sus resultados son pasados por referencia, y sólo una copia del Bean de registro del usuario existente en la Java VM a la vez.

En el caso de reportar resultados como en el método **RegistrationServer.findLowCreditAccounts**, la clase **RegistrationImpl** se puede usar una copia del objeto remoto. Si no extendemos la clase **UnicastRemoteObject** en la definición de la clase **RegistrationImpl**, se devolverá un nuevo objeto **Registration** en cada petición. En efecto los valores son pasados pero no la referencia al objeto en el servidor.

Recolección de Basura Distribuida

Al usar referencias remotas a objetos en el servidor desde fuera del cliente el recolector de basura del servidor introduce algunos problemas potenciales con la memoria. ¿Cómo conoce el servidor cuando se mantiene una referencia a un objeto **Registration** que no está siendo usado por ningún cliente porque abortó o se cayó la conexión de red?

Para evitar bloqueos de memoria en el servidor desde los clientes, RMI usa un mecanismo de alquiler cuando ofrecen las referencias a los objetos exportados. Cuando se exporta un objeto, la JVM incrementa la cuenta del número de referencias a este objeto y configura el tiempo de expiración, o tiempo de préstamo, por el número de referencias del objeto.

Cuando el alquiler expira, la cuenta de referencias de este objeto se decrementa y si alcanza 0, el objeto es seleccionado para la recolección de basura por la JVM. Hay que configurar el cliente que mantiene un pico de referencia al objeto remoto a que renueve el alquiler si necesita el objeto más alla del tiempo de alquiler. Este pico de referencia es una forma de referirse a un objeto en la memoria sin mantenerlo lejos del recolector de basura.

Este tiempo de alquiler es una propiedad configurable medida en segundos. Si tenemos una red rápida, podríamos acortar el valor por defecto, y crear un gran número de referencias a objetos transitorias.

El siguiente código selecciona el tiempo de alquiler a 2 minutos.

```
        Property prop = System.getProperties();
        prop.put("java.rmi.dgc.leaseValue", 120000);
```

Los métodos **create** y **findByPrimaryKey** son prácticamente idénticos a las otras versiones del servidor Registration. La principal diferecia es que en el lado del servidor, el registro registration es referenciado como **RegistrationImpl**, que es la implementación de **Registration**. En el lado del cliente, se usa **Registration** en su lugar.

El método **findLowCreditAccounts** cosntruye un **ArrayList** de objetos **RegistrationImpl** serializables y llama al método remoto en la clase **SellerBean** para pasar el resultado de vuelta. Los resultados on generado por una clase **Thread** interna porque el método retorna antes de que el resultado esté completo. El objeto **SellerBean** espera a que sea llamado el método **updateAccounts** antes de mostrar la página HTML. En un cliente escrito en Java, no sería necesario esperar, podríamos mostrar la actualización en tiempo real.

```
public class RegistrationServer
        extends UnicastRemoteObject
        implements RegistrationHome {

  public registration.RegistrationPK
    create(String theuser,
        String password,
```

```
      String emailaddress,
      String creditcard)
        throws registration.CreateException{
  // code to insert database record
}

public registration.Registration
  findByPrimaryKey(registration.RegistrationPK pk)
      throws registration.FinderException {
    if ((pk == null) || (pk.getUser() == null)) {
          throw new FinderException ();
    }
    return(refresh(pk));
}

private Registration refresh(RegistrationPK pk)
      throws FinderException {

  if(pk == null) {
    throw new FinderException ();
  }

  Connection con = null;
  PreparedStatement ps = null;
  try{
    con=getConnection();
    ps=con.prepareStatement("select password,
      emailaddress,
      creditcard,
      balance from registration where theuser = ?");
    ps.setString(1, pk.getUser());
    ps.executeQuery();
    ResultSet rs = ps.getResultSet();
    if(rs.next()) {
      RegistrationImpl reg=null;
      try{
        reg= new RegistrationImpl();
      }catch (RemoteException e) {}
        reg.theuser = pk.getUser();
        reg.password = rs.getString(1);
        reg.emailaddress = rs.getString(2);
        reg.creditcard = rs.getString(3);
        reg.balance = rs.getDouble(4);
        return reg;
    }else{
        throw new FinderException ();
    }
  }catch (SQLException sqe) {
         throw new FinderException();
  }finally {
    try{
      ps.close();
      con.close();
    }catch (Exception ignore) {}
    }
}

public void findLowCreditAccounts(
              final ReturnResults client)
      throws FinderException {
  Runnable bgthread = new Runnable() {
    public void run() {
      Connection con = null;
      ResultSet rs = null;
```

```
            PreparedStatement ps = null;
            ArrayList ar = new ArrayList();

            try{
              con=getConnection();
              ps=con.prepareStatement("select theuser,
                 balance from registration
                 where balance < ?");
              ps.setDouble(1, 3.00);
              ps.executeQuery();
              rs = ps.getResultSet();
              RegistrationImpl reg=null;
              while (rs.next()) {
                try{
                  reg= new RegistrationImpl();
                }catch (RemoteException e) {}
                  reg.theuser = rs.getString(1);
                  reg.balance = rs.getDouble(2);
                  ar.add(reg);
                }
              rs.close();
              client.updateResults(ar);
              }catch (Exception e) {
                System.out.println("findLowCreditAccounts: "+e);
                return;
              }
              finally {
                try{
                  if(rs != null) {
                    rs.close();
                  }
                  if(ps != null) {
                    ps.close();
                            }
                  if(con != null) {
                    con.close();
                  }
              }catch (Exception ignore) {}
            }
          } //run
        };
        Thread t = new Thread(bgthread);
        t.start();
      }
    }
```

El método **main** carga el driver JDBC. Esta versión usa la base de datos Postgres, instala el **RMISecurityManager**, y contacta con el registro RMI para unir el objeto remoto **RegistrationHome** al nombre **registration2**. No necesita unir el interface remoto, **Registration** porque la clase es cargada cuando es referenciada por **RegistrationHome**.

Por defecto, el servidor de nombres usa el puerto 1099. Si queremos usar un número de puerto diferente, podemos añadirlo con dos puntos de esta forma: **kq6py:4321**. Si cambiamos aquí el número de puerto, debemos arrancar el RMI Registry con el mismo número de puerto.

El método **main** también instala un **RMIFailureHandler**. Si el servidor falla al crear el socket servidor, el manejador de fallos devuelve **true** que instruye al servidor RMI para que reintente la operación.

```
    public static void main(String[] args){
       try {
          new pool.JDCConnectionDriver(
                  "postgresql.Driver",
                  "jdbc:postgresql:ejbdemo",
                  "postgres", "pass");
          } catch (Exception e){
             System.out.println(
               "error in loading JDBC driver");
             System.exit(1);
          }
```

```
            try {
               Properties env=System.getProperties();
               env.put("java.rmi.server.codebase",
                 "http://phoenix.eng.sun.com/registration");
               RegistrationServer rs=
                 new RegistrationServer();
               if (System.getSecurityManager() == null )  {
                   System.setSecurityManager(
                    new RMISecurityManager());
               }
               RMISocketFactory.setFailureHandler(
                    new RMIFailureHandlerImpl());

               Naming.rebind("
                    //phoenix.eng.sun.com/registration2",rs);
            }catch (Exception e) {
               System.out.println("Exception thrown "+e);
            }
         }
      }

      class RMIFailureHandlerImpl
              implements RMIFailureHandler {
         public boolean failure(Exception ex ){
            System.out.println("exception "+ex+" caught");
            return true;
         }
      }
```

Interface Registration

El interface Registration declara los métodos implementados por **RegistrationImpl** en el fichero fuente **RegistrationServer.java**.

```
package registration;

import java.rmi.*;
import java.util.*;

public interface Registration extends Remote {
   boolean verifyPassword(String password)
               throws RemoteException;
   String getEmailAddress() throws RemoteException;
   String getUser() throws RemoteException;
   int adjustAccount(double amount)
         throws RemoteException;
   double getBalance() throws RemoteException;
}
```

Interface RegistrationHome

El interface RegistrationHome declara los métodos implementados por la clase **RegistrationServer**. Estos métodos reflejan el interface Home definido en el ejemplo JavaBeans de Enterprise. El método **findLowCreditAccounts** toma un interface remoto como su único parámetro.

```
    package registration;

    import java.rmi.*;
    import java.util.*;

    public interface RegistrationHome extends Remote {
      RegistrationPK create(String theuser,
            String password,
            String emailaddress,
            String creditcard)
                    throws CreateException,
```

```
                RemoteException;

  Registration findByPrimaryKey(RegistrationPK theuser)
                throws FinderException, RemoteException;

  public void findLowCreditAccounts(ReturnResults rr)
                throws FinderException, RemoteException;
}
```

Interface ReturnResults

El interface ReturnResults declara el método implementado por la clase **SellerBean**. El método **updateResults** es llamado desde **RegistrationServer**.

```
package registration;

import java.rmi.*;
import java.util.*;

public interface ReturnResults extends Remote {
    public void updateResults(ArrayList results)
        throws FinderException, RemoteException;
}
```

La Clase SellerBean

La clase SellerBean incluye la implementación del método callback y llama al objeto **RegistrationServer** usando RMI. El método **updateAccounts** se hace accesible mediante una llamada a **UnicastRemoteObject.exportObject(this);**. El método **auditAccounts** espera un objeto method **Boolean**.

El método **updateAccounts** envía una notificación a todos los métodos que esperan el objeto **Boolean** cuando ha sido llamado desde el servidor y recibe los resultados.

```
package seller;

import java.rmi.RemoteException;
import java.rmi.*;
import javax.ejb.*;
import java.util.*;
import java.text.NumberFormat;
import java.io.Serializable;
import javax.naming.*;
import auction.*;
import registration.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;


public class SellerBean
        implements SessionBean, ReturnResults {

    protected SessionContext ctx;
    javax.naming.Context ectx;
    Hashtable env = new Hashtable();
    AuctionServlet callee=null;
    Boolean ready=new Boolean("false");
    ArrayList returned;

    public int insertItem(String seller,
        String password,
        String description,
        int auctiondays,
        double startprice,
        String summary)
                throws RemoteException {

      try{
```

```
    RegistrationHome regRef = (
            RegistrationHome)Naming.lookup(
            "//phoenix.eng.sun.com/registration2");
    RegistrationPK rpk= new RegistrationPK();
    rpk.setUser(seller);
    Registration newseller = (
          Registration)regRef.findByPrimaryKey(rpk);
    if((newseller == null) ||
          (!newseller.verifyPassword(password))) {
        return(Auction.INVALID_USER);
    }

    AuctionItemHome home = (
            AuctionItemHome) ectx.lookup(
                              "auctionitems");
    AuctionItem ai= home.create(seller,
            description,
            auctiondays,
            startprice,
            summary);
    if(ai == null) {
      return Auction.INVALID_ITEM;
    }else{
      return(ai.getId());
    }
    }catch(Exception e){
      System.out.println("insert problem="+e);
      return Auction.INVALID_ITEM;
    }
}

public void updateResults(java.util.ArrayList ar)
            throws RemoteException {
   returned=ar;
   synchronized(ready) {
      ready.notifyAll();
   }
}

public ArrayList auditAccounts() {
    this.callee=callee;
    try {
       RegistrationHome regRef = (
            RegistrationHome)Naming.lookup(
            "//phoenix.eng.sun.com/registration2");
       regRef.findLowCreditAccounts(this);
       synchronized(ready) {
          try {
             ready.wait();
          } catch (InterruptedException e){}
       }
    return (returned);
   }catch (Exception e) {
      System.out.println("error in creditAudit "+e);
   }
  return null;
}

public void ejbCreate()
            throws javax.ejb.CreateException,
            RemoteException {
  env.put(
    javax.naming.Context.INITIAL_CONTEXT_FACTORY,
    "weblogic.jndi.TengahInitialContextFactory");
```

```
        try{
          ectx = new InitialContext(env);
        } catch (NamingException e) {
          System.out.println(
            "problem contacting EJB server");
          throw new javax.ejb.CreateException();
        }
          Properties env=System.getProperties();
          env.put("java.rmi.server.codebase",
            "http://phoenix.eng.sun.com/registration");
          env.put("java.security.policy","java.policy");
          UnicastRemoteObject.exportObject(this);
      }

      public void setSessionContext(SessionContext ctx)
                  throws RemoteException {
        this.ctx = ctx;
      }

      public void unsetSessionContext()
                  throws RemoteException {
          ctx = null;
      }

      public void ejbRemove() {}
      public void ejbActivate() throws RemoteException {
          System.out.println("activating seller bean");
      }
      public void ejbPassivate() throws RemoteException {
          System.out.println("passivating seller bean");
      }
  }
```

_____

[1] Cuando se usan en toda esta site, los términos, "Java virtual machine" o "JVM" significan una máquina virtual de la plataforma Java.

Ozito

Common Object Request Broker Architecture (CORBA)

Las implementaciones de RMI y de JavaBeans Enterprise de la aplicación de subasta usan el lenguaje Java para implementar los distintos servicios de la subasta. Sin embargo, podríamos necesitar intergrarlo con aplicaciones escritas en C, C++ u otros lenguajes y ejecutarlo en un millar de sistemas operativos y máquinas distintas.

Una forma de integración con otras aplicciones es transmitir datos en un formato común como caracteres de 8 bits sobre sockets TCP/IP. La desventaja es tener que gastar mucho tiempo en derivar un mensaje de protocolo y mapeado de varias estructuras de datos hacia y desde el formato de transmisión común para que los datos puedan ser enviados y recibidos sobre la conexión TCP/IP.

Aquí es donde pueden ayudar el "Common Object Request Broker Architecture" (CORBA) y su "Interface Definition Language" (IDL). IDL proporciona un formato común para representar un objeto que puede ser distribuido a otras aplicaciones. Las otras aplicaciones podrían incluso no entender de objetos, pero mientras puedan proporcionar un mapeado entre el formato común IDL y sus propia representación de datos, la aplicación podrá compartir los datos.

Este capítulo describe el esquema de mapeo de IDL a lenguaje Java, y cómo reemplazar el original **RegistrationBean** basado en contenedor controlador por su equivalente servidor CORBA. Los programas **SellerBean.java** y **AuctionServlet.java** también se modifican para interoperar con el programa CORBA **RegistrationServer**.

Esquema de Mapeo IDL

Muchos lenguajes de programación proporcionan un mapeo entre sus tipos de datos y el formato común denominado IDL, y el lenguaje Java no es una excepción. El lenguaje Java puede enviar objetos definidos por IDL a otras aplicaciones distribuidas CORBA y recibir objetos definidos mediante IDL desde otras aplicaciones distribuidas CORBA.

Esta sección descrbie el esquema de mapedo de Java a IDL y, cuando sea necesario, presenta problemas que debemos tener en consideración.

Referencia Rápida

Aquí tenemos una tabla de referencia rápida de los tipos de datos del lenguaje Java y los de IDL CORBA, y las excepciones de tiempo de ejecución que se pueden lanzar cuando la conversión falla. Los tipos de datos de esta tabla que necesitan explicación se cubren más abajo.

Tipo de Dato Java	Formato IDL	Exception
byte	octet	
boolean	boolean	
char	char	DATA_CONVERSION
char	wchar	
double	double	
float	float	
int	long	
int	unsigned long	
long	long long	
long	unsigned long long	

short	short	
short	unsigned short	
java.lang.String	string	DATA_CONVERSION
java.lang.String	wstring	MARSHAL

Valores sin Signo: Los tipos de datos Java: **byte**, **short**, **int**, y **long** están representados por entereros de complemento a dos de 8, 16, 32 y 64 bits. Esto significa que un valor **short** Java representa un rango desde -2^{15} hasta $2^{15} - 1$ ó desde -32768 hasta 32767 inclusives. El tipo con signo equivalente IDL para un short, **short**, es igual en el rango, pero el tipo **short** IDL sin signo usa el rango desde 0 hata 2^{15} ó desde 0 hasta 65535.

Esto significa que en el caso de **short**, si un valor short sin signo mayor de 32767 es pasado a un programa escrito en Java, el valor **short** será representado como un número negativo. Esto puede causar confusión en los límites de test para valores mayores que 32767 o menores que 0.

Tipos char IDL: El lenguaje Java usa un unicode de 16 Bits, pero los tipos **char** y **string** de IDL son carcateres de 8 bits. Podemos mapear un **char** Java a un **char** IDL de 8 bits para transmitir caracteres multi-byte si usamos un array para hacerlo. Sin embargo, el tipo de caracter ancho de IDL **wchar** está especialmente diseñado para lenguajes con caracteres multi-bytes y aloja el número fijo de bytes que sea necesario para contener el conjunto del lenguaje para cada una de las letras.

Cuando se mapea desde el tipo **char** de Java al tipo **char** de IDL, se puede lanzar la excepción **DATA_CONVERSION** si el caracter no entra en los 8 bits.

Tipos string IDL: El tipo **string** IDL puede ser lanzado como una secuencia de tipos **char** IDL, también lanza la excepción **DATA_CONVERSION**. El tipo **wstring** IDL es equivalente a una secuencua de **wchars** terminada por un **wchar NULL**.

Un tipo **string** y un tipo **wstring** de IDL pueden tener un tamaño fijo o sin máximo definido. Si intentamos mapear un **java.lang.String** a un **string** IDL de tamaño fijo y el **java.lang.String** es demasidado largo, se lanzará una excepción **MARSHAL**.

Configurar el Mapeo IDL

El mapeo del lenguaje Java a IDL se sitúa en un fichero con extensión **.idl**. El fichero es compilado para que pueda ser accedido por los programas CORBA que necesitan enviar y recibir datos. Esta sección explica cómo construir los mapeos para las sentencias de paquete y los tipos de datos Java. La siguiente sección en Implementación CORBA de RegistrationServer describe cómo usar esta información para configurar el fichero de mapeo IDL para el servidor **Registration** CORBA.

Paquetes e Interfaces Java: Las sentencias de paquete Java son equivalentes al tipo **module** de IDL. Este tipo puede ser anidado, lo que resulta en que las clases Java generadas se crean en subdirectorios anidados.

Por ejemplo, si un programa CORBA contiene esta sentencia de paquete:

```
package registration;
```

el fichero de mapeo debería tener este mapeo a módulo IDL para ella:

```
module registration {
};
```

Si un programa CORBA contiene una herencia de paquete como esta:

```
package registration.corba;
```

su mapeo IDL de módulo será este:

```
module registration {
  module corba {
  };
};
```

Las clases distribuidas están definidas como interfaces Java y se mapean al tipo interface de IDL. IDL no define accesos como **public** o **private** que podríamos encontrar en el lenguaje Java, pero permite descender desde otros interfaces.

Este ejemplo añade el interface Java **Registration** a un **registration module** IDL.

```
module registration {
   interface Registration {
   };
}
```

Este ejemplo añade el interface Java **Registration** a un **registration module** IDL, e indica que el interface **Registration** desciende del interface **User**.

```
module registration {
   interface Registration: User {
   };
}
```

Métodos Java: Los métodos Java se mapean a operaciones IDL. Las operaciones IDL son similares a los métodos Java excepto en que no hay el concepto de control de acceso. También tenemos que ayudar al compilador IDL especificando qué parámetros son de entrada **in**, de entrada/salida **inout** o de salida **out**, definidos de esta forma:

- in - El parámetro se pasa dentro del método pero no se modifica.
- inout - El parámetro se pasa al método y se podría devolver modificado.
- out - El parámetro se podría devolver modificado.

Este mapeo IDL incluye los métodos de los interfaces **Registration** y **RegistrationHome** a operaciones IDL usando un tipo módulo IDL.

```
module registration {

  interface Registration {
    boolean verifyPassword(in string password);
    string getEmailAddress();
    string getUser();
    long adjustAccount(in double amount);
    double getBalance();
  };

  interface RegistrationHome {
    Registration findByPrimaryKey(
                  in RegistrationPK theuser)
                  raises (FinderException);
  }
}
```

Arrays Java: Los Arrays Java son mapeados a los tipos **array** o **sequence** IDL usando una definición de tipo.

Este ejemplo mapea el array Java **double balances[10]** a un tipo **array** IDL del mismo tamaño.

```
typedef double balances[10];
```

Estos ejemplo mapean el array Java **double balances[10]** a un tipo **sequence** IDL. El primer **typedef sequence** es un ejemplo de secuencia sin límite, y el segundo tiene el mismo tamaño que el array.

```
typedef sequence<double> balances;
typedef sequence<double,10> balances;
```

Excepciones Java: Las excepciones Java son mapeadas a excepciones IDL. Las operaciones usan **exceptions** IDL especificándolas como del tipo **raises**.

Este ejemplo mapea la **CreateException** desde la aplicación subastas al tipo **exception** IDL, y le añade el tipo **raises** a la operación. Las excepciones IDL siguen las sintaxis C++, por eso en lugar de lanzar una excepción (como se haría en lenguaje Java), la operación alcanza (raise) una excepción.

```
exception CreateException {
};

interface RegistrationHome {
  RegistrationPK create(
                  in string theuser,
                  in string password,
                  in string emailaddress,
                  in string creditcard)
                  raises (CreateException);
}
```

Otros Tipos IDL

Estos otros tipos básicos IDL no tienen un equivalente exacto en el lenguaje Java. Muchos de estos deberían sernos familiares si hemos usado C ó C++. El lenguaje Java proporciona mapeo para estos tipos porque los programas escritos en Java pueden recibir datos desde programas escritos en C ó C++.

- **attribute** IDL
- **enum** IDL
- **struct** IDL
- **union** IDL
- **Any** IDL
- **Principal** IDL
- **Object** IDL

atributo IDL: El tipo **attribute** IDL es similiar a los métodos **get** y **set** usados para acceder a los campos en el software de JavaBeans.

En el caso de un valor declarado como un atributo IDL, el compilador IDL genera dos métodos con el mismo nombre que el atributo IDL. Un método devuelve el campo y otro lo selecciona. Por ejemplo, este tipo **attribute**:

```
interface RegistrationPK {
  attribute string theuser;
};
```

define estos métodos:

```
//return user
  String theuser();
//set user
  void theuser(String arg);
```

enum IDL: El lenguaje Java tiene una clase **Enumeration** para representar una colección de datos. El tipo **enum** IDL es diferente porque es declarado como un tipo de dato y no una colección de datos.

El tipo **enum** IDL es una lista de valores que pueden se referenciados por un nombre en vez de por su posición en la lista. En el ejemplo, podemos ver que referirnos al código de estado de un **enum** IDL por un nombre es mucho más legible que hacerlo por su número. Esta línea mapea los valores **static final int** de la clase **final LoginError**. Podemos referirnos a estos valores como lo haríamos con un campo estático:**LoginError.INVALID_USER**.

```
enum LoginError {
  INVALID_USER, WRONG_PASSWORD, TIMEOUT};
```

Aquí hay una versión del tipo **enum** que incluye un subrayado anterior para que pueda ser usado en sentencias **switch**:

```
switch (problem) {
  case LoginError._INVALID_USER:
    System.out.println("please login again");
    break;
}
```

struct IDL: Un tipo **struct** IDL puede ser comparado con una clase Java que sólo tiene campos, que es cómo lo mapea el compilador IDL.

Este ejemplo declara una **struct** IDL. Observamos que los tipos IDL pueden referenciar otros tipos IDL. En este ejemplo **LoginError** viene del tipo **enum** declarado arriba.

```
struct ErrorHandler {
  LoginError errortype;
  short retries;
};
```

union IDL: Una **union** IDL puede representar un tipo de una lista de tipos definidos para esa unión. La **union** mapea a una clase Java del mismo nombre con un método **discriminator** usado para determinar el tipo de esa unión.

Este ejemplo mapea la unión **GlobalErrors** a una clase Java con el nombre **GlobalErrors**. Se podría añadir un case por defecto **case: DEFAULT** para manejar cualquier elemento que podría estar en el tipo **LoginErrors enum**, y no está especificado con una sentencia **case** aquí.

```
  union GlobalErrors switch (LoginErrors) {
     case: INVALID_USER: string message;
     case: WRONG_PASSWORD: long attempts;
     case: TIMEOUT: long timeout;
  };
```

En un programa escrito en lenguaje Java, la clase unión **GlobalErrors** se crea de esta forma:

```
  GlobalErrors ge = new GlobalErrors();
  ge.message("please login again");
```

El valor **INVALID_USER** se recupera de esta forma:

```
  switch (ge.discriminator().value()) {
    case: LoginError._INVALID_USER:
      System.out.println(ge.message);
      break;
  }
```

Tipo Any: si no sabemos que tipo está siento pasado o devuelto desde una operación, podemos usar el tipo **Any**, que representa cualquier tipo IDL. La siguiente operación retorna y pasa un tipo desconocido:

```
interface RegistrationHome {
  Any customSearch(Any searchField, out count);
};
```

Para crear un tipo **Any**, se pide el tipo al "Object Request Broker" (ORB). Para seleccionar un valor de un tipo **Any**, usamos un método **insert_<type>**. Para recuperar un valor, usamos el método **extract_<type>**.

Este ejemplo pide un objeto del tipo **Any**, y usa el método **insert_type** para seleccionar un valor.

```
Any sfield = orb.create_any();
sfield.insert_long(34);
```

El tipo **Any** tiene un valor **TypeCode** asignado que puede consultarse usando **type().kind().value()** sobre el objeto. El siguiente ejemplo muestra una prueba del **TypeCode double**. Este ejemplo incluye una referencia al **TypeCode** IDL encontrado que contiene el objeto **Any**. El **TypeCode** se usa para todos los objetos. Podemos analizar el tipo de un objeto CORBA usando los métodos **_type()** o **type()**.

```
public Any customSearch(Any searchField, IntHolder count){
  if(searchField.type().kind().value() == TCKind._tk_double){
// return number of balances greater than supplied amount
    double findBalance=searchField.extract_double();
```

Principal: El tipo **Principal** identifica al propietario de un objeto CORBA, por ejemplo, un nombre de usuario. El valor puede consultarse desde el campo **request_principal** de la clase **RequestHeader** para hacer la identificación. **Object**: El tipo **Object** es un objeto CORBA. Si necesitamos enviar objetos Java, tenemos que traducirlos a un tipo IDL o usar un mecanismo para serializarlos cuando sean transferidos.

CORBA en la Aplicación de Subasta

El RegistrationBean controlado por contenedor de la aplicación subasta es totalmente reemplazado con un RegistrationServer solitario CORBA que implementa el servicio de registro. El **RegistrationServer** CORBA está construido creando y compilando ficheros de mapeo IDL para que los programas clientes se puedan comunicar con el servidor de registros.

Los ficheros **SellerBean.java** y **AuctionServlet.java** se han actualizado para que busquen el servidor de registro CORBA.

Implementación del RegistrationServer CORBA

Esta sección describe el fichero Registration.idl, que mapea los interfaces remotos **RegistrationHome** y **Registration** desde la aplicación de subastas de JavaBeans de Enterprise a sus equivalentes IDL y muestra como compilar el fichero **Registration.idl** en las clases del servidor de registos CORBA.

El servidor de registros CORBA implementa los métodos **create** y **findByPrimaryKey** desdel el fichero **RegistrationBean.java** original, y lo amplía con los dos métodos siguientes para ilustrar las retrollamadas CORBA, y como usar el tipo **Any**.

- **findLowCreditAccounts(in ReturnResults rr)**, que usa una callback para devolver una lista de cuentas con bajo saldo.
- **any customSearch(in any searchfield, out long count)**, que devuelve un resultado de búsqueda diferente dependiendo del tipo de campo enviado.

Fichero de Mapeos IDL

Aquí está el fichero Registration.idl que mapea los tipos de datos y métodos usados en los programas **RegistrationHome** y **Registration** a sus equivalentes IDL.

```
module registration {

interface Registration {
   boolean verifyPassword(in string password);
   string getEmailAddress();
   string getUser();
   long adjustAccount(in double amount);
   double getBalance();
};

interface RegistrationPK {
   attribute string theuser;
};

enum LoginError {INVALIDUSER, WRONGPASSWORD, TIMEOUT};

exception CreateException {
};
```

```
exception FinderException {
};

typedef sequence<Registration> IDLArrayList;

interface ReturnResults  {
  void updateResults(in IDLArrayList results)
        raises (FinderException);
};

interface RegistrationHome {
  RegistrationPK create(in string theuser,
                        in string password,
                        in string emailaddress,
                        in string creditcard)
                           raises (CreateException);

  Registration findByPrimaryKey(
                in RegistrationPK theuser)
                raises (FinderException);
  void findLowCreditAccounts(in ReturnResults rr)
                raises (FinderException);
  any customSearch(in any searchfield, out long count);
};
};
```

Compilar el Fichero de Mapeos IDL

El fichero IDL tiene que ser convertido en clases Java que puedan ser usadas en una red distribuida CORBA. La plataforma Java 2 compila los ficheros **.idl** usando el programa **idltojava**. Este programa será reemplazado eventualmente con el comando **idlj**.

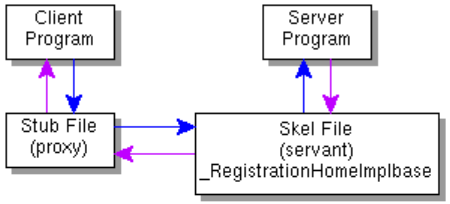
Los argumentos **-fno-cpp** indican que no hay compilador C++ instalado.

```
idltojava -fno-cpp Registration.idl
```

Otros compiladores Java IDL también deberían funcionar, por ejemplo, **jidl** de ORBacus puede generar clases que pueden ser usadas por el ORB de Java 2.

Stubs y Skeletons

Corba y RMI son similares en que la compilación genera un fichero stub para el cliente y un fichero skeleton para el servidor. El stub (o proxy), y el skeleton (o sirviente) se usan para envolver o desenvolver datos entre el cliente y el servidor. El skeleton (o sirviente) está implementado mediante el servidor. En este ejemplo, el interface **RegistrationHome** genera una clase **_RegistrationHomeImplBase** (la clase skeleton o sirviente) que extiende la clase **RegistrationServer** generada.


Client
Program
Server
Program
Stub File
(proxy)
Skel File
(servant)
_RegistrationHomeImplbase

Cuando se solicita un objeto CORBA remoto o se llama a un método remoto, la llamada del cliente pasa a través de la clase stub antes de alcanzar el servidor. Este clase proxy invoca la petición CORBA para el programa cliente. El siguiente ejemplo es el código generado automáticamente por la clase **RegistrationHomeStub.java**.

```
org.omg.CORBA.Request r = _request("create");
r.set_return_type(
      registration.RegistrationPKHelper.type());
org.omg.CORBA.Any _theuser = r.add_in_arg();
```

Object Request Broker

El centro de una red distribuida CORBA es el "Object Request Broker" o ORB. El ORB se encarga de empaquetar y desempaquetar los objetos entre el cliente y el servidor. Otros servicios como Servicios de Nombres y Servicios de Eventos funcionan con el ORB.

La plataforma Java 2 incluye un ORB en la distribución llamado el IDL ORB. Este ORB es diferente de otros muchos ORBs porque no incluye un distintivo de "Basic Object Adapter" (BOA) o "Portable Object Adapter" (POA).

Una adaptador de objetos maneja la creacción y ciclo de vida de los objetos en un espacio distribuido CORBA. Esto puede ser comparado con el contenedor del servidor de JavaBeans Enterprise que maneja el ciclo de vida de los beans de entidad y de sesión.

Los programas AuctionServlet y SellerBean crean e inicializan un ORB de Java 2 de esta forma:

```
ORB orb = ORB.init(args, null);
```

En el programa RegistrationServer, el objeto servidor puede ser distribuido en unión con el ORB usando el método **connect**:

```
RegistrationServer rs = new RegistrationServer();
orb.connect(rs);
```

Un objeto conectado a un ORB puede ser eleminado con el método **disconnect**:

```
orb.disconnect(rs);
```

Una vez conectado a un objeto servidor CORBA, el ORB Java2 mantiene vivo el servidor y espera peticiones del cliente para el servidor CORBA.

```
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
  sync.wait();
}
```

Poner Disponible el Servidor CORBA

Aunque este objeto está ahora siendo mapeado por el ORB, los clientes todavía no tienen el mecanismo para encontrar el objeto remoto. Esto puede resolverse uniendo el objeto servidor CORBA a un servicio de nombres.

El servicio de nombres Java 2 llamado **tnameserv**, por defecto usa el puerto 900; sin embargo, este valor puede modificarse seleccionado el argumento **-ORBInitialPort portnumber** cuando se arranca **tnameserv** o seleccionando la propiedad **org.omg.CORBA.ORBInitialPort** cuando arrancamos los procesos cliente y servidor.

Las siguientes secciones describen el método **main** de la clase **RegistrationServer**.

```
java.util.Properties props=System.getProperties();
props.put("org.omg.CORBA.ORBInitialPort", "1050");
System.setProperties(props);
ORB orb = ORB.init(args, props);
```

Las siguientes líneas muestran que la referencia inicial de nombres es inicializada por la petición del servicio llamado **NameService**. El **NamingContext** es recuperado y el nombre construido y unido al servicio de nombres como elementos **NameComponent**. El nombre de este ejemplo tiene una raíz llamada **auction** que es este objeto que se está uniendo como **RegistrationBean** desde la raíz **auction**. El nombre podría ser comparado por una clase mediante el nombre de **auction.RegistrationBean**.

```
org.omg.CORBA.Object nameServiceObj =
      orb.resolve_initial_references("NameService") ;
NamingContext nctx =
      NamingContextHelper.narrow(nameServiceObj);
NameComponent[] fullname = new NameComponent[2];
fullname[0] = new NameComponent("auction", "");
fullname[1] = new NameComponent(
                    "RegistrationBean", "");

NameComponent[] tempComponent = new NameComponent[1];
```

```
for(int i=0; i < fullname.length-1; i++ ) {
    tempComponent[0]= fullname[i];
    try{
     nctx=nctx.bind_new_context(tempComponent);
    }catch (Exception e){
      System.out.println("bind new"+e);}
}
tempComponent[0]=fullname[fullname.length-1];
try{
 nctx.rebind(tempComponent, rs);
}catch (Exception e){
 System.out.println("rebind"+e);
}
```

Conectar un nuevo ORB

El ORB IDL de Java 2 realmente no incluye ninguno de los servicios disponibles en muchos otros ORBs comerciales como los servicios de seguridad o eventos (notificación). Podemos usar otro ORB en el runtime de Java 2 configurando dos propiedades e incluyendo cualquier codigo de objeto adaptador que sea necesario.

Usar un nuevo ORB en el servidor de registros requiere que las propiedades **org.omg.CORBA.ORBClass** y **org.omg.CORBA.ORBSingletonClass** apunten a las clases ORB apropiadas. En este ejemplo se usa el ORB ORBacus en lugar del ORB IDL de Java 2. Para usar otro ORB, el código de abajo debería conectarse dentro del método **RegistrationServer.main**.

En el código de ejemplo, se usa un ORB **SingletonClass**. Este ORB no es un ORB completo, y su uso primario es como factoría para **TypeCodes**. La llamada a **ORB.init()** en la última línea crea el ORB Singleton.

```
Properties props= System.getProperties();
props.put("org.omg.CORBA.ORBClass",
          "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass",
      "com.ooc.CORBA.ORBSingleton");
System.setProperties(props);
ORB orb = ORB.init(args, props) ;
```

En el IDL de Java 2, no hay un objeto adaptador distinto. Como se muestra en el segmento de código inferior, usar el "Basic Object Adapter" desde ORBacus requiere un conversión explícita al ORB ORBacus, El "Broker Object Architecture" (BOA) es notificado de que el objeto ya está distribuido llamando al método **impl_is_ready(null)**.

```
BOA boa = ((com.ooc.CORBA.ORB)orb).BOA_init(
                                    args, props);
...
boa.impl_is_ready(null);
```

Aunque los dos ORBs **ORBSingletonClass** y **ORBClass** construyen el nombre del objeto usando **NameComponent**, tenemos que usar un diferente servicio de nombres ORBacus. El servicio **CosNaming.Server** se arranca de la siguiente forma, donde el parámetro **-OAhost** es opcional:

```
java com.ooc.CosNaming.Server -OAhost localhost -OAport 1060
```

Una vez arrancado el servicio de nombres, los programas cliente y servidor pueden encontrar el servicio de nombres usando el protocolo IIOP hacia el host y el puerto nombrados cuando se arrancó el servicio de nombrado:

```
java registration.RegistrationServer
      -ORBservice NameService
      iiop://localhost:1060/DefaultNamingContext
```

Acceso al Servicio de Nombres por los Clientes CORBA

Los cliente CORBA acceden al servicio de nombres de una forma similar a como lo hace el servidor, excepto que en lugar de unir un nombre, el cliente resuelve el nombre construido desde el **NameComponents**.

Las clases **AuctionServlet** y **SellerBean** usan el siguiente código para buscar el servidor CORBA:

```
NameComponent[] fullname = new NameComponent[2];
fullname[0] = new NameComponent("auction", "");
fullname[1] = new NameComponent(
                    "RegistrationBean", "");

RegistrationHome regRef =
```

```
        RegistrationHomeHelper.narrow(
                        nctx.resolve(fullname));
```

En el caso del ORB ORBacus, los clientes también necesitan un "Basic Object Adapter" si se usan retrollamadas en el método **SellerBean.auditAccounts**. El contexto de nombrado también se configura de forma diferente para el servidor ORBacus arrancado anteriormente:

```
Object obj =
      ((com.ooc.CORBA.ORB)orb).get_inet_object (
              "localhost",
              1060,
              "DefaultNamingContext");
NamingContext nctx = NamingContextHelper.narrow(obj);
```

Clases Helper y Holder

Las referencias a objetos remotos en CORBA usan una clase **Helper** para recuperar un valor desde ese objeto. Un método usado comunmente es el método **Helper**, que asegura que el objeto está encastado correctamente.

Las clases **Holder** contienen valores devueltos cuando se usan parámetros **inout** o **out** en un método. El llamador primero ejemplariza la clase **Holder** apropiada para ese tipo y recupera el valor desde la clase cuando la llamada retorna. En el siguiente ejemplo, el valor del contador **customSearch** se configura y recupera después de que se haya llamado a **customSearch**. En el lado del servidor el valor del contador se selecciona llamando a **count.value=newvalue**.

```
IntHolder count= new IntHolder();
sfield=regRef.customSearch(sfield,count);
System.out.println("count now set to "+count.value);
```

Recolección de Basura

Al contrario que RMI, CORBA no tiene un mecanismo de recolección de basura distribuido. Las referencias a un objeto son locales al proxy del cliente y al sirviente del servidor. Esto significa que cada Máquina Virtual Java[1] (JVM) es libre de reclamar un objeto y recoger la basura si no tiene más referencias sobre él. Si un objeto no es necesario en el servidor, necesitamos llamar a **orb.disconnect(object)** para permitir que el objeto sea recolectado.

Retrollamadas (Callbacks) CORBA

El nuevo método **findLowCreditAccounts** es llamado desde el **AuctionServlet** usando la URL **http://localhost:7001/AuctionServlet?action=auditAccounts**.

El método **AuctionServlet.auditAccounts** llama al método **SellerBean.auditAccounts**, que devuelve un **ArrayList** de registros de Registration.

```
//AuctionServlet.java
  private void auditAccounts(ServletOutputStream out,
    HttpServletRequest request) throws IOException{

//   ...

   SellerHome home = (SellerHome) ctx.lookup("seller");
   Seller si= home.create();

   if(si != null) {
     ArrayList ar=si.auditAccounts();
     for(Iterator i=ar.iterator(); i.hasNext();) {
       Registration user=(Registration)(i.next());
       addLine("<TD>"+user.getUser() +
         "<TD><TD>"+user.getBalance()+
         "<TD><TR>", out);
     }
          addLine("<TABLE>", out);
    }
```

El objeto **SellerBean** llama al método CORBA **RegistrationHome.findLowCreditAccounts** implementado en el fichero **RegistrationServer.java**, y se pasa una referencia a sí mismo. La referencia es pasada siempre que la clase **SellerBean** implemente el interface **ReturnResults** declarado en el **Registration.idl**.

```
//SellerBean.java
  public ArrayList auditAccounts() {
    try{
      NameComponent[] fullname = new NameComponent[2];
      fullname[0] = new NameComponent("auction", "");
```

```
        fullname[1] = new NameComponent(
                            "RegistrationBean", "");

        RegistrationHome regRef =
            RegistrationHomeHelper.narrow(
                                nctx.resolve(fullname));
          regRef.findLowCreditAccounts(this);
          synchronized(ready) {
            try{
             ready.wait();
            }catch (InterruptedException e){}
          }
          return (returned);
        }catch (Exception e) {
         System.out.println("error in auditAccounts "+e);
        }
        return null;
    }
```

El método **RegistrationServer.findLowCreditAccounts** recupera los registros de usuario desde la tabla Registration de la base de datos que tengan un valor de crédito menor de tres. Entonces devuelve la lista de registros Registration en un **ArrayList** llamando al método **SellerBean.updateResults** que tiene una referencia a ella.

```
//RegistrationServer.java
  public void findLowCreditAccounts(
                final ReturnResults client)
                throws Finder Exception {
    Runnable bgthread = new Runnable() {
      public void run() {
        Connection con = null;
        ResultSet rs = null;
        PreparedStatement ps = null;
        ArrayList ar = new ArrayList();

        try{
          con=getConnection();
          ps=con.prepareStatement(
                "select theuser,
                balance from registration
                where balance < ?");
          ps.setDouble(1, 3.00);
          ps.executeQuery();
          rs = ps.getResultSet();
          RegistrationImpl reg=null;
          while (rs.next()) {
            try{
             reg= new RegistrationImpl();
            }catch (Exception e) {
             System.out.println("creating reg"+e);
            }
            reg.theuser = rs.getString(1);
            reg.balance = rs.getDouble(2);
            ar.add(reg);
          }
          rs.close();

          RegistrationImpl[] regarray =
                (RegistrationImpl [])ar.toArray(
                new RegistrationImpl[0]);
          client.updateResults(regarray);
        }catch (Exception e) {
         System.out.println(
                    "findLowCreditAccounts: "+e);
          return;
        }
        finally {
```

```
        try{
            if(rs != null) {
              rs.close();
            }
            if(ps != null) {
             ps.close();
            }
            if(con != null) {
              con.close();
            }
          }catch (Exception ignore) {}
        }
      }//run
    };
  Thread t = new Thread(bgthread);
  t.start();
  }
```

El método **SellerBean.updateResults** actualiza el **ArrayList** global de registros de Registration devuelto por el objeto **RegistrationServer** y notifica al método **SellerBean/auditAccounts** que puede devolver este **ArrayList** de registros Registration al **AuctionServlet**.

```
  public void updateResults(Registration[] ar)
        throws registration.FinderException {
    if(ar == null) {
          throw new registration.FinderException();
    }
    try{
      for(int i=0; i< ar.length; i++) {
        returned.add(ar[i]);
      }
    }catch (Exception e) {
     System.out.println("updateResults="+e);
     throw new registration.FinderException();
    }
     synchronized(ready) {
     ready.notifyAll();
    }
  }
```

Usar el Tipo Any

El método **RegistrationServer.customSearch** usa el tipo **Any** de IDL para pasar y devolver resultados. El **customSearch** es llamado por el **AuctionServlet** de esta forma:

```
  http://localhost.eng.sun.com:7001/
     AuctionServlet?action=customSearch&searchfield=2
```

El parámetro **searchfield** puede ser seleccionado como un número o un string. El método **AuctionServlet.customFind** pasa el campo de búsqueda directamente al método **SellerBean.customFind** que recupera un **String** que luego es mostrado al usuario:

```
  private void customSearch(ServletOutputStream out,
               HttpServletRequest request)
               throws IOException{

    String text = "Custom Search";
    String searchField=request.getParameter(
                                "searchfield");

    setTitle(out, "Custom Search");
    if(searchField == null ) {
      addLine("Error: SearchField was empty", out);
      out.flush();
      return;
    }
    try{
      addLine("<BR>"+text, out);
```

```
      SellerHome home = (SellerHome)
                          ctx.lookup("seller");
      Seller si= home.create();
      if(si != null) {
        String displayMessage=si.customFind(
                                    searchField);
        if(displayMessage != null ) {
          addLine(displayMessage+"<BR>", out);
        }
      }
     }catch (Exception e) {
      addLine("AuctionServlet customFind error",out);
      System.out.println("AuctionServlet " +
                       "<customFind>:"+e);
     }
     out.flush();
   }
```

El método **SellerBean.customFind** llama al objeto **RegistrationHome** implementado en la clase **RegistrationServer.java**, y dependiendo de si el **searchField** puede ser convertido a un double o a un string, inserta este valor dentro de un objeto del tipo **Any**. El objeto **Any** se crea mediante una llamada al ORB, **orb.create_any();**

El método **customFind** también usa un parámetro **out**, **count**, del tipo **int** que devuelve el número de registros encontrados. El valor de **count** se recupera usando **count.value** cuando la llamada retorna:

```
//SellerBean.java
  public String customFind(String searchField)
           throws javax.ejb.FinderException,
           RemoteException{

  int total=-1;
  IntHolder count= new IntHolder();

  try{
      NameComponent[] fullname = new NameComponent[2];
      fullname[0] = new NameComponent("auction", "");
      fullname[1] = new NameComponent(
                          "RegistrationBean", "");

      RegistrationHome regRef =
        RegistrationHomeHelper.narrow(
                            nctx.resolve(fullname));
      if(regRef == null ) {
        System.out.println(
                     "cannot contact RegistrationHome");
        throw new javax.ejb.FinderException();
      }
      Any sfield=orb.create_any();
      Double balance;
      try{
        balance=Double.valueOf(searchField);
        try {
            sfield.insert_double(balance.doubleValue());
        }catch (Exception e) {
         return("Problem with search value"+balance);
        }
       sfield=regRef.customSearch(sfield,count);
       if(sfield != null ) {
         total=sfield.extract_long();
       }
       return(total+"
        accounts are below optimal level from" +
        count.value+" records");
      }catch (NumberFormatException e) {
       sfield.insert_string(searchField);
       Registration reg;
```

```
        if((reg=RegistrationHelper.extract(
                  regRef.customSearch(
                           sfield,count)))
                           != null ) {
          return("Found user "+reg.getUser() +"
                  who has email address "+
                  reg.getEmailAddress());
        }else {
         return("No users found who have email address " +
                  searchField);
        }
       }catch(Exception e){
           System.out.println("customFind problem="+e);
           throw new javax.ejb.FinderException();
       }
     }
```

El valor devuelto desde la llamada a **customFind** se extrae dentro de un objeto del tipo **Any** y se construye un **String** con la salida mostrada al usuario. Para los tipos sencillos se puede usar el método **extract_<type>** de **Any**. Sin embargo, para el tipo **Registration**, se usa la clase **RegistrationHelper**.

```
Registration reg =
  RegistrationHelper.extract(
              regRef.customSearch(sfield,count))
```

El método **RegistrationServer.customSearch** determina el tipo del objeto que está siendo pasado en el parámetro **searchField** chequeando el **.type().kind().value()** del objeto **Any**.

```
if(searchField.type().kind().value() ==
                         TCKind._tk_double)
```

Finalmente, como el método **customSearch** devuelve un objeto del tipo **Any**, se requiere una llamada a **orb.create_any()**. Para tipos sencillos como **double**, se usa el método **insert_<type>**. Para el tipo Registration , se usa la clase **RegistrationHelper**: **RegistrationHelper.insert(returnResults, regarray[0])**.

```
//RegistrationServer.java
  public Any customSearch(Any searchField,
                          IntHolder count){
    Any returnResults= orb.create_any();

    int tmpcount=count.value;
    if(searchField.type().kind().value() ==
                          TCKind._tk_double){
// return number of balances greater
// than supplied amount
      double findBalance=searchField.extract_double();
      Connection con = null;
      ResultSet rs = null;
      PreparedStatement ps = null;
      try{
        con=getConnection();
        ps=con.prepareStatement("select count(*) from
                    registration where balance < ?");
        ps.setDouble(1, findBalance);
        ps.executeQuery();
        rs = ps.getResultSet();
        if(rs.next()) {
          tmpcount = rs.getInt(1);
        }
        count.value=tmpcount;
        rs.close();
       }catch (Exception e) {
                 System.out.println("custom search: "+e);
                 returnResults.insert_long(-1);
                 return(returnResults);
       }
       finally {
```

```
            try{
              if(rs != null) { rs.close(); }
              if(ps != null) { ps.close(); }
              if(con != null) { con.close(); }
            } catch (Exception ignore) {}
          }
            returnResults.insert_long(tmpcount);
            return(returnResults);
        }else if(searchField.type().kind().value() ==
                  TCKind._tk_string) {
         // return email addresses that match supplied address
         String findEmail=searchField.extract_string();
         Connection con = null;
         ResultSet rs = null;
         PreparedStatement ps = null;
         ArrayList ar = new ArrayList();
         RegistrationImpl reg=null;
         try{
           con=getConnection();
           ps=con.prepareStatement("select theuser,
              emailaddress from registration
              where emailaddress like ?");
           ps.setString(1, findEmail);
           ps.executeQuery();
           rs = ps.getResultSet();
           while (rs.next()) {
             reg= new RegistrationImpl();
             reg.theuser = rs.getString(1);
             reg.emailaddress = rs.getString(2);
             ar.add(reg);
           }
           rs.close();

           RegistrationImpl[] regarray =
                (RegistrationImpl [])ar.toArray(
                new RegistrationImpl[0]);
             RegistrationHelper.insert(
                                 returnResults,
                                 regarray[0]);
             return(returnResults);
          }catch (Exception e) {
           System.out.println("custom search: "+e);
           return(returnResults);
          }
          finally {
           try{
             if(rs != null) { rs.close(); }
             if(ps != null) { ps.close(); }
             if(con != null) { con.close(); }
           } catch (Exception ignore) {}
         }
       }
       return(returnResults);
     }
```

Conclusión

Como hemos podido ver, convertir una aplicación para que use RMI o CORBA requiere muy pocos cambios en el corazón del programa. La principal diferencia ha sido la inicialización y el servicio de nombres. Mediante la abstracción de estas dos áreas en nuestra aplicación fuera de la lógica del negocio podemos migrar fácilmente entre diferentes arquitecturas de objetos distribuidos.

_____

[1] Cuando se usan en toda esta site, los términos, "Java virtual machine" o "JVM" significa una máquina virtual de la plataforma Java.

Ozito

JDBC

La aplicación de subasta con JavaBeans Enterpise y con sus dos variantes de "Remote Method Invocation" (RMI) y "Common Object Request Broker" (CORBA) han usado llamadas sencillas de JDBC JDBC™ para actualizar y consultar información desde una base de datps usando una conexión JDBC. Por defecto, el acceso a bases de datos JDBC implica abrir una conexión con la base de datos, ejecutar comandos SQL en un sentencia, procesar los datos devueltos y cerrar la conexión con la base de datos.

En conjunto, la aproximación por defecto funciona bien para bajos volúmenes de acceso a la base de datos, pero ¿cómo podemos manejar un gran número de peticiones que actualizan muchas tablas relacionadas a la vez y aún así asegurar la integridad de los datos? Esta sección explica cómo hacerlo con los siguientes tópicos:

Drivers JDBC

La conexión con la base de datos está manejada por la clase Driver JDBC. El SDK de Java contiene sólo un driver JDBC, un puente **jdbc-odbc** que comunica con un driver "Open DataBase Conectivity" (ODBC) existente. Otras bases de datos necesitan un driver JDBC espécifico para esa base de datos.

Para obtener un idea general de lo que hacer un driver JDBC, podemos examinar el fichero **JDCConnectionDriver.java**. La clase JDCConnectionDriver implemta la clase **java.sql.Driver** y actúa como un driver "pass-through" re-enviando peticiones JDBC al driver JDBC real de la base de datos. La clase driver JDBC se

carga con un llamada a **Class.forName(drivername)**.

Estas líneas de código muestran cómo cargar tres clases diferentes de drivers JDBC:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("postgresql.Driver");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Cada driver JDBC está configurado para entender una URL específica, por eso se pueden cargar varios drivers JDBC a la vez. Cuando especificamos una URL en el momento de la conexión, se selecciona el primer driver JDBC que corresponda.

El puente jdbc-odbc acepta URLS que empiecen con **jdbc:odbc:** y usa el siguiente campo de esa URL para especificar el nombre de la fuente de los datos. Este nombre identifica el esquema de la base de datos particular a la que queremos acceder. La URL también puede incluir más detalles sobre cómo contactyar con la base de datos e introducir la cuenta.

```
//access the ejbdemo tables
  String url = "jdbc:odbc:ejbdemo";
```

El siguiente ejemplo contiene la información de Oracle SQL*net sobre una base de datos particular llamada ejbdemo en la máquina dbmachine:

```
String url = "jdbc:oracle:thin:user/password@(
      description=(address_list=(
                    address=(protocol=tcp)
      (host=dbmachine)(port=1521)))(source_route=yes)
      (connect_data=(sid=ejbdemo)))";
```

Este siguiente ejemplo usa **mysql** para conectar con la base de datos ejbdemo en la máquina local. También se incluyen los detalles del nombre de usuario y la password para el login.

```
String url =
 "jdbc:mysql://localhost/ejbdemo?user=user;
    password=pass";
```

Los drivers JDBC se dividen en cuatro tipos. También se pueden categorizar como puro java o drivers pequeños para indicar si son usados por aplicaciones clientes (drivers puro java) o por applets (drivers pequeños).

Drivers del Tipo 1

Los drivers JDBC del tipo 1 son drivers puente como el puente jdbc.odbc. Estos drivers utilizan un intermediario como el ODBC para transferir las llamadas SQL a la base de datos. Los drivers puente cuentan con código nativo, aunque la librería

de código nativo del puente jdbc-odbc forma parte de la Máquina Virtual Java 2[1].

Drivers del Tipo 2

Los drivers del tipo 2 usan el API existente de la base de datos para comunicarla con el cliente. Aunque los drivers del tipo 2 son más rápidos que los del tipo 1, los del tipo 2 usan código nativo y requieren permisos adicionales para funcionar en un applet.

Un driver del tipo 2 podría necesitar código de base de datos en el lado del cliente para conectar a través de la red.

Drivers del Tipo 3

Los Drivers del tipo 3 llaman al API de la base de datos en el servidor. Las peticiones JDBC desde el cliente son primero comprobadas por el Driver JDBC en el servidor para ejecutarse. Los drivers del tipo 3 y 4 pueden usarse en clientes applets ya que no necesitan código nativo.

Driveres del Tipo 4

El nivel más alto de drivers reimplementa el API de red para base de datos en el lenguaje Java. Los Drivers del tipo 4 también pueden usarse en clientes applets porque no necesitan código nativo.

Conexiones a Bases de Datos

Una conexión con una base de datso puede establecerese con un llamada al método **DriverManager.getConnection**. La llamada toma una URL que identifica la base de datos, y opcionalmente el nombre de usuario y la password para la base de datos.

```
Connection con = DriverManager.getConnection(url);
Connection con = DriverManager.getConnection(url,
                     "user", "password");
```

Después de establecer la conexión, se puede ejecutar una sentencia contra la base de datos. Los resultados de la sentencias pueden recuperarse y cerrarse la conexión.

Una características útil de la clase **DriverManager** es el método **setLogStream**. Podemos usar este método para generar información de seguimiento para ayudarnos a dignosticar problemas de conexión que normalmente no serían visibles. Para generar la información de seguimiento, sólo tenemos que llamar al método de esta forma:

```
DriverManager.setLogStream(System.out);
```

La sección Connection Pooling en el capítulo 8 muestra cómo podemos mejorar las conexión JDBC sin cerrrar la conexión una vez completada la sentencia. Cada conexión JDBC a una base de datos provoca una sobrecarga al abrir un nuevo socket y usar el nombre de usuario y la password para login en la base de datos. La reutilización de las conexiones reduce la sobrecarga. Las colas de Conexiones mantienen una lista de conexiones abiertas y limpia cualquier conexión que no pueda ser reutilizada.

Sentencias

Hay tres tipos básicos de sentencias SQL usadas en el API JDBC: **CallabelStatement**, **Statement**, y **PreparedStatement**. Cuando se envía una sentencias **Statement** o **PreparedStatement** a la base de datos, el driver la traduce a un formato que la base de datos pueda reconocer.

Sentencias Callable

Una vez que hemos establecido una conexión con una base de datos, podemos usar el método **Connection.prepareCall** para crear una sentencia callable. Estas sentencias nos permite ejecutar prodecimientos almacenados SQL.

El siguiente ejemplo crea un objeto **CallableStatement** con tres parámetros para almacenar información de la cuenta de login:

```
CallableStatement cs =
      con.prepareCall("{call accountlogin(?,?,?)}");
cs.setString(1,theuser);
cs.setString(2,password);
cs.registerOutParameter(3,Types.DATE);

cs.executeQuery();
Date lastLogin = cs.getDate(3);
```

Statements

El interface **Statement** nos permite ejecutar una simple sentencias SQL sin parámetros. Las instrucciones SQL son insertadas dentro del objeto **Statement** cuando se llama al método **Statement.executeXXX** method.

Sentencias Query: Este segmento de código crea un objeto **Statement** y llama al método **Statement.executeQuery** para seleccionar texto desde la base de datos **dba**. El resultado de la consulta se devuelve en un objeto **ResultSet**. Cómo recuperar los resultados desde este objeto **ResultSet** se explica más abajo en Hoja de Resultados.

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

```
ResultSet results = stmt.executeQuery(
                        "SELECT TEXT FROM dba ");
```

Sentencias Update: Este segmento de código crea un objeto **Statement** y llama al método **Statement.executeUpdate** para añadir una dirección de email a una tabla de la base de datos **dba**:

```
String updateString =
          "INSERT INTO dba VALUES (some text)";
int count = stmt.executeUpdate(updateString);
```

Setencias Prepared

El interface **PreparedStatement** desciende del interface **Statement** y usa una plantilla para crear peticiones SQL. Se usa una **PreparedStatement** para enviar sentencias SQL precompiladas con uno o más parámetros.

Query PreparedStatement: Creamos un objeto **PreparedStatement** especificando la definición de plantilla y la situación de los parámetros. Los datos de los parámetros se insertan dentro del objeto **PreparedStatement** llamando a sus métodos **setXXX** y especificando el parámetro y su dato. Las instrucciones SQL y los parámetros son enviados a la base de datos cuando se llama al método **executeXXX**.

Este segmento de código crea un objeto **PreparedStatement** para seleccionar datos de usuarios basados en la dirección email del usuario. El interrogante ("?") indica que este sentencia tiene un parámetro:

```
PreparedStatement pstmt = con.prepareStatement("
                            select theuser from
                            registration where
                            emailaddress like ?");
//Initialize first parameter with email address
  pstmt.setString(1, emailAddress);
  ResultSet results = ps.executeQuery();
```

Una vez que se ha inicializado la plantilla **PreparedStatement** sólo se insertan los valores modificados para cada llamadas:

```
pstmt.setString(1, anotherEmailAddress);
```

Nota: No todos los drivers de bases de datos compilan sentencias preparadas.

Update PreparedStatement: Este segmento de código crea un objeto **PreparedStatement** para actualizar el registro de un vendedor. La plantilla tiene

cinco parámetros, que se seleccionan con cinco llamadas a los métodos **PreparedStatement.setXXX** apropiados.

```
PreparedStatement ps = con.prepareStatement(
        "insert into registration(theuser, password,
           emailaddress, creditcard,
           balance) values (
           ?, ?, ?, ?, ?)");
  ps.setString(1, theuser);
  ps.setString(2, password);
  ps.setString(3, emailaddress);
  ps.setString(4, creditcard);
  ps.setDouble(5, balance);
  ps.executeUpdate();
```

Cachear los Resultados de la Base de Datos

El concepto **PreparedStatement** de reutilizar peticiones puede extenderse al cacheo de resultados de una llamada JDBC. Por ejemplo, una descripción de un ítem de la subastas permanece igual hasta que el vendedor lo cambia. Si el ítem recibe cientos de peticiones, el resultado de la sentencia: **query "select description from auctionitems where item_id='4000343'"** podría ser almacenado de forma más eficiente en un tabla hash.

Almacenar resultados en una tbal hash requiere que la llamada JDBC sea interceptada antes de crear una sentencia real que devuelva los resultados cacheados, y la entrada del caché debe limpiarse si hay una actualización correspondiente con ese **item_id**.

Hoja de Resultados

El interface **ResultSet** maneja accesos a datos devueltos por una consulta. Los datos devueltos son igual a una línea de la base de la tabla de la base de datos. Algunas consultas devuelven una línea, mientras que muchas consultas devuelven múltiples líneas de datos.

Se utilizan los métodos **getType** para recuperar datos desde columnas específicas para cada fila devuelta en la consulta. Este ejemplo recupera la columna **TEXT** de todas las tablas con una columna **TEXT** en la base de datos**dba**. El método **results.next** mueve hasta la siguiente fila recuperada hasta que se hayan procesado todas las filas devueltas:

```
  Statement stmt = con.createStatement();
  ResultSet results = stmt.executeQuery(
                        "SELECT TEXT FROM dba ");
  while(results.next()){
```

```
      String s = results.getString("TEXT");
      displayText.append(s + "\n");
    }
    stmt.close();
```

Hoja de Resultados Scrollable

Antes del JDBC 2.0, los dirvers JDBC devolvían hojas de resultado de sólo lectura con cursores que sólo se movían en una dirección, hacia adelante. Cada elemento era recuperado mediante una llamada al método **next** de la hoja de resultados.

JDBC 2.0 introduce las hojas de resultados scrollables cuyos valores pueden ser leídos y actualizados si así lo permite la base de datos original. Con las hojas de resultados scrollables, cualquier fila puede ser seleccionada de forma aleatorio, y nos podemos mover por la hoja de resultados hacia adelante y hacia atrás.

Una ventaja de la nueva hoja de resultados es que podemos actualizar un conjunto de filas correspondientes son tener que enviar una llamada adicional a **executeUpdate**. Las actualizaciones se hacen llamando a JDBC y no se necesitan comandos SQL personalinzados. Esto aumenta la portabilidad del código de la base de datos que creemos.

Tanto **Statements** como **PreparedStatements** tienen un constructor adicional que acepta un parámetro tipo scroll y otro tipo update. El valor del tipo scroll puede ser uno de los siguientes valores:

- **ResultSet.TYPE_FORWARD_ONLY**
  Comportamiento por defecto en JDBC 1.0, la aplicación sólo puede llamar a **next()** sobre la hoja de resultados.
- **ResultSet.SCROLL_SENSITIVE**
  La hoja de resultados es totalmente navegable y las actualizaciones son reflejadas en la hoja de resultados cuando ocurren.
- **ResultSet.SCROLL_INSENSITIVE**
  La hoja de resultados es totalmente navegable pero las actualizaciones son sólo visibles cuando se cierra la hoja de resultados. Necesitamos crear una nueva hoja de resultados para verlos.

El parámetro del tipo update puede ser uno de estos dos valores:

- **ResultSet.CONCUR_READ_ONLY**
  La hoja de resultados es de sólo lectura.
- **ResultSet.CONCUR_UPDATABLE**
  La hoja de resultados puede ser actualizada.

Podemos verificar que nuestra base de datos soporta estos tipos llamando al método **con.getMetaData().supportsResultSetConcurrency()** como se ve aquí:

```
    Connection con = getConnection();
```

```
if(con.getMetaData().supportsResultSetConcurrency(
   ResultSet.SCROLL_INSENSITIVE,
   ResultSet.CONCUR_UPDATABLE)) {

   PreparedStatement pstmt = con.prepareStatement(
     "select password, emailaddress,
     creditcard, balance from
     registration where theuser = ?",
     ResultSet.SCROLL_INSENSITIVE,
     ResultSet.CONCUR_UPDATABLE);
}
```

Navegar por la Hoja de Resultados

La hoja de resultados totalmente scrollable devuelve un cursor que puede moverse usando comandos sencillos. Por defecto el cursor de la hoja de resultados apunta a la fila antes de la primera fila en la hoja de resultados. Una llamada a **next()** recupera la primera fila de la hoja de resultados. el cursor puede tambien moverse llamando a uno de los siguientes métodos de **ResultSet**:

- **beforeFirst()**: Posición por defecto. Pone el cursor antes de la primera fila de la hoja de resultados.
- **first()**: Pone el cursor en la primera fila de la hoja de resultados.
- **last()**: Pone el cursor antes de la última fila de la hoja de resultados.
- **afterLast()** Pone el cursor más allá de la última fila de la hoja de resultados. Se llama a **previous** para movernos hacia atrás en la hoja de resultados.
- **absolute(pos)**: Pone el cursor en el número de fila indicado donde absolute(1) es la primera fila y absolute(-1) es la última fila.
- **relative(pos)**: Pone el cursor en una línea relativa a la posición actual donde relative(1) mueve el cursor una fila hacia adelante.

Actualizar la Hoja de Resultados

Podemos actualizar un valor en la hoja de resultados llamando al método **ResultSet.update<type>** sobre la fula donde está posicionado el cursor. El valor del tipo aquí es el midmo usando cuando se recupera un valor de una hoja de resultados, por ejemplo, **updateString** actualiza un valor String en la hoja de resultados.

El siguiente código actualiza el balance de un usuario desde la hoja de resultados creada anteriormente. La actualización sólo se aplica a la hoja de resultados hasta que se llama a **rs.updateRow()**, que actualiza la base de datos original. Cerrando la hoja de resultados antes de llamar a **updateRow** se perderá cualquier edición aplicada en la hoja de resultados.

```
rs.first();
```

```
updateDouble("balance",
      rs.getDouble("balance") - 5.00);
```

Insertar una nueva fila usa los mismos métodos **update<type>**. La única diferencia es que se llama al método **rs.moveToInsertRow** de que los datos hayan sido inicializados y después se llama a **rs.insertRow()**. Podemos borrar la fila actual con una llamada a **rs.deleteRow()**.

Trabajos Batch

Por defecto, cada sentencia JDBC se envía individualmente a la base de datos. Aparte de las peticiones de red adicionales, este proceso provoca retrasos adicionales si la transación expande varias sentencias. JDBC 2.0 nos permite envíar varias sentencias a la vez con el método **addBatch**.

El siguiente método muestra cómo usar la sentencia **addBatch**. Las llamadas a **stmt.addBatch** añaden sentencias a la **Statement** original, y la llamada a **executeBatch** envía la sentencia completa con todos los apéndices a la base de datos.

```
Statement stmt = con.createStatement();
stmt.addBatch(
     "update registration set balance=balance-5.00
     where theuser="+theuser);
stmt.addBatch(
     "insert into auctionitems(
                    description, startprice)
     values("+description+","+startprice+")");

int[] results = stmt.executeBatch();
```

La hoja de resultados del método **addBatch** es un array de cuentas de filas afectadas por cada sentencia ejecutada en el trabajo batch. Si ocurre un problema se lanzará una **java.sql.BatchUpdateException**. Se puede obteener un array incompleto de contador de fila de **BatchUpdateException** llamando a su método **getUpdateCounts**.

Almacenar Clases, Imágenes y otros Objetos Grandes

Muchas bases de datos pueden almacenar datos binarios como parte de una fila si el campo es asignado como **long raw**, **longvarbinary**, u otro tipo similar. Esto campos pueden ocupar hasta 2 Gigabytes de datos. Esto significa que podemos convertir los datos en un stram binario o un array de bytes, puede ser almacenado o recuperado desde una base de datos como lo sería un string o un double.

Esta técnica peude usarse para almacenar y recuperar imágenes y objetos Java.

Almacenar y recuperar una imagen: Es muy fácil almacenar un objeto que

puede ser serializado o convertido en un array de bytes. Desafortunadamente **java.awt.Image** no es **Serializable**. Sin embargo, como se ve en el siguiente ejemplo de código, podemos almacenar los datos de la imagen en un fichero y almacenar la información del fichero como bytes en un campo binario de la base de datos.

```
int itemnumber=400456;

File file = new File(itemnumber+".jpg");
FileInputStream fis = new FileInputStream(file);
PreparedStatement pstmt = con.prepareStatement(
      "update auctionitems
      set theimage=? where id= ?");
pstmt.setBinaryStream(1, fis, (int)file.length()):
pstmt.setInt(2, itemnumber);
pstmt.executeUpdate();
pstmt.close();
fis.close();
```

Para recuperar esta imagen y crear un array de bytes que pueda ser pasado a **createImage**, hacemos los siguiente:

```
int itemnumber=400456;
byte[] imageBytes;

PreparedStatement pstmt = con.prepareStatement(
  "select theimage from auctionitems where id= ?");
pstmt.setInt(1, itemnumber);
ResultSet rs=pstmt.executeQuery();
if(rs.next()) {
  imageBytes = rs.getBytes(1);
}
pstmt.close();
rs.close();

Image auctionimage =
      Toolkit.getDefaultToolkit().createImage(
                                        imageBytes);
```

Almacenar y Recuperar un Objeto: Una clase puede ser serializada a un campo binario de la base de datos de la misma forma que se hizo con la imagen en el ejemplo anterior. En este ejemplo, la clase **RegistrationImpl** se ha modificado para soportar la serialización por defecto añadiendole **implements Serializable** a la declaración de la clase.

Luego, se crea un array **ByteArrayInputStream** para pasarlo como un Stream Binario a JDBC. Para crear el **ByteArrayInputStream**, **RegistrationImpl** primero

pasa a través de un **ObjectOutputStream** hacia el **ByteArrayInputStream** con una llamada a **RegistrationImpl.writeObject**. Luego el **ByteArrayInputStream** es convertido a un array de bytes, que puede ser utilizado para crear el **ByteArrayInputStream**. El método **create** en **RegistrationServer.java** se ha modificado de esta forma:

```
public registration.RegistrationPK create(
      String theuser,
      String password,
      String emailaddress,
      String creditcard)
      throws registration.CreateException{

   double balance=0;
   Connection con = null;
   PreparedStatement ps = null;;

   try {
      con=getConnection();
      RegistrationImpl reg= new RegistrationImpl();
      reg.theuser = theuser;
      reg.password = password;
      reg.emailaddress = emailaddress;
      reg.creditcard = creditcard;
      reg.balance = balance;

      ByteArrayOutputStream regStore =
              new ByteArrayOutputStream();
      ObjectOutputStream regObjectStream =
              new ObjectOutputStream(regStore);
         regObjectStream.writeObject(reg);

      byte[] regBytes=regStore.toByteArray();
      regObjectStream.close();
      regStore.close();
      ByteArrayInputStream regArrayStream =
              new ByteArrayInputStream(regBytes);
      ps=con.prepareStatement(
              "insert into registration (
              theuser, theclass) values (?, ?)");
      ps.setString(1, theuser);
      ps.setBinaryStream(2, regArrayStream,
           regBytes.length);

      if (ps.executeUpdate() != 1) {
```

```
            throw new CreateException ();
          }
          RegistrationPK primaryKey =
                          new RegistrationPKImpl();
          primaryKey.theuser(theuser);
          return primaryKey;
        } catch (IOException ioe) {
          throw new CreateException ();
        } catch (CreateException ce) {
          throw ce;
        } catch (SQLException sqe) {
          System.out.println("sqe="+sqe);
          throw new CreateException ();
        } finally {
          try {
             ps.close();
            con.close();
          } catch (Exception ignore) {
          }
        }
    }
```

El objeto es recuperado y resconstruido extrayendo los bytes desde la base de datos, creando un **ByteArrayInputStream** desde aquellos bytes leídos desde un **ObjectInputStream**, y llamando a **readObject** para crear de nuevo el ejemplar.

El siguiente ejemplo muestra los cambios necesarios en el método **RegistrationServer.refresh** para recuperar el ejemplar Registration desde la base de datos.

```
    private Registration refresh(RegistrationPK pk)
                  throws FinderException {

        if (pk == null) {
            throw new FinderException ();
        }
        Connection con = null;
        PreparedStatement ps = null;
        try {
           con=getConnection();
           ps=con.prepareStatement("
                  select theclass from
                  registration where theuser = ?");
           ps.setString(1, pk.theuser());
           ps.executeQuery();
           ResultSet rs = ps.getResultSet();
```

```
            if(rs.next()){
               byte[] regBytes = rs.getBytes(1);
               ByteArrayInputStream regArrayStream =
                   new ByteArrayInputStream(regBytes);
               ObjectInputStream regObjectStream =
                   new ObjectInputStream(
                         regArrayStream);
               RegistrationImpl reg=
                   (RegistrationImpl)
                          regObjectStream.readObject();
               return reg;
            }
            else {
                throw new FinderException ();
            }
        } catch (Exception sqe) {
            System.out.println("exception "+sqe);
            throw new FinderException ();
        }
        finally {
           try {
              rs.close();
              ps.close();
              con.close();
           }
           catch (Exception ignore) {}
        }
    }
```

BLOBs y CLOBs: Almacenar grandes campos en un tabla con otros datos no es necesariamente el lugar óptimo especialmente si los datos tienen un tamaño variable. una forma de manejar objetos de tamaño grande y variable es con el tipo "Large Objects" (LOBs). Este tipo usa un localizador, esencialmente un puntero, en el registro de la base de datos que apunta al campo real en la base de datos.

Hay dos tipos de LOBs: "Binary Large Objects" (BLOBs) y "Character Large Objects" (CLOBs). Cuando accedemos a BLOB o CLOB, los datos no se copian en el cliente. Para recuperar los datos reales desde una hoja de resultados, tenemos que recuperar el puntero con una llamada a **BLOB blob=getBlob(1)** o **CLOB clob=getClob(1)**, y luego recuperar los datos con una llamada a **blob.getBinaryStream()** o **clob.getBinaryStream()**.

Controlar Transaciones

Por defecto, las sentencias JDBC son procesadas en el modo full auto-commit. Este modo funciona bien para una sola consulta a la base de datos, pero si la operación

depende de varias sentencias de la base de datos que todas deben completarse con éxito o toda la operación será cancelada, se necesita una transación más adecuada.

Una descripción de los niveles de aislamiento en la transaciónse cubre con más detalles en el Capítulo 3: Manejaio de Datos y Transaciones. Para usar control de transaciones en la plataforma JDBC, primero necesitamos desactivar el moco "full auto-commit" llamando a:

```
Connection con= getConnection();
con.setAutoCommit(false);
```

En este punto, podemos **enviar** cualquier siguiente sentencia JDBC o deshacer cualquier actualización llamando al método **Connection.rollback**. La llamada **rollback** se sitúa normalmente en el manejador de excepciones, aunque puede situarse en cualquier lugar en le flujo de la transación.

El siguiente ejemplo inserta un ítem en la subasta y decrementa el balance del usuario. Si el balance es menor de cero, se deshace la transación complera y el ítem de susbasta es eliminado.

```
public int insertItem(String seller,
      String password,
      String description,
      int auctiondays,
      double startprice,
      String summary) {
   Connection con = null;
   int count=0;
   double balance=0;
   java.sql.Date enddate, startdate;
   Statement stmt=null;

   PreparedStatement ps = null;
   try {
     con=getConnection();
     con.setAutoCommit(false);
     stmt= con.createStatement();
     stmt.executeQuery(
            "select counter from auctionitems");
     ResultSet rs = stmt.getResultSet();
     if(rs.next()) {
       count=rs.getInt(1);
     }
     Calendar currenttime=Calendar.getInstance();
     java.util.Date currentdate=currenttime.getTime();
```

```
startdate=new java.sql.Date(
                currentdate.getTime());
currenttime.add(Calendar.DATE, auctiondays);
enddate=new java.sql.Date((
         currenttime.getTime()).getTime());

ps=con.prepareStatement(
         "insert into auctionitems(
         id, description, startdate, enddate,
         startprice, summary)
         values (?,?,?,?,?,?)");
ps.setInt(1, count);
ps.setString(2, description);
ps.setDate(3, startdate);
ps.setDate(4, enddate);
ps.setDouble(5, startprice);
ps.setString(6, summary);
ps.executeUpdate();
ps.close();

ps=con.prepareStatement(
         "update registration
         set balance=balance -0.50
         where theuser= ?");
ps.setString(1, seller);
ps.close();
stmt= con.createStatement();
stmt.executeQuery(
         "select balance from registration
         where theuser='"+seller+"'");
rs = stmt.getResultSet();
if(rs.next()) {
  balance=rs.getDouble(1);
}
stmt.close();
if(balance <0) {
  con.rollback();
  con.close();
  return (-1);
}

stmt= con.createStatement();
stmt.executeUpdate(
         "update auctionitems set
         counter=counter+1");
stmt.close();
```

```
      con.commit();
      con.close();
      return(0);
    } catch(SQLException e) {
      try {
        con.rollback();
        con.close();
        stmt.close();
        ps.close();
      }catch (Exception ignore){}
    }
    return (0);
  }
```

Caracteres de Escape

El API JDBC proporciona la palabr clave **escape** para que podamos especificar el caracter que querramos usar como caracter de escape. Por ejemplo, si queremos usar el signo de tanto por ciento (**%**) como el símbolo de tanto por ciento que que no se interprete como un comodín SQL usando en consultas SQL **LIKE**, tenemos que escaparlo con el caracter de escape que especifiquemos con la palabra clave **escape**.

La siguiente sentencia muestra cómo podemos usar la palabra clave **escape** para buscar por el valor **10%**:

```
stmt.executeQuery(
 "select tax from sales where tax like
     '10\%' {escape '\'}");
```

Si nuestro programa almacena nombres y direcciones en la base de datos introducidos desde la línea de comandos o desde un interface de usuario, el símbolo de comilla simple (**'**) podría aparecer en los datos. Pasar una comilla simple directamente a un string SQL causa problemas cuando la sentencia es analizada porque SQL le da a este símbolo otro significado a menos que se le escape.

Para resolver este problem, el siguiente método escapa cualquier símbolo **'** encontrado en la línea de entrada. Este método puede ser extendido para escapar cualquier otro caracter como las comas **,** que la base de datos o su driver podrían interpretar de otra forma:

```
static public String escapeLine(String s) {
  String retvalue = s;
  if (s.indexOf ("'") != -1 ) {
    StringBuffer hold = new StringBuffer();
```

```
      char c;
      for(int i=0; i < s.length(); i++ ) {
        if ((c=s.charAt(i)) == '\'' ) {
        hold.append ("''");
      }else {
        hold.append(c);
      }
    }
    retvalue = hold.toString();
    }
    return retvalue;
  }
```

Sin embargo, si usamos un **PreparedStatement** en lugar de una simple **Statement**, muchos de estos problemas de escape desaparecen. Por ejemplo, en lugar de esta línea con la secuencia de escape:

```
stmt.executeQuery(
"select tax from sales where tax like
     '10\%' {escape '\'}");
```

Podríamos usar esta línea:

```
preparedstmt = C.prepareStatement(
                   "update tax set tax = ?");
```

Mapear Tipos de Base de Datos

Aparte de unos pocos tipos como **INTEGER** que son representados como **INTEGER** en las bases de datos más populares, podríamos encontrar que el tipo JDBC de una columna de la tabla no corresponde con el tipo representado en la base de datos. Esto significa que lllamar a **ResultSet.getObject**, **PreparedStatement.setObject** y **CallableStatement.getObject()** fallará bastantes veces.

Nuestro programa puede determinar los tipos de las columnas de la base de datos desde los datos meta de la base de datos y usar esta información para chequear el valor antes de recuperarlo. Este código chequea que el valor es del tipo **INTEGER** antes de recuperarlo.

```
    int count=0;
    Connection con=getConnection();
    Statement stmt= con.createStatement();
    stmt.executeQuery(
           "select counter from auctionitems");
    ResultSet rs = stmt.getResultSet();
    if(rs.next()) {
```

```
  if(rs.getMetaData().getColumnType(1) ==
                              Types.INTEGER) {
  Integer i=(Integer)rs.getObject(1);
  count=i.intValue();
  }
}
rs.close();
```

Mapeo de Tipos Date

El tipo **DATE** es donde ocurren más errores. Es porque la clase **java.util.Date** representa tanto la Fecha como la Hora, pero SQL tiene estos tres tipos para representar informaciónde fecha y hora:

- Un tipo **DATE** que representa sólo fechas (03/23/99).
- Un tipo **TIME** que específica sólo la hora (12:03:59).
- Un tipo **TIMESTAMP** que representa el valor de la hora en nanosegundos.

Estos tres tipos adiciones los proporciona el paquete **java.sql** como **java.sql.Date**, **java.sql.Time** y **java.sql.Timestamp** y son todos suclases de **java.util.Date**. Esto significa que podemos usar valores **java.util.Date** convertidos al tipo necesario para que sean compatibles con el tipo de la base de datos.

Nota: la clase **Timestamp** pierde precisión cuando se convierte a **java.util.Date** porque **java.util.Date** no contiene un campo de nanosegundos, es mejro no convertir un ejemplar **Timestamp** si el valor va a ser escrito de vuelta en la base de datos.

Este ejemplo usa la clase **java.sql.Date** para convertir el valor **java.util.Date** devuelto por la llamada a **Calendar.getTime** hacia **java.sql.Date**.

```
Calendar currenttime=Calendar.getInstance();
java.sql.Date startdate=
   new java.sql.Date((
         currenttime.getTime()).getTime());
```

También podemo usar la clase **java.text.SimpleDateFormat** para hacer la conversión. Este ejemplo usa la clase **java.text.SimpleDateFormat** para convertir un objeto **java.util.Date** a un objeto **java.sql.Date**:

```
SimpleDateFormat template =
      new SimpleDateFormat("yyyy-MM-dd");
java.util.Date enddate =
      new java.util.Date("10/31/99");
java.sql.Date sqlDate =
      java.sql.Date.valueOf(
```

```
                                template.format(enddate));
```

Si encontramos que una representación de fecha de una base de datos no puede ser mapeada a un tipo Java con una llamada a **getObject** o **getDate**, recuperamos el valor con una llamada a **getString** y formateamos el string como un valor **Date** usando la clase **SimpleDateFormat** mostrada arriba.

_____

[1] Cuando se usan en toda esta site, los términos, "Java virtual machine" o "JVM" significa una máquina virtual de la plataforma Java.

Servlets

Un servelt es un programa del lado del servidor escrito en lenguaje Java que interactúa con clientes y que normalmente está unido a unservidor de "HyperText Transfer Protocol" (HTTP). Uno uso común para un servlet es ampliar un servidor web proporcionando contenidos web dinámicos.

Los servelts tienen la ventaja sobre otras tecnologías que de están compilados, tienen capacidad de threads interna, y proporcionan un entorno de programación seguro. Incluso las sites web que antes no proporcionaban soporte para servlets, pueden hacerlo ahora usando programas como JRun o el módulo Java para el servidor Web Apache.

La aplicación subastas basada en web usa un servelt para aceptar y procesar entradas del comprador y vendedor a través del navegador y devuelve dinámicamente información sobre el ítem de la subasta hacia el navegador. El programa **AuctionServlet** se creo extendiendo la clase **HttpServlet**. Esta clase proporciona un marco de trabajo para manejar peticiones y respuestas HTTP.

Esta sección examina el **AuctionServlet** e incluye información sobre cómo usar objetos **Cookie** y **Session** en un servlet.

HttpServlet

La clase AuctionServlet extiende la clase **HttpServlet**, que es una clase abastracta.

```
public class AuctionServlet extends HttpServlet {
```

Un servlet puede cargarse cuando se arranca el servidor web o cuando se solicita una petición HTTP a una URL que especifica el servlet. El servlet normalmente es cargado mediante un cargador de clases separado en el sevidor web porque esto permite que el servlet sea recargado descargando el cargador de clases que cargo la clase servlet. Sin embargo, si el servlet depende de otras clases y una de estas clases cambia, necesitaremos actualiza el sello de la fecha del servlet para recargarlo.

Después de cargar un servlet, el primer estado en su ciclo de vida es la llamada a su método **init** por parte del servidor web. Una vez cargado e inicializado, el siguiente estado en el ciclo de vida del servlet es para servir peticiones. El servlet sirve peticiones a través de las implementaciones de su métodos **service**, **doGet**, o **doPost**.

El servlet puede opcionalmente implementar un método **destroy** para realizar operaciones de limpieza antes de que el servidor web descarge el servlet.

El método init

El método **init** sólo se llama una vez por el servidor web cuando se arranca el servlet por primera vez. A este método se le pasa un objeto **ServletConfig** que contiene la información de inicialización perteniente al servidor web donde se está ejecutando la aplicación.

El objeto **ServletConfig** es usado para acceder a la información mantenida por el servidor web incluyendo valores del parámetro **initArgs** en el fichero de propiedades del servlet. El código del método **init** usa el objeto **ServletConfig** para recuperar los valores de **initArgs** llamando al método **config.getInitParameter("parameter")**.

El método **AuctionServlet.init** también contacta con el servidor de JavaBeans Enterprise para crear un objeto contexto (**ctx**). Este objeto e susado en el método **service** para establecer una conexión con el servidor de JavaBeans Enterprise.

```
Context ctx=null;
private String detailsTemplate;

public void init(ServletConfig config)
                throws ServletException{
  super.init(config);
```

```
    try {
      ctx = getInitialContext();
    }catch (Exception e){
      System.err.println(
                  "failed to contact EJB server"+e);
    }
    try {
      detailsTemplate=readFile(
              config.getInitParameter("detailstemplate"));
    } catch(IOException e) {
      System.err.println(
                  "Error in AuctionServlet <init>"+e);
    }
  }
```

El método destroy

El método **destroy** es un método de ciclo de vida implementado por servlets que necesitan grabar su estado entre cargas y descargas del servlet. Por ejemplo, el método **destroy** podría gabar el estado actual del servlet, y la siguiente vez que el servlet sea cargado, el estado grabado podría ser recuperado por el método **init**. Deberíamos tener cuidado con que no se podría haber llamado al método **destroy** si la máquina servidor se bloquea.

```
  public void destroy() {
    saveServletState();
  }
```

El método service

El **AuctionServlet** es un servlet HTTP que maneja peticiones de clientes y genera respuestas a través de su método **service**. Acepta como parámetros los objetos de petición y respuesta **HttpServletRequest** y **HttpServletResponse**.

- **HttpServletRequest** conteine las cabeceras y los streams de entrada desde el cliente hacia el servidor.
- **HttpServletResponse** es el stream de salida que se utiliza para enviar información de vuelta desde el servidor hacia el cliente.

El método **service** maneja peticiones HTTP estándars del cliente recibidas mediante su parámetro **HttpServletRequest** y delengando la petición a uno de los siguientes métodos designados para manejar peticiones. Los diferentes tipos de peticiones se describen en la sección Peticiones HTTP.

- doGet para GET, GET condicional, y peticiones HEAD.
- doPost para peticiones POST.
- doPut para peticiones PUT.

- doDelete para peticiones DELETE.
- doOptions para peticiones OPTIONS.
- doTrace para peticiones TRACE.

el programa **AuctionServlet** proporciona su propia implementación del método **service** que llama a uno de los siguiente métodos basándose en el valor devuelto por la llamada a **cmd=request.getParameter("action")**. Estas implementaciones de métodos corresponden a las implementacione por defecto proporcionadas por los métodos **doGet** y **doPost** llamadas por el método **service**, pero añade algunas funcionalidades específicas de la aplicación subasta para buscar Beans Enterprise.

- listAllItems(out)
- listAllNewItems(out)
- listClosingItems(out)
- insertItem(out, request)
- itemDetails(out, request)
- itemBid(out, request)
- registerUser(out, request)

```
public void service(HttpServletRequest request,
                HttpServletResponse response)
                throws IOException {

  String cmd;
  response.setContentType("text/html");
  ServletOutputStream out = response.getOutputStream();
  if (ctx == null ) {
    try {
      ctx = getInitialContext();
    }catch (Exception e){
      System.err.println(
                  "failed to contact EJB server"+e);
    }
  }

  cmd=request.getParameter("action");
  if(cmd !=null)  {
    if(cmd.equals("list")) {
      listAllItems(out);
    }else
    if(cmd.equals("newlist")) {
      listAllNewItems(out);
    }else if(cmd.equals("search")) {
```

```
        searchItems(out, request);
      }else if(cmd.equals("close")) {
        listClosingItems(out);
      }else if(cmd.equals("insert")) {
        insertItem(out, request);
      }else if (cmd.equals("details")) {
        itemDetails(out, request );
      }else if (cmd.equals("bid")) {
        itemBid(out, request) ;
      }else if (cmd.equals("register")) {
        registerUser(out, request);
      }
    }else{
      // no command set
      setTitle(out, "error");
    }
    setFooter(out);
    out.flush();
  }
```

Peticiones HTTP

Una petición es un mensaje enviado desde un programa cliente como un navegador a un programa servidor. La primera línea del mensaje de petición contiene un método que indica la acción a realizar sobre la URL que viene después. Los dos mecanismos más comunes para enviar información al servidor son **POST** y **GET**.

- Las peticiones GET podrían pasar parámetros a una URL añadiéndolas a la URL. Estas peticiones pueden ser guardadas en el bookmark o enviadas por correro e incluyen la información de la URL de respuesta.
- Las peticiones POST podrían pasar datos adicionales a la URL enviándolas directamente al servidor de forma separada a la URL. Estas peticiones no pueden ser almacenadas en el bookmark ni enviadas por email y no cambiar la URL de la respuesta.

Las peticiones PUT son la inversa de la peticiones GET. En lugar de leer la página, las peticiones PUT escriben (o almacenan) la página.

Las peticiones DELETE son para eliminar páginas Web.

Las peticiones OPTIONS son para obtener información sobre las opciones de comunicación disponibles en la cadena petición/respuesta.

Las peticiones TRACE son para realizar pruebas de diagnóstico porque permite que el cliente vea lo que se está recibiendo al orto final de la cadena de petición.

Usar Cookies en servlets

LAs cookies HTTP son exencialmente cabeceras HTTP personalizadas que son pasadas entre el cliente y el servidor. Aunque las cookies no son muy populares, permiten que el estado sea compartido entre dos máquinas. Por ejemplo, cuando un usuario hace login en una site, una cookie puede mantener una referencia verificando que el usuario ha pasado el chequeo de password y puede usar esta referencia para identificar al mismo usuario en futuras visitas.

Las cookies normalmente están asociadas con un servidor. Si configuramos el dominio a **.java.sun.com**, entonces la cookies está asociada con ese dominio. Si no se configura nignún dominio, la cookie sólo está asociada con el servidor que creó la cookie.

Configurar una Cookie

El API Servlet de Java incluye una clase **Cookie** que podemos usar para configurar o recuperar la cookie desde la cabecera HTTP. Las cookies HTTP incluyen un nombre y una pareja de valores.

El método **startSession** mostrado aquí está en el programa LoginServlet. En este método, el nombre en la pareja nombre valor usado para crea el **Cookie** es **JDCAUCTION**, y un identificador único generado por el servidor es el valor.

```
protected Session startSession(String theuser,
              String password,
              HttpServletResponse response) {
  Session session = null;
  if ( verifyPassword(theuser, password) ) {
  // Create a session
    session = new Session (theuser);
    session.setExpires (sessionTimeout + i
      System.currentTimeMillis());
    sessionCache.put (session);

  // Create a client cookie
    Cookie c = new Cookie("JDCAUCTION",
      String.valueOf(session.getId()));
    c.setPath ("/");
    c.setMaxAge (-1);
    c.setDomain (domain);
    response.addCookie (c);
  }
  return session;
}
```

Versiones posteriores del API Servlet incluye un API Session, para crear una sesión

usando el API Servelt en el ejemplo anterior podemos usar el método **getSession**.

```
HttpSession session = new Session (true);
```

El método **startSession** es llamado mediante peticiones de acción login desde un **POST** al **LoginServlet** de esta forma:

```
<FORM ACTION="/LoginServlet" METHOD="POST">
<TABLE>
<INPUT TYPE="HIDDEN" NAME="action" VALUE="login">
<TR>
<TD>Enter your user id:</TD>
<TD><INPUT TYPE="TEXT" SIZE=20
  NAME="theuser"></TD>
</TR>
<TR>
<TD>Enter your password:<TD>
<TD><INPUT TYPE="PASSWORD" SIZE=20
  NAME="password"></TD>
</TR>
</TABLE>
<INPUT TYPE="SUBMIT" VALUE="Login" NAME="Enter">
</FORM>
```

La cookie es creada con un edad máxima de -1, lo que significa que el cookie es almacenado pero permanece vivo miesntras el navegador se esté ejecutando. El valor se selecciona en segunod, aunque cuando s eusan valores menores que unos pocos segundos necesitamos tener cuidado con que los tiempos de las máquinas pudieran estar ligeramente desincronizados.

El valor de path puede ser usado para especificar que el cookie sólo se aplica a directorios y ficheros bajo el path seleccionado en esa máquina. En este ejemplo, el path raíz **/** significa que el cookie es aplicable a todos los directorios.

El valor del dominio en este ejemplo es leído desde los parámetros de inicialización del servlet. Si el dominio es **null**, el cookie es sólo aplicado a esa máquina de domino.

Recuperar un Cookie

El cookie es recuperado desde las cabeceras HTTP con una llamada al método **getCookies** para solicitarlo:

```
Cookie c[] = request.getCookies();
```

Posteriormente podemos recuperar la pareja de selecciones nombre y valor llamando al método **Cookie.getName** para recuperar el nombre y al método **Cookie.getValue** para recuperar el valor.

LoginServlet tiene un método **validateSession** que chequea las cookies del usuario para encontrar un cookie **JDCAUCTION** que fué enviada en este dominio:

```
private Session validateSession
              (HttpServletRequest request,
              HttpServletResponse response) {
  Cookie c[] = request.getCookies();
  Session session = null;
  if( c != null ) {
    Hashtable sessionTable = new Hashtable();
    for (int i=0; i < c.length &&
           session == null; i++ ) {
      if(c[i].getName().equals("JDCAUCTION")) {
        String key = String.valueOf (c[i].getValue());
        session=sessionCache.get(key);
      }
    }
  }
  return session;
}
```

Si usamos el API Servlet podemos usar el siguiente método, observamos que el parámetro es false para especificar que el valor de sesión es devuelto y que no se cree una nueva sesión:

```
HttpSession session = request.getSession(false);
```

Generar Sesiones

El método **LoginServlet.validateSession** devuelve un objeto **Session** representado por la clase Session. Esta clase usa un generado desde una secuencia numérica. Esta identificador de sesión numerada es la parte del valor de la pareja de nombe y valor almacenadas en el cookie.

La única forma de referenciar el nombre del usuario en el servidor es con este identificador de sesión, que está almacenado en un sencillo caché de memoria con los otros identificadores de sesión. Cuando un usuario termina una sesión, se llama a la acción **LoginServlet** de esta forma:

```
http://localhost:7001/LoginServlet?action=logout
```

El caché de sesión implementado en el programa SessionCache.java incluye un thread para eliminar sesiones más viejas que el tiempo preseleccionado. Este tiempo podría medise en horas o días, dependiendo del tráfico de la web site.

Evitar el Caché de Páginas

El método **LoginServlet.setNoCache** selecciona los valores **Cache-Control** o **Pragma** (dependiendo de la versión del protocolo HTTP que estemos usando) en la cabecera de respuesta a **no-cache**. La cabecera de expiración **Expires** también se selecciona a 0, alternativamente podemos seleccionar la hora para que se la hora actual del sistema. Incluso si el cliente no cachea la página, frecuentemente hay servidores proxys en una red corporativa que si lo harían. Sólo las páginas que usan Secure Socket Layer (SSL) no son cacheadas por defecto.

```
private void setNoCache (HttpServletRequest request,
                HttpServletResponse response) {
  if(request.getProtocol().compareTo ("HTTP/1.0") == 0) {
    response.setHeader ("Pragma", "no-cache");
  } else if (request.getProtocol().compareTo
                ("HTTP/1.1") == 0) {
    response.setHeader ("Cache-Control", "no-cache");
  }
  response.setDateHeader ("Expires", 0);
}
```

Restringir Accesos y Redireccionamientos

Si instalamos el **LoginServlet** como el servlet por defecto o el servler a ejecutar cuando se sirva cualquier página bajo el documento raiz, odemos usar cookies para restringir los usuarios a ciertas secciones de la site. Por ejemplo, podemos permitir que los usuarios que tengan cookies con el estado de que han introducido su passweord acceder a secciones de la site que requieren un login y mantener a los otros fuera.

El programa LoginServlet chequea un directorio restringido en este método **init**. El método **init** mostrado abajo configura la variable **protectedDir** a **true** si la variable **config** pasada a él especifica un directorio protegido. El fichero de configuración del servidor Web proporciona las configuraciones pasadas a un servlet en la variable **config**.

```
public void init(ServletConfig config)
                  throws ServletException  {
  super.init(config);
  domain = config.getInitParameter("domain");
  restricted = config.getInitParameter("restricted");
  if(restricted != null) {
    protectedDir=true;
  }
```

Más tarde en los métodos **validateSession** y **service**, la variable **protectedDir** es comprobada y se llama al método **HttpResponse.sendRedirect** para viar al usuario a la página correcta basándose en sus estados de login y sesión

```
if(protectedDir) {
  response.sendRedirect (restricted+"/index.html");
}else{
  response.sendRedirect (defaultPage);
}
```

El método **init** también recupera el contexto del servlet para el servlet **FileServlet** para que los métodos puedan ser llamados sobre **FileServlet** en el método **validateSession**. La ventaja de llamar a los métodos sobre el servlet **FileServlet** para servir los ficheros desde dentro del servlet **LoginServlet**, es que obtenemos todas las ventajas de la funcionalidades añadidas dentro del servlet **FileServlet** como el mepeo de memoria o el chaché de ficheros. La parte negativa es que el código podría no ser portable a otros servidores que no tengan un servlet **FileServlet**. Este código recupera el contexto **FileServlet**:

```
FileServlet fileServlet=(FileServlet)
      config.getServletContext().getServlet("file");
```

El método **validateSession** evita que los usuarios sin login de sesión accedan a los directorios restringidos.

Códigos de Error HTTP

Podemos devolver un código de error HTTP usando el método **sendError**. Por ejemplo, el código de error HTTP 500 indica un error interno en el seridor, y el código de error 404 indica página no encontrada. Este segmento de código devuelve el código de error HTTP 500:

```
protected void service (HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException {
   response.sendError (500);
}
```

Leer Valores GET y POST

El API Servlet tiene un método **getParameter** en la clase **HttpServletRequest** que devuelve el valor **GET** o **POST** para el nombre suministrado.

- La petición HTTP **GET** maneja parejas de nombre/valor como parte de la URL. El método **getParameter** analiza la URL pasada, recupera las parejas **name=value** determinadas por el caracter (**&**), y devuelve el valor.
- La petición HTTP **POST** lle el nombre de las parejas nombre/valor desde el stream de entrada desde el cliente. El método **getParameter** analiza en el stream de entrada las parejas de nombre/valor.

El método **getParameter** funciona vien para servlet sencillos, pero si necesitamos recuperar los parámetros **POST** en el orden enque fueron situados en la página wev o manejar posts multi-parte, podemos escribir nuestro propio para analizar el stram de entrada.

El siguiente ejemplo devuelve los parámetros POST en el orden en que fueron recibidos desde la página Web. Normalmento, los parámetros son almacenados en un **Hashtable** que no mantiene el orden de secuencia de los elementos almacenados. El ejemplo mantiene una referencia a cada pareja nombre/valoren un vector que puede ser ser analizado para devolver valores en el orden en que fueron recibidos por el servidor.

```
package auction;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PostServlet extends HttpServlet {
  private Vector paramOrder;
  private Hashtable parameters;

  public void init(ServletConfig config)
        throws ServletException  {
    super.init(config);
  }


  public void service(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    if(request.getMethod().equals("POST")
        && request.getContentType().equals(
        "application/x-www-form-urlencoded")) {

      parameters=parsePostData(
                  request.getContentLength(),
                  request.getInputStream());
    }

    for(int i=0;i<paramOrder.size();i++) {
      String name=(String)paramOrder.elementAt(i);
```

```
    String value=getParameter((
                    String)paramOrder.elementAt(i));
    out.println("name="+name+" value="+value);
  }
  out.println("</body></html>");
  out.close();
}

private Hashtable parsePostData(int length,
      ServletInputStream instream) {
  String valArray[] = null;
  int inputLen, offset;
  byte[] postedBytes = null;
  boolean dataRemaining=true;
  String postedBody;
  Hashtable ht = new Hashtable();
  paramOrder= new Vector(10);
  StringBuffer sb = new StringBuffer();

  if (length <=0) {
    return null;
  }
  postedBytes = new byte[length];
  try {
     offset = 0;
     while(dataRemaining) {
       inputLen = instream.read (postedBytes,
                      offset,
                      length - offset);
       if (inputLen <= 0) {
         throw new IOException ("read error");
       }
       offset += inputLen;
       if((length-offset) ==0) {
         dataRemaining=false;
       }
     }
  } catch (IOException e) {
    System.out.println("Exception ="+e);
    return null;
  }

  postedBody = new String (postedBytes);
  StringTokenizer st =
          new StringTokenizer(postedBody, "&");
```

```
  String key=null;
  String val=null;

  while (st.hasMoreTokens()) {
    String pair = (String)st.nextToken();
    int pos = pair.indexOf('=');
    if (pos == -1) {
      throw new IllegalArgumentException();
    }
    try {
       key = java.net.URLDecoder.decode(
                      pair.substring(0, pos));
       val = java.net.URLDecoder.decode(
                      pair.substring(pos+1,
                      pair.length()));
    } catch (Exception e) {
       throw new IllegalArgumentException();
    }
    if (ht.containsKey(key)) {
      String oldVals[] = (String []) ht.get(key);
      valArray = new String[oldVals.length + 1];
      for (int i = 0; i < oldVals.length; i++) {
         valArray[i] = oldVals[i];
      }
      valArray[oldVals.length] = val;
    } else {
      valArray = new String[1];
      valArray[0] = val;
    }
    ht.put(key, valArray);
    paramOrder.addElement(key);
  }
  return ht;
}

public String getParameter(String name) {
  String vals[] = (String []) parameters.get(name);
  if (vals == null) {
    return null;
  }
  String vallist = vals[0];
  for (int i = 1; i < vals.length; i++) {
    vallist = vallist + "," + vals[i];
  }
  return vallist;
```

```
  }
}
```

Para saber si una petición es POST o GET, llamados al método **getMethod** de la clase **HttpServletRequest**. Para determinar el formato de los datos que están siendo posteados, llamamos al método **getContentType** de la clase **HttpServletRequest**. Para sencillas páginas HTML, el tipo devuelto por está llamada será **application/x-www-form-urlencoded**.

Si necesitamos crear un post con más de una parte como el creado por el siguiente formulario HTML, el servler necesitará ller el stream de entrada desde el post para alcanzar las secciones individuales. Cada sección se dstingue por un límite definido en la cabecera post.

```
<FORM ACTION="/PostMultiServlet"
   METHOD="POST" ENCTYPE="multipart/form-data">
<INPUT TYPE="TEXT" NAME="desc" value="">
<INPUT TYPE="FILE" NAME="filecontents" value="">
<INPUT TYPE="SUBMIT" VALUE="Submit" NAME="Submit">
</FORM>
```

El siguiente ejemplo extrae una descripción y un fichero desde los navegadores del cliente. Lee el stream de entrada buscando una línea que corresponda con un string de límite, lee el contenido de la línea y lueo lee los datos asociados con esa parte. El fichero suvido se muestra simplemente, pero también puede ser escrito en disco:

```
package auction;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PostMultiServlet extends HttpServlet {

   public void init(ServletConfig config)
        throws ServletException  {
     super.init(config);
   }

   public void service(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

     response.setContentType("text/html");
     PrintWriter out = response.getWriter();
```

```
    if (request.getMethod().equals("POST")
       && request.getContentType().startsWith(
       "multipart/form-data")) {

      int index = request.getContentType().indexOf(
                                "boundary=");
      if (index < 0) {
        System.out.println("can't find boundary type");
        return;
      }

      String boundary =
               request.getContentType().substring(
                                           index+9);
      ServletInputStream instream =
                             request.getInputStream();
      byte[] tmpbuffer = new byte[8192];
      int length=0;
      String inputLine=null;
      boolean moreData=true;

//Skip until form data is reached
      length = instream.readLine(
                       tmpbuffer,
                       0,
                       tmpbuffer.length);
      inputLine = new String (tmpbuffer, 0, 0,
                               length);

      while(inputLine.indexOf(boundary)
                        >0 && moreData) {
        length = instream.readLine(
                       tmpbuffer,
                       0,
                       tmpbuffer.length);
        inputLine = new String (tmpbuffer, 0, 0,
                                 length);
        if(inputLine !=null)
          System.out.println("input="+inputLine);
          if(length<0) {
            moreData=false;
          }
        }

        if(moreData) {
          length = instream.readLine(
```

```
                tmpbuffer,
                0,
                tmpbuffer.length);
  inputLine = new String (tmpbuffer, 0, 0,
                          length);

  if(inputLine.indexOf("desc") >=0) {
    length = instream.readLine(
                tmpbuffer,
                0,
                tmpbuffer.length);
    inputLine = new String (tmpbuffer, 0, 0,
                            length);
    length = instream.readLine(
                tmpbuffer,
                0,
                tmpbuffer.length);
    inputLine = new String (tmpbuffer, 0, 0,
                            length);
    System.out.println("desc="+inputLine);
  }
}

while(inputLine.indexOf(boundary)
                  >0 && moreData) {
  length = instream.readLine(
                tmpbuffer,
                0,
                tmpbuffer.length);
  inputLine = new String (tmpbuffer, 0, 0,
                          length);
}
if(moreData) {
  length = instream.readLine(
                tmpbuffer,
                0,
                tmpbuffer.length);
  inputLine = new String (tmpbuffer, 0, 0,
                          length);

  if(inputLine.indexOf("filename") >=0) {
    int startindex=inputLine.indexOf(
                              "filename");
    System.out.println("file name="+
                inputLine.substring(
                startindex+10,
```

```
                        inputLine.indexOf("\"",
                        startindex+10)));
            length = instream.readLine(
                        tmpbuffer,
                        0,
                        tmpbuffer.length);
            inputLine = new String (tmpbuffer, 0, 0,
                                    length);
          }
        }
        byte fileBytes[]=new byte[50000];
        int offset=0;
        if (moreData) {
          while(inputLine.indexOf(boundary)
                              >0 && moreData) {
            length = instream.readLine(
                        tmpbuffer,
                        0,
                        tmpbuffer.length);
            inputLine = new String (tmpbuffer, 0, 0, length);
            if(length>0 && (
                 inputLine.indexOf(boundary) <0)) {
              System.arraycopy(
                        tmpbuffer,
                        0,
                        fileBytes,
                        offset,
                        length);
              offset+=length;
            } else {
             moreData=false;
            }
          }
        }
// trim last two newline/return characters
// before using data
        for(int i=0;i<offset-2;i++) {
          System.out.print((char)fileBytes[i]);
        }
      }
      out.println("</body></html>");
      out.close();
  }
}
```

Threads

Un servlet debe ser capaz de manejar múltipels peticiones concurrentes. Cualquier número de usuarios puede en un momento dado invocar al servlet, y mientras que el método **init** ejecuta siempre un único trehad, el método **service** es multi-thread para manejar múltiples peticiones.

Esto significa que cualquier campo estático o público accedido por el método **service** deberían estár restringidos a accesos de un thread. el ejemplo de abajo usa la palabra clave **synchronized** para restringir el acceso a un contador para que sólo pueda ser actualizado por un thread a la vez:

```
int counter
Boolean lock = new Boolean(true);

synchronized(lock){
  counter++;
}
```

HTTPS

Muchos servidores, navegadores, y el java Plug-In tiene la posibilidad de soportar el protocolo HTTP seguro llamado HTTPS. Este similar al HTTP excepto en que los datos on tramitidos a través de una capa de socket seguro (SSL) en lugar de una conexión de socket normal. Los navegadores web escuchan peticiones HTTP en un puerto mientras escuchan las peticiones HTTPS en otro puerto.

Los datos encriptados que son enviados a través de la red incluyen chequeos para verificar si los dato se han modificado en el tránsito. SSL también autentifica el servidor web a sus clientes proporcionando un certificado de clave pública. en el SSL 3.0 el cliente también puede autentificarse a sí mismo con el servidor, usxando de nuevo un certificado de clave pública.

La clave pública criptográfica (también llamada clave de encriptación asimétrerica) usa una pareja de claves pública y privada. Cualquier mensaje encriptado (hecho ininteligible) con la clave privada de la pareja sólo puede ser desencriptado con la correspondiente clave pública. Los certificados son sentencias firmadas digitalmente generadas por un tercera parte conocidad como "Autoridad de Certificación" Certificate Authority. Esta Autorizar necesita asegurarse de que nosotros somos quien decimos ser porque los clientes se creeran el certificado que reciban. Si es así, este certificado puede contener la clave pública de la pareja de clave pública/privada. El certificado está firmado por la clave privada de la Autoridad de Certificación, y muchos navegadores conocen las claves públicas la mayoría de las Autoridades de Certificación.

Mientras que la encriptaciónde clavepública es buena para propósitos de autentificación, no es tan rápida como la encriptación asimétrica y por eso el

protocolo SSL usa ambos tipos de claves en el ciclo de vida de una conexión SSL. El cliente y el servidor empiezan una transación HTTPS con una inicialización de conexión o fase de estrechamiento de manos.

Es en ese momento en el que el servidor es autentificado usando el certificado que el cliente ha recibido. El cliente usa la clave pública del servidor para encriptar los mensajes enviados al servidor. Después de que el cliente haya sido autentificado y el algoritmo de encriptación se ha puesto de acuerdo entre las dos partes, se usan unas nuevas claves de sesión simétrica para encriptar y desencriptar las comunicaciones posteriores.

El algoritmo de encriptación puede ser uno de los más populares algoritmos como "Rivest Shamir and Adleman" (RSA) o "Data Encryption Standard" (DES). Cuando mayor sea el número de bits usados para crear la clave, mayores dificultades para poder romper las claves mediante la fuerza bruta.

HTTPS usando criptografía de clave pública y certificados nos permite proporcionar una gran privacidad a las aplicacioens que necesitan transaciones seguras. Los servidores, navegadores y Java Plug-In tienen sus propias configuraciones para permitir usar Comunicaciones SSL. En general, estos pasos requieren:

- Obtener una clave privada y un certificado firmado digitalmente con la clave pública correspondente.
- Instalar el certificado en una localización especificada por el software que estamos usando (servidor, navegador o Java Plug-In).
- Activar las características SSL y especificar nuestros ficheros de certificado y de clave privada como se explica en nuestra documentación.

Siempre que activemos las características SSL de acuerdo con los requerimientos de la aplicación dependiendo del nivel de seguridad de necesitemos. Por ejemplo no necesitamos verificar la autenticidad de los clientes para navegar por los ítems de la subasta, pero sí querremos ecriptar la información de la tarjeta de crédido y otras informaciones suministradas cuando los compradores y vendedores se registran para participar.

HTTPS puede ser usado para cualquier dato, no sólo ara páginas web HTTP. Los programas escritos en lenguaje Java pueden ser descaradoa a trravés de conexiones HTTPS, y podemos abrir una conexión con un servidor HTTPS en el Java Plug-In. Para escribir un programa en Java que use SSL, este necesita una librería SSL y un conecimiento detallado del proceso de negociación HTTPS. Nuestra librería SSL podría cubir los pasos necesarios ya que está información es restringida por el control de exportación de seguridad.

Tecnología JNI

La plataforma Java es relativamente nueva, lo que significa qie algunas veces podríamos necesitar integrar programas escritos en Java con servicios, programas o APIs existentes escritos en lenguajes distintos en Java. La plataforma Java proporciona el Interfa Nativo Java (JNI) para ayudarnos con este tipo de integración.

El JNI define una convención de nombres y llamadas para que la Máquina Virtual Java[1] pueda localizar e invocar a los métodos nativos. De hecho, JNI está construido dentro de la máquina virtual Java, por lo que ésta puede llamar a sistemas locales para realizar entrada/salida, g´raficos, trabajos de red y operaciones de threads sobre el host del sistema operativo.

Este capítulo explica como usar JNI en programas escritos en Java para llamar a cualquier librería de la máquina local, llamar a métodos del lenguaje Java desde dentro del código nativo, y cómo crear y ejecutar un ejemplar de la JVM. Para mostrar cómo podemos hacer funcionar el JNI, los ejemplos de este capítulo incluyen integración de JNI con el API de bases de datos Xbase de C++. y cómo podemos llamar a una función matemática. Xbase tiene fuentes que podemos descargar.

¿Tienes Prisa?

Esta tabla cotiene enlaces a los tópicos específicos.

Tópico	Sección
Ejemplo JNI	• Sobre el Ejemplo • Generar el Fichero de Cabecera • Firma del Método • Implementar el Método Nativo • Compilar las Librerías Dinámicas o de Objetos Compartidos • Ejecutar el Ejemplo

Strings, Arrays, y Fields	• Pasar Strings • Pasar Arrays • Pinning Array • Arrays de Objetos • Arrays Multi-Dimensionales • Acceder a Campos
Otros Problemas de Programación	• Problemas de Lenguaje • Métodos Llamantes • Acceder a Campos • Threads y Sincronización • Problemas de Memoria • Invocación • Adjuntar Threads

_____

[1] Cuando se usan en toda esta site, los términos, "Java virtual machine" o "JVM" significa una máquina virtual de la plataforma Java.

Ejemplos JNI

Esta sección presenta el programa de ejemplo **ReadFile**. Este ejemplo muestra cómo podemos usar JNI para invocar un método nativo que hace llamadas a funciones C para mapear en fichero en la memoria.

Sobre el Ejemplo

Podemos llamar a código escrito en cualquier lenguaje de programación desde un pograma escrito en leguaje Java declarando un método nativo Java, cargando la librería que contiene el código nativo, y luego llamando al método nativo. El código fuente de **ReadFile** que hay más abajo hace exactamente esto.

Sin embargo, el exíto en la ejecución del programa requiere uno pocos pasos adicionales más allá de la compilación del fichero fuente Java. Después de compilar, pero antes de ejecutar el ejemplo, tenemos que generar un fichero de cabecera. El código nativo implementa las definiciones de funciones contenidas en el fichero de cabecera generado y también implementa la lógica de negocio. Las siguientes sección pasan a través de estos pasos:

```
import java.util.*;

class ReadFile {
//Native method declaration
  native byte[] loadFile(String name);
//Load the library
  static {
    System.loadLibrary("nativelib");
  }

  public static void main(String args[]) {
    byte buf[];
```

```
//Create class instance
    ReadFile mappedFile=new ReadFile();
//Call native method to load ReadFile.java
    buf=mappedFile.loadFile("ReadFile.java");
//Print contents of ReadFile.java
    for(int i=0;i<buf.length;i++) {
      System.out.print((char)buf[i]);
    }
  }
}
```

Declaración del método nativo

La declaración **native** proporciona el puente para ejecutar la función nativa en una JVM[1]. En este ejemplo, la función **loadFile** se mapea a un función C llamada **Java_ReadFile_loadFile**. La implementación de la función implementa un **String** que representa un nombre de fichero y devuelve el contenido de ese fichero en un array de bytes.

```
native byte[] loadFile(String name);
```

Cargar la Librería

La librería que contiene la implementación del código nativo se carga con una llamada a **System.loadLibrary()**. Situando esta llamada en un inicializador estático nos aseguramos de que la librería sólo se cargará una vez por cada clase. La librería puede cargarse desde fuera del bloque estático si la aplicación así lo requiere. Podríamos necesitar configurar nuestro entorno para que el método **loadLibrary** pueda encontrar nuesta librería de código nativo:

```
static {
  System.loadLibrary("nativelib");
}
```

Compilar el Programa

Para compilar el program, sólo ejecutamos el comando del compilador **javac** como lo haríamos normalmente:

```
javac ReadFile.java
```

Luego, necesitamos generar un fichero de cabecera con la declaración del método nativo y la implementación del método nativo para llamar a funciones para la carga y lectura de un fichero.

Generar el Fichero de Cabecera

Para generar un fichero de cabecera, ejecutamos el comando **javah** sobre la clase **ReadFile**. En este ejemplo, el fichero de cabecera generadp se llama **ReadFile.h**. Proporciona una firma de método que debemos utilizar cuando implementemos la función nativa **loadfile**.

```
javah -jni ReadFile
```

Firma del Método

El fichero de cabecera **ReadFile.h** define el interface para mapear el método en lenguaje Java a la función nativa C. Utiliza una firma de método para mapear los argumentos y valor de retorno del método **mappedfile.loadFile** java al método nativo **loadFile** de la librería **nativelib**. Aquí está la firma del método nativo **loadFile**:

```
/*
 * Class:     ReadFile
 * Method:    loadFile
 * Signature: (Ljava/lang/String;)[B
 */
JNIEXPORT jbyteArray JNICALL Java_ReadFile_loadFile
  (JNIEnv *, jobject, jstring);
```

Los parámetros de la firma de la función son los siguientes:

- **JNIEnv ***: Un puntero al entorno JNI. Este puntero es un manejador del thread actual en la máquina virtual Java y contiene mapeos y otra información útil.
- **jobject**: Una referencia a un método que llama a este código nativo. Si el método llamante es estático, esta parámetro podría ser del tipo **jclass** en lugar de **jobject**.
- **jstring**: El parámetro suministrado al método nativo. En este ejemplo, es el nombre del fichero a leer.

Implementar el Método Nativo

En este fichero fuente nativo C, la definición de **loadFile** es una copia de la declaración C contenida en el fichero **ReadFile.h**. La definición es seguida por la implementación del método nativo. JNI proporciona mapeo por defecto tanto para C como para C++.

```
JNIEXPORT jbyteArray JNICALL Java_ReadFile_loadFile
  (JNIEnv * env, jobject jobj, jstring name) {
    caddr_t m;
```

```
    jbyteArray jb;
    jboolean iscopy;
    struct stat finfo;
    const char *mfile = (*env)->GetStringUTFChars(
                env, name, &iscopy);
    int fd = open(mfile, O_RDONLY);

    if (fd == -1) {
      printf("Could not open %s\n", mfile);
    }
    lstat(mfile, &finfo);
    m = mmap((caddr_t) 0, finfo.st_size,
                PROT_READ, MAP_PRIVATE, fd, 0);
    if (m == (caddr_t)-1) {
      printf("Could not mmap %s\n", mfile);
      return(0);
    }
    jb=(*env)->NewByteArray(env, finfo.st_size);
    (*env)->SetByteArrayRegion(env, jb, 0,
        finfo.st_size, (jbyte *)m);
    close(fd);
    (*env)->ReleaseStringUTFChars(env, name, mfile);
    return (jb);
}
```

Podemos aproximarnos a llamar a un función C existente enlugar de implementar una, de alguna de estas formas:

1. Mapear el nombre generado por JNI a un nombre de función C ya existente. La sección Problemas de Lenguaje muestra como mapear entre funciones de base de datos Xbase y código Java.
2. Usar el código Stub compartido disponible desde la página JNI en la site de java.sun.com.

Compilar la Librería Dinámica o de Objetos Compartidos

La librería necesita ser compilada como una librería dinámica o de objetos compartidos para que pueda ser cargada durante la ejecución. Las librerías o archivos estáticos son compiladas dentro de un ejecutable y no pueden ser cargadas en tiempo de ejecución. La librería dinámica para el ejemplo **loadFile** se compila de esta forma:

Gnu C/Linux:

```
gcc  -o libnativelib.so -shared -Wl,-soname,libnative.so
  -I/export/home/jdk1.2/
```

```
include -I/export/home/jdk1.2/include/linux nativelib.c
  -static -lc

Gnu C++/Linux with Xbase

g++ -o libdbmaplib.so -shared -Wl,-soname,libdbmap.so
  -I/export/home/jdk1.2/include
  -I/export/home/jdk1.2/include/linux
  dbmaplib.cc -static -lc -lxbase
```

Win32/WinNT/Win2000

```
cl -Ic:/jdk1.2/include
  -Ic:/jdk1.2/include/win32
  -LD nativelib.c -Felibnative.dll
```

Ejecutar el Ejemplo

Para ejecutar el ejemplo, la máquina virtual Java necesita poder encontrar la librería nativa. Para hacer esto, configurarmos el path de librerías al path actual de esta forma:

Unix or Linux:

```
  LD_LIBRARY_PATH=`pwd`
  export LD_LIBRARY_PATH
```

Windows NT/2000/95:

```
  set PATH=%path%;.
```

Con el path de librerías especificado de forma apropiada a nuestra plataforma, llamamos al programa como lo haríamos normalmente con el intérprete de comandos:

```
  java ReadFile
```

---

[1] Cuando se usan en toda esta site, los términos, "Java virtual machine" o "JVM" significa una máquina virtual de la plataforma Java.

Strings y Arrays

Esta sección explica cómo pasar datos string y array entre un programa escrito en Java y otros lenguajes.

Pasar Strings

El objeto **String** en el lenguaje Java, que está representado como **jstring** en JNI, es string unicode de 16 bits. En C un string por defecto está construido con caracteres de 8 bits. Por eso, para acceder a objetos **String** Java pasados a un función C ó C++ o devolver objetos un string C ó C++ a un método Java, necesitamos utilizar las funciones de conversión JNI en nuestra implementación del método nativo.

La función **GetStringUTFChar** recupera caracteres de bits desde un **jstring** de 16 bits usando el Formato de Transformación Unicode (UTF). UTF representa los caracteres Unicode como un string de 8 ó 16 bits sin perder ninguna información. El terpcer parámetro **GetStringUTFChar** es el resultado **JNI_TRUE** si se hace una copia olcar de **jstring** o **JNI_FALSE** si no se hace.

```
C Version:
  (*env)->GetStringUTFChars(env, name, iscopy)

C++ Version:
  env->GetStringUTFChars(name, iscopy)
```

La siguiente función C de JNI convierte un array de caracteres C en un **jstring**:

```
  (*env)->NewStringUTF(env, lastfile)
```

El siguiente ejemplo convierte el array de caracteres C **lastfile[80]** en un **jstring**, que es devuelto al método Java que lo llamó:

```
static char lastfile[80];

JNIEXPORT jstring JNICALL Java_ReadFile_lastFile
  (JNIEnv *env, jobject jobj) {
   return((*env)->NewStringUTF(env, lastfile));
```

```
}
```

Para permitir quela JVM[1] conozca como hemos terminado la representación UTF, llamamos a la función de conversión **ReleaseStringUTFChars** como se muestra abajo. El segundo argumento es el valor del **jstring** original usado para construir la representación UTF, y el tercer argumento es la referencia a la representación local de ese **String**.

```
(*env)->ReleaseStringUTFChars(env, name, mfile);
```

Si nuestro código nativo puede funcionar con Unicode, sin necesidar de representaciones UTF intermedias, llamamos al función **GetStringChars** para recuperar el string Unicode, y liberar la referencia con una llamada a **ReleaseStringChars**:

```
JNIEXPORT jbyteArray JNICALL Java_ReadFile_loadFile
  (JNIEnv * env, jobject jobj, jstring name) {
    caddr_t m;
    jbyteArray jb;
    struct stat finfo;
    jboolean iscopy;
    const jchar *mfile = (*env)->GetStringChars(env,
              name, &iscopy);
//...
    (*env)->ReleaseStringChars(env, name, mfile);
```

Pasar Arrays

En el ejemplo presentado en la última sección, el método nativo **loadFile** devuelve el contenido de un fichero en un array de bytes, que es un tipo primitivo del lenguaje Java. Podemos recuperar y crear tipos primitivos java llamando a la función **Type**Array apropiada.

Por ejemplo, para crear un nuevo array de floats, llamamos a **NewFloatArray**, o para crear un nuevo array de bytes, llamamos a **NewByteArray**. Este esquema de nombres se extiende para la recuperación de elementos, para añadir elementos, y para modificar elementos del array. Para obtener un nuevo array de bytes, llamamos a **GetByteArrayElements**. Para añadir o modificar elementos en el array, llamamos a **Set<type>ArrayElements**.

La función **GetByteArrayElements** afecta a todo el array. Para trabajar con un proción del array, llamamos a **GetByteArrayRegion**. Sólo hay una función **Set<type>ArrayRegion** para modificar elementos de un array. Sin embargo la región podría tener un tamaño 1, lo que sería equivalente a la no-existente **Sete<type>ArrayElements**.

Tipo de Código Nativo	Funciones usadas

jboolean	NewBooleanArray
	GetBooleanArrayElements
	GetBooleanArrayRegion/SetBooleanArrayRegion
	ReleaseBooleanArrayRegion
jbyte	NewByteArray
	GetByteArrayElements
	GetByteArrayRegion/SetByteArrayRegion
	ReleaseByteArrayRegion
jchar	NewCharArray
	GetCharArrayElements
	GetCharArrayRegion/SetCharArrayRegion
	ReleaseCharArrayRegion
jdouble	NewDoubleArray
	GetDoubleArrayElements
	GetDoubleArrayRegion/SetDoubleArrayRegion
	ReleaseDoubleArrayRegion
jfloat	NewFloatArray
	GetFloatArrayElements
	GetFloatArrayRegion/SetFloatArrayRegion
	ReleaseFloatArrayRegion
jint	NewIntArray
	GetIntArrayElements
	GetIntArrayRegion/SetIntArrayRegion
	ReleaseIntArrayRegion
jlong	NewLongArray
	GetLongArrayElements
	GetLongArrayRegion/SetLongArrayRegion
	ReleaseLongArrayRegion
jobject	NewObjectArray
	GetObjectArrayElement/SetObjectArrayElement
jshort	NewShortArray
	GetShortArrayElements
	GetShortArrayRegion/SetShortArrayRegion
	ReleaseShortArrayRegion

En el método nativo **loadFile** del ejemplo de la sección anterior, se actualiza el array entero especificando una región que tiene el tamño del fichero que está siendo leído:

```
jbyteArray jb;

jb=(*env)->NewByteArray(env, finfo.st_size);
(*env)->SetByteArrayRegion(env, jb, 0,
                finfo.st_size, (jbyte *)m);
close(fd);
```

El array es devuelto al método Java llamandte, que luego, envía al recolector de basura la referencia del array cuando ya no es utilizado. El array puede ser liberado explícitamente con la siguiente llamada:

```
(*env)-> ReleaseByteArrayElements(env, jb,
                                        (jbyte *)m, 0);
```

El último argumento de la función **ReleaseByteArrayElements** puede tener los siguientes valores:

- 0: Las actualizaciones del array desde dentro del código C serán reflejadas en la copia Java.
- **JNI_COMMIT**: La copia Java es actualizada, pero el **jbyteArray** local no es liberado.
- JNI_ABORT: Los Cambios no son copiados de vuelta, pero el **jbyteArray** es liberado. El valor usado su el array se obtiene con el mode get de **JNI_TRUE** significa que el array es una copia.

Pinning Array

Cuando recuperamos un array, podemos especificar si es una copia (**JNI_TRUE**) o una referecia del array que reside en el programa Java (**JNI_FALSE**). Si usamos una referencia al array, querremos que el array permanezca en la pila java y que no sea eliminado por el recolector de basura cuando compacte la pila de memoria. Para evitar que las referencias al array sean eliminadas, la Máquina Virtual Java "clava" el array en la memoria. Clavar el array nos asegura que cuando el array sea liberado, los elementos correctos serán actualziados en la JVM.

En el método nativo **loadfile** del ejemplo de la página anterior, el array no se liberó explícitamente. Una forma de asegurarnos de que el array es recolectado por el recolector de basura cuando ya no lo necesitamos, es llamar al método Java, pasarle el array de bytes y luego liberar la copia local del array. Esta técnica se muestra en la sección Arrays Multi-Dimensionales.

Arrays de Objetos

Podemos almacenar cualquier objeto Java enun array con llamadas a las funciones **NewObjectArray** y **SetObjectArrayElement**. La principal diferencia entre un array de objetos y un array de tipos primitivos es que cuando se construyen se usa una clase **jobjectarray** Java, como un parámetro.

El siguiente ejemplo C++ muestra cómo llamar a **NewObjectArray** para crear un array deobjetos **String**. El tamaño del array se configurará a cinco. la definición de la clase es devuelta desde una llamada a **FindClass**, y los elementos del array serán inicializados con un cadena vacía. Los elementos del array se actualizarán llamando a **SetObjectArrayElement** con la posició y el valor a poner en el array.

```
#include <jni.h>
#include "ArrayHandler.h"

JNIEXPORT jobjectArray JNICALL
             Java_ArrayHandler_returnArray
(JNIEnv *env, jobject jobj){

  jobjectArray ret;
  int i;

  char *message[5]= {"first",
      "second",
      "third",
      "fourth",
      "fifth"};

  ret= (jobjectArray)env->NewObjectArray(5,
       env->FindClass("java/lang/String"),
       env->NewStringUTF(""));

  for(i=0;i<5;i++) {
      env->SetObjectArrayElement(
              ret,i,env->NewStringUTF(message[i]));
  }
  return(ret);
}
```

La clase java que llama a este método nativo es la siguiente:

```
public class ArrayHandler {
  public native String[] returnArray();
  static{
      System.loadLibrary("nativelib");
  }

  public static void main(String args[]) {
      String ar[];
      ArrayHandler ah= new ArrayHandler();
      ar = ah.returnArray();
      for (int i=0; i<5; i++) {
```

```
            System.out.println("array element"+i+
                                "=" + ar[i]);
        }
    }
}
```

Arrays Multi-Dimensionales

Podríamos necesitar llamar a liberías numéricas y matemáticas existentes como la librería de álgebra lineal CLAPACK/LAPACK u otros programas de cálculo de matrices desde nuestro programa Java. Muchas de estas librerías y programas usando arrays de dos o más dimensiones.

En el lenguaje java, cualquier array que tenga más de una dimensión es tratado como un array de arrys. Por ejemplo, un array de enteros de dos dimensiones es manejado como un array de arrays de enteros. El array se lee horizontalmente, o también conocido como órden de fila.

Otros lenguajes como FORTRAN usan la ordenación por columnas, por eso es necesario un cuidado extra su nuestro programa maneja un array Java a una función FORTRAN. También, los elementos de un array de una aplicación Java no está garantizado que sean contiguos en la memoria. Algunas librerías usan el conocimiento de que los elementos de un array se almacenan uno junto al otro en la memoria para realizar optimizaciones de velocidad, por eso podríamos necesitar hacer una copia local del array para pasarselo a estas funciones.

El siguiente ejemplo pasad un array de dos dimensiones a un método nativo que extrae los elementos, realiza un cálculo, y llama al método Java para devolver los resultados.

El array es pasado como un objeto array que contiene un array de **jints**. Los elementos individuales se extraen primero recuperando un ejemplar de **jintArray** desde el objeto array llamando a **GetObjectArrayElement**, y luego se extraen los elementos desde la fila **jintArray**.

El ejemplo usa una matriz de tamaño fijo. Su no conocemos el tamaño del array que se está utilizando, la función **GetArrayLength(array)** devuelve el tamaño del array más exterior. Necesitaremos llamar a la función **GetArrayLength(array)** sobre cada dimensión del array para descubrir su tamaño total.

El nuevo array enviado de vuelta al programa Java está construido a la inversa. Primero, se crea un ejemplar de **jintArray** y este ejemplar se pone en el objeto array llamando a **SetObjectArrayElement**.

```
public class ArrayManipulation {
  private int arrayResults[][];
  Boolean lock=new Boolean(true);
```

```
  int arraySize=-1;

  public native void manipulateArray(
                int[][] multiplier, Boolean lock);

  static{
    System.loadLibrary("nativelib");
  }

  public void sendArrayResults(int results[][]) {
    arraySize=results.length;
    arrayResults=new int[results.length][];
    System.arraycopy(results,0,arrayResults,
                        0,arraySize);
  }

  public void displayArray() {
    for (int i=0; i<arraySize; i++) {
      for(int j=0; j <arrayResults[i].length;j++) {
        System.out.println("array element "+i+","+j+
          "= "  + arrayResults[i][j]);
      }
    }
  }

  public static void main(String args[]) {
    int[][] ar = new int[3][3];
    int count=3;
    for(int i=0;i<3;i++) {
      for(int j=0;j<3;j++) {
        ar[i][j]=count;
      }
      count++;
    }
    ArrayManipulation am= new ArrayManipulation();
    am.manipulateArray(ar, am.lock);
    am.displayArray();
  }
}

#include <jni.h>
#include <iostream.h>
#include "ArrayManipulation.h"

JNIEXPORT void
     JNICALL Java_ArrayManipulation_manipulateArray
```

```
(JNIEnv *env, jobject jobj, jobjectArray elements,
                            jobject lock){

  jobjectArray ret;
  int i,j;
  jint arraysize;
  int asize;
  jclass cls;
  jmethodID mid;
  jfieldID fid;
  long localArrayCopy[3][3];
  long localMatrix[3]={4,4,4};

  for(i=0; i<3; i++) {
     jintArray oneDim=
        (jintArray)env->GetObjectArrayElement(
                              elements, i);
     jint *element=env->GetIntArrayElements(oneDim, 0);
     for(j=0; j<3; j++) {
        localArrayCopy[i][j]= element[j];
     }
  }

// With the C++ copy of the array,
// process the array with LAPACK, BLAS, etc.

  for (i=0;i<3;i++) {
    for (j=0; j<3 ; j++) {
      localArrayCopy[i][j]=
        localArrayCopy[i][j]*localMatrix[i];
     }
  }

// Create array to send back
  jintArray row= (jintArray)env->NewIntArray(3);
  ret=(jobjectArray)env->NewObjectArray(
        3, env->GetObjectClass(row), 0);

  for(i=0;i<3;i++) {
    row= (jintArray)env->NewIntArray(3);
    env->SetIntArrayRegion((jintArray)row,(
        jsize)0,3,(jint *)localArrayCopy[i]);
    env->SetObjectArrayElement(ret,i,row);
  }

  cls=env->GetObjectClass(jobj);
```

```
  mid=env->GetMethodID(cls, "sendArrayResults",
                             "([[I)V");
  if (mid == 0) {
    cout <<"Can't find method sendArrayResults";
    return;
  }

  env->ExceptionClear();
  env->MonitorEnter(lock);
  env->CallVoidMethod(jobj, mid, ret);
  env->MonitorExit(lock);
  if(env->ExceptionOccurred()) {
    cout << "error occured copying array back" << endl;
    env->ExceptionDescribe();
    env->ExceptionClear();
  }
  fid=env->GetFieldID(cls, "arraySize",  "I");
  if (fid == 0) {
    cout <<"Can't find field arraySize";
    return;
  }
  asize=env->GetIntField(jobj,fid);
  if(!env->ExceptionOccurred()) {
    cout<< "Java array size=" << asize << endl;
  } else {
    env->ExceptionClear();
  }
  return;
}
```

_____

[1] Cuando se usan en toda esta site, los términos, "Java virtual machine" o "JVM" significa una máquina virtual de la plataforma Java.

Ozito

Otros Problemas de Programación

Esta sección presenta información sobre acceso a clases, métodos y campos, y cubre los threads, la memoria y la JVM[1].

Problemas de Lenguaje

Hasta ahora, los ejemplos de métodos nativos han cuvierto llamadas solitarias a funciones C y c++ que o devuelven un resultado o modifican los parámetro pasados a la función. Sin embargo, C++ al igual que utiliza ejemplares de clases. si creamos una clase en un método nativo, la referencia a esta clase no tiene una clase equivalente en el lenguaje Java, lo que hace díficil llamar a funciones de la clase C++ que se creó primero.

Una forma de manejar esta situación es mantener un registtro de las clases C++ referencias y pasadas de vuelta a un proxy o al programa llamante. Para asegurarnos de que una clase C++ persiste a través de llamadas a método nativos, usamos el operador **new** de C++ para crear una referencia al objeto C++ en la pila.

El siguiente código proporciona un mapeo entre la base de datos Xbase y código en lenguaje Java. La base de datos Xbase tiene un API C++ y usa inicializaciónde clases para realizar operaciones subsecuentes en la base de datos. Cuando se crea el objeto clase, se devuelve un puntero a este objeto como una valor **int** al lenguaje Java. Podemos usar un valor **long** o mayor para máquinas mayores de 32 bits.

```
public class CallDB {
  public native int initdb();
  public native short opendb(String name, int ptr);
  public native short GetFieldNo(
                        String fieldname, int ptr);

  static {
    System.loadLibrary("dbmaplib");
  }

  public static void main(String args[]) {
    String prefix=null;
```

```
    CallDB db=new CallDB();
    int res=db.initdb();
    if(args.length>=1) {
      prefix=args[0];
    }
    System.out.println(db.opendb("MYFILE.DBF", res));
    System.out.println(db.GetFieldNo("LASTNAME", res));
    System.out.println(db.GetFieldNo("FIRSTNAME", res));
   }
}
```

El valor del resultado devuelto desde la llamada al método nativo **initdb**, se pasa a las sigueintes llamadas al método nativo. El código nativo incluido en la librería **dbmaplib.cc** des-referencia el objeto Java pasado como parámetro y recupera el objeto puntero. La línea **xbDbf* Myfile=(xbDbf*)ptr;** fuerza el valor del puntero **init** a ser un punetro del tipo Xbase **xbDbf**.

```
#include <jni.h>
#include <xbase/xbase.h>
#include "CallDB.h"

JNIEXPORT jint JNICALL Java_CallDB_initdb(
        JNIEnv *env, jobject jobj) {
  xbXBase* x;
  x= new xbXBase();
  xbDbf* Myfile;
  Myfile =new xbDbf(x);
  return ((jint)Myfile);
}

JNIEXPORT jshort JNICALL Java_CallDB_opendb(
                          JNIEnv *env, jobject jobj,
                          jstring dbname, jint ptr) {
  xbDbf* Myfile=(xbDbf*)ptr;
  return((*Myfile).OpenDatabase( "MYFILE.DBF"));
}

JNIEXPORT jshort JNICALL Java_CallDB_GetFieldNo
                          (JNIEnv *env, jobject jobj,
                          jstring fieldname,
                          jint ptr) {
  xbDbf* Myfile=(xbDbf*)ptr;
  return((*Myfile).GetFieldNo(
        env->GetStringUTFChars(fieldname,0)));
}
```

Llamar a Métodos

La sección sobre los arrays iluminó algunas razones por las que llamar a método Java desde dentro de código nativo; por ejemplo, cuando necesitamos liberar el resultado que intentamos devolver. Otros usos de las llamadas a método java desde dentro de código nativo podría ser si necesitamos devolver más de un resultado o simplemente queremos modificar valores jaba desde dentro del código nativo.

Llamar a métodos Java desde dentro de código nativo implica estos tres pasos:

1. Recuperar una Referencia a la Clase.
2. Recuperar un identificador de método.
3. LLamar a los métodos.

Recuperar una Referencia de Clase

Es primer paso es recuperar una referencia a una clase que contenga los métodos a los que queremos acceder. Para recuperar una referencia, podemos usar el método **FindClass** o aceder a los argumentos **jobject** p **jclass** para el método nativo:

Usa el método FindClass:

```
JNIEXPORT void JNICALL Java_ArrayHandler_returnArray
(JNIEnv *env, jobject jobj){
jclass cls = (*env)->FindClass(env, "ClassName");
}
```

Usa el argumento jobject:

```
JNIEXPORT void JNICALL Java_ArrayHandler_returnArray
(JNIEnv *env, jobject jobj){
jclass cls=(*env)->GetObjectClass(env, jobj);
}
```

Usa el argumento jclass:

```
JNIEXPORT void JNICALL Java_ArrayHandler_returnArray
(JNIEnv *env, jclass jcls){
jclass cls=jcls;
}
```

Recuperar un identificador de Método

Una vez que hemos obtenido la clase, el segundo paso es llamar a la función **GetMethodID** para recuperar un identificador para un método que seleccionemos de la clase. El identificador es necesario cuando llamamos al método de este ejemplar de la clase. Como el lenguaje Java soporta sobrecarga de método, también necesitamos específicar la firma particular del método al que queremos llamar. Para encontar qué

firma usa nuestro método Java, ejecutamos el comando **javap** de esta forma:

```
javap -s Class
```

La firma del método usasa se muestra como un comentario después de cada declaración de método como se ve aquí:

```
bash# javap -s ArrayHandler
Compiled from ArrayHandler.java
public class ArrayHandler extends java.lang.Object {
  java.lang.String arrayResults[];
   /*   [Ljava/lang/String;   */
  static {};
   /*   ()V   */
  public ArrayHandler();
   /*   ()V   */
  public void displayArray();
   /*   ()V   */
  public static void main(java.lang.String[]);
   /*   ([Ljava/lang/String;)V   */
  public native void returnArray();
   /*   ()V   */
  public void sendArrayResults(java.lang.String[]);
   /*   ([Ljava/lang/String;)V   */
}
```

Usamos la función **GetMethodID** para llamar a métodos de ejemplar de un ejemplar del objeto. o usamos la función **GetStaticMethodID** para llamar a un método estático. Sus listas de argumentos son iguales.

Llamar a Métodos

Tercero, se llama al método de ejemplar correspndiente usando una función **Call<type>Method**. El valor **type** puede ser **Void**, **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float**, o **Double**.

Los paramétros para el método pueden pasarse como una lista separada por coma, un array de valores a la función **Call<type>MethodA**, o como una **va_list**. El **va_list** es una construccuón usada frecuentemente como lista de argumentos en C. **CallMethodV** es la función usada para pasar un **va_list ()**.

Los métodos estáticos son llamados de una forma similar excepto en que el nombre del método incluye un indenficador Satic adicional, **CallStaticByteMethodA**, y se usa el valor **jclass** en lugar del valor **jobject**.

El siguiente ejemplo devuelve un objeto array llamando al método **sendArrayResults** desde la clase **ArrayHandler**.

```
// ArrayHandler.java
public class ArrayHandler {
```

```
  private String arrayResults[];
  int arraySize=-1;

  public native void returnArray();

  static{
    System.loadLibrary("nativelib");
  }

  public void sendArrayResults(String results[]) {
    arraySize=results.length;
    arrayResults=new String[arraySize];
    System.arraycopy(results,0,
                     arrayResults,0,arraySize);
  }

  public void displayArray() {
    for (int i=0; i<arraySize; i++) {
      System.out.println("array element "+i+ "= " + arrayResults[i]);
    }
  }

  public static void main(String args[]) {
    String ar[];
    ArrayHandler ah= new ArrayHandler();
    ah.returnArray();
    ah.displayArray();
  }
}
```

El código nativo C++ se define de esta forma:

```
#include <jni.h>
#include <iostream.h>
#include "ArrayHandler.h"

JNIEXPORT void JNICALL Java_ArrayHandler_returnArray
(JNIEnv *env, jobject jobj){

  jobjectArray ret;
  int i;
  jclass cls;
  jmethodID mid;

  char *message[5]= {"first",
                "second",
                "third",
                "fourth",
```

```
                "fifth"};

    ret=(jobjectArray)env->NewObjectArray(5,
        env->FindClass("java/lang/String"),
        env->NewStringUTF(""));

    for(i=0;i<5;i++) {
      env->SetObjectArrayElement(
          ret,i,env->NewStringUTF(message[i]));
    }

    cls=env->GetObjectClass(jobj);
    mid=env->GetMethodID(cls,
          "sendArrayResults",
          "([Ljava/lang/String;)V");
    if (mid == 0) {
      cout "<<Can't find method sendArrayResults";
      return;
    }

    env->ExceptionClear();
    env->CallVoidMethod(jobj, mid, ret);
    if(env->ExceptionOccurred()) {
      cout << "error occured copying array back" <<endl;
      env->ExceptionDescribe();
      env->ExceptionClear();
    }
    return;
  }
```

Para construir esto sobre Linux, ejecutamos los siguientes comandos:

```
    javac ArrayHandler.java
    javah -jni ArrayHandler

    g++  -o libnativelib.so
          -shared -Wl,-soname,libnative.so
          -I/export/home/jdk1.2/include
          -I/export/home/jdk1.2/include/linux nativelib.cc
          -lc
```

Si queremos especificar un método de superclase, por ejemplo para llamar al constructor de padre, podemos hacerlo llamando a las funciones **CallNonvirtual<type>Method**.

Un punto importante cuando llamamos a métodos Java o a campos desde dentro del código nativo es que necesitamos capturar las excepciones lanzadas. La función **ExceptionClear** limpia cualquier excepción pendiente miesntras que la función **ExceptionOccured** chequea para ver si se ha lanzado alguna excepción en la sesión

actual JNI.

Acceder a Campos

Acceder a campos Java desde dentro de código nativo es similar a llamar a métodos Java. Sin emnargo, el campo es recuperado con un ID de campo en lugar de un ID de método.

Lo primero que necesitamos es recuperar el ID de un campo. Podemos usar la función **GetFieldID**, especificando el nombre del campo y la firma en lugar del nombre y la firma del método. Una vez que tenemos el ID del campo, llamamos a una función **Get<type>Field**. El **<type>** es el mismo tipo nativo que está siendo devuelto excepto que se quita la **j** y la primera letra se pone en mayúsculas. Por ejemplo el valor **<type>** es **Int** para el tipo nativo **jint**, y **Byte** para el tipo nativo **jbyte**.

El resultado de la función **Get<type>Field** es devuelto como el tipo nativo. Por ejemplo, para recuperar el campo **arraySize** de la clase **ArrayHandler**, llamamos a **GetIntField** como se ve en el siguiente ejemplo.

El campo puede ser seleccionado llamando a las funciones **env->SetIntField(jobj, fid, arraysize)** . Los campos estáticos pueden ser configurados llamando a **SetStaticIntField(jclass, fid, arraysize)** y recuperados llamando a **GetStaticIntField(jobj, fid)**.

```
#include <jni.h>
#include <iostream.h>
#include "ArrayHandler.h"

JNIEXPORT void JNICALL Java_ArrayHandler_returnArray
(JNIEnv *env, jobject jobj){

    jobjectArray ret;
    int i;
    jint arraysize;
    jclass cls;
    jmethodID mid;
    jfieldID fid;

    char *message[5]= {"first",
                "second",
                "third",
                "fourth",
                "fifth"};

    ret=(jobjectArray)env->NewObjectArray(5,
        env->FindClass("java/lang/String"),
        env->NewStringUTF(""));
```

```
    for(i=0;i<5;i++) {
      env->SetObjectArrayElement(
        ret,i,env->NewStringUTF(message[i]));
    }

    cls=env->GetObjectClass(jobj);
    mid=env->GetMethodID(cls,
        "sendArrayResults",
        "([Ljava/lang/String;)V");
    if (mid == 0) {
        cout <<Can't find method sendArrayResults";
        return;
    }

    env->ExceptionClear();
    env->CallVoidMethod(jobj, mid, ret);
    if(env->ExceptionOccurred()) {
       cout << "error occured copying
                        array back" << endl;
       env->ExceptionDescribe();
       env->ExceptionClear();
    }
    fid=env->GetFieldID(cls, "arraySize",  "I");
    if (fid == 0) {
        cout <<Can't find field arraySize";
        return;
    }
    arraysize=env->GetIntField(jobj, fid);
    if(!env->ExceptionOccurred()) {
       cout<< "size=" << arraysize << endl;
    } else {
       env->ExceptionClear();
    }
    return;
}
```

Threads y Sincronización

Aunque la librería nativa se carga una vez por cada clase, los threads individuales de una aplicación escrita en Java usan su propio puntero interface cuando llaman a un método nativo. Si necesitamos restringir el acceso a un objeto Java desde dentro del código nativo, podemos asegurarnos de los métodos Java a los que llamamos tienen sincronización explícita o podemos usar las funciones **MonitorEnter** y **MonitorExit**.

En el lenguaje Java, el código está protegido por un monitor siempre que especifiquemos la palabra clave **synchronized**. En Java el monitor que entra y sale de las rutinas normalmente está oculto para el desarrollador de la aplicación. En JNI, necesitamos delinear explícitamente los puntos de la entrada y de salida del código de

seguridad del thread.

El siguiente ejemplo usa un objeto **Boolean** para reestringir el acceso a la función **CallVoidMethod**.

```
env->ExceptionClear();
env->MonitorEnter(lock);
env->CallVoidMethod(jobj, mid, ret);
env->MonitorExit(lock);
if(env->ExceptionOccurred()) {
  cout << "error occured copying array back" << endl;
  env->ExceptionDescribe();
  env->ExceptionClear();
}
```

Podríamos encontrar que en caso donde queremos accder a recursos locales del sistema como un manejador MFC windows o una cola de mensajes, es mejor usar un **Thread** Java y acceder a la cola de eventos nativa o al sistema de mensajes dentro del código nativo.

Problemas de Memoria

Por defecto, JNI usa referencias locales cuando crea objetos dentro de un método nativo. Esto significa que cuando el método retorna, las referencias están disponibles para el recolector de basura. Si queremos que un objeto persista a través de las llamadas a un método nativo, debemos usar una referencia golbal. Una referencia global se crea desde una referencia local llamando a **NewGlobalReference** sobre la referencia local.

Podemos marcar explíctamente para el recolector de basura llamando a **DeleteGlobalRef** sobre la referencia. También podemos crear una referencia global al estilo weak que sea accesible desde fuera del método, pero puede ser recolectado por el recolector de basura. Para crear una de estas referencias, llamamos a **NewWeakGlobalRef** y **DeleteWeakGlobalRef** para marcar la referencia para la recolección de basura.

Incluso podemos marcar explícitamente una referencia local para la recolección de basura llamando al método **env->DeleteLocalRef(localobject)**. Esto es útil si estamo usando una gran cantidad de datos temporales:

```
static jobject stringarray=0;

JNIEXPORT void JNICALL Java_ArrayHandler_returnArray
(JNIEnv *env, jobject jobj){

  jobjectArray ret;
  int i;
  jint arraysize;
  int asize;
```

```
  jclass cls, tmpcls;
  jmethodID mid;
  jfieldID fid;

  char *message[5]= {"first",
             "second",
             "third",
             "fourth",
             "fifth"};

  ret=(jobjectArray)env->NewObjectArray(5,
      env->FindClass("java/lang/String"),
      env->NewStringUTF(""));

//Make the array available globally
  stringarray=env->NewGlobalRef(ret);

//Process array
// ...

//clear local reference when finished..
  env->DeleteLocalRef(ret);
}
```

Invocaciones

La sección sobre llamadas a métodos nos mostraba como llamar a un método o campo Java usando el interface JNI y una clase cargada usando la función **FindClass**. Con un poco más de código, podemos crear un programa que invoque a la máquina virtual Java e incluya su propio puntero al interface JNI que puede ser usado para crear ejemplares de clases Java. En Java 2, el programa de ejecución llamando **java** es una pequeña aplicación JNI que hace exactamente esto.

Podemos crear una máquina virtual Java con una llamada a **JNI_CreateJavaVM**, y desconectar la máquina virtual Java creada con una llamada a **JNI_DestroyJavaVM**. Una JVM también podría necesitar algunas propiedades adicionales de entorno. Estas propiedades podrían pasarse a la función **JNI_CreateJavaVM** en un estructura **JavaVMInitArgs**.

La estructura **JavaVMInitArgs** contiene un puntero a un valor **JavaVMOption** usado para almacenar información del entorno como el classpath y la versión de la máquina virtual Java, o propiedades del sistema que podrían pasarse normalmente en la línea de comandos del programa.

Cuando retorna la función **JNI_CreateJavaVM**, podemos llamar a método y crear ejemplares de clases usando las funciones **FindClass** y **NewObject** de la misma forma que lo haríamos con código nativo embebido.

Nota: La invocación de la máquina virtual Java sólo se usa para threads

nativos en máquinas virtuales Java. Algunas antiguas máquinas virtuales Java tienen una opción de threads verdes que es estable para el uso de invocaciones, Sobre una plataforma Unix, podríamos necesitar enlazar explícitamente con **-lthread** o **-lpthread**.

El siguiente programa invoca una máquina virtual Java, carga la clase **ArrayHandler** y recupera el campo **arraySize** que debería tener el valor menos uno. Las opciones de la máquina virtual Java incluyen el path actual en el classpath y desactivar del compilador Just-In_Time (JIT) **-Djava.compiler=NONE**.

```
#include <jni.h>

void main(int argc, char *argv[], char **envp) {
  JavaVMOption options[2];
  JavaVMInitArgs vm_args;
  JavaVM *jvm;
  JNIEnv *env;
  long result;
  jmethodID mid;
  jfieldID fid;
  jobject jobj;
  jclass cls;
  int i, asize;

  options[0].optionString = ".";
  options[1].optionString = "-Djava.compiler=NONE";

  vm_args.version = JNI_VERSION_1_2;
  vm_args.options = options;
  vm_args.nOptions = 2;
  vm_args.ignoreUnrecognized = JNI_FALSE;

  result = JNI_CreateJavaVM(
             &jvm,(void **)&env, &vm_args);
  if(result == JNI_ERR ) {
    printf("Error invoking the JVM");
    exit (-1);
  }

  cls = (*env)->FindClass(env,"ArrayHandler");
  if( cls == NULL ) {
    printf("can't find class ArrayHandler\n");
    exit (-1);
  }
  (*env)->ExceptionClear(env);
  mid=(*env)->GetMethodID(env, cls, "<init>", "()V");
  jobj=(*env)->NewObject(env, cls, mid);
```

```
  fid=(*env)->GetFieldID(env, cls, "arraySize", "I");
  asize=(*env)->GetIntField(env, jobj, fid);

  printf("size of array is %d",asize);
  (*jvm)->DestroyJavaVM(jvm);
}
```

Adjuntar Threads

Después de invocar la máquina virtual Java, hay un thread local ejecutándose en ella. Podemos crear más threads en el sistema operativo local y adjuntar threads en la máquina virtual Java para estos nuevos threads. Podriamos querer hacer esto su nuestra aplicación nativa es multi-threads.

Adjuntamos el thread local a la máquina virtual Java con una llamada a **AttachCurrentThread**. Necesitamos suministrar punteros al ejemplar de la máquina virtual Java y al entorno JNI. En la plataforma Java 2, podemos específicar en el tercer parámetro el nombre del thread y/o el grupo bajo el que queremos que viva nuestro thread. Es importante eliminar cualquier thread que haya sido préviamente adjuntado; de otra forma, el programa no saldrá cuando llamemos a **DestroyJavaVM**.

```
#include <jni.h>
#include <pthread.h>

JavaVM *jvm;

void *native_thread(void *arg) {
  JNIEnv *env;
  jclass cls;
  jmethodID mid;
  jfieldID fid;
  jint result;
  jobject jobj;
  JavaVMAttachArgs args;
  jint asize;

  args.version= JNI_VERSION_1_2;
  args.name="user";
  args.group=NULL;
  result=(*jvm)->AttachCurrentThread(
        jvm, (void **)&env, &args);

  cls = (*env)->FindClass(env,"ArrayHandler");
  if( cls == NULL ) {
    printf("can't find class ArrayHandler\n");
    exit (-1);
  }
  (*env)->ExceptionClear(env);
```

```
  mid=(*env)->GetMethodID(env, cls, "<init>", "()V");
  jobj=(*env)->NewObject(env, cls, mid);
  fid=(*env)->GetFieldID(env, cls, "arraySize", "I");
  asize=(*env)->GetIntField(env, jobj, fid);
  printf("size of array is %d\n",asize);
  (*jvm)->DetachCurrentThread(jvm);
}

void main(int argc, char *argv[], char **envp) {
  JavaVMOption *options;
  JavaVMInitArgs vm_args;
  JNIEnv *env;
  jint result;
  pthread_t tid;
  int thr_id;
  int i;

  options = (void *)malloc(3 * sizeof(JavaVMOption));

  options[0].optionString = "-Djava.class.path=.";
  options[1].optionString = "-Djava.compiler=NONE";

  vm_args.version = JNI_VERSION_1_2;
  vm_args.options = options;
  vm_args.nOptions = 2;
  vm_args.ignoreUnrecognized = JNI_FALSE;

  result = JNI_CreateJavaVM(&jvm,(void **)&env, &vm_args);
  if(result == JNI_ERR ) {
    printf("Error invoking the JVM");
    exit (-1);
  }

  thr_id=pthread_create(&tid, NULL, native_thread, NULL);

// If you don't have join, sleep instead
//sleep(1000);
  pthread_join(tid, NULL);
  (*jvm)->DestroyJavaVM(jvm);
  exit(0);
}
```

[1] Cuando se usan en toda esta site, los términos, "Java virtual machine" o "JVM" significa una máquina virtual de la plataforma Java.

Ozito

Proyecto Swing: Construir un Interface de Usuario

Las arquitecturas Java Foundation Classes (JFC) y JavaBeans Enterprise comparten un elemento de diseño clave: la separación de los datos de su aspecto en pantalla o la manipulación de los datos. En las aplicaciones JavaBeans Enterprise, el beande entidad proporciona una vista de los datos. El mecanismo de los datos oculto puede ser solapado y modificado sin modificar la vista del bean de entidad o recompilar cualquier código que use la vista.

El proyecto Swing separa la vista y control de un componente visual de sus contenidos, o medelo de datos. Sin embargo, aqunque el Proyecto Swing tiene los componentes que crean la arquitectura Modelo-Vista-Controlador (MVC), es más seguro describirlo como una arquitectura de modelo-delegado. Esteo eso por la parte controlador de un interface Swing, frecuentemente usa el eventos del ratón y de teclado para responder al componente, es combinada con la vista física en un objeto "User Interface delegate" (UI delegate).

Cada componente, por ejemplo un **JButton** o un **JScrollBar**, tiene una clase UI delegate separada que desciende desde la clase **ComponentUI** y está bajo el control de un controlador UI separado. Mientras que cada componente tiene un UI delgate básico, no está más unido con los datos ocultos por lo que se pueden intercambiar mientras que la aplicación todavía se está ejecutando. La posibilidad de cambiar el aspecto y comportamiento refleja la característica del aspecto y comportamiento conectable (PLAF) disponible en Swing.

Este capítulo describe componentes de usuario Swing en términos de la aplicación **AuctionClient**.

¿Tienes Prisa?

Esta tabla contiene enlaces directos a los tópicos específicos.

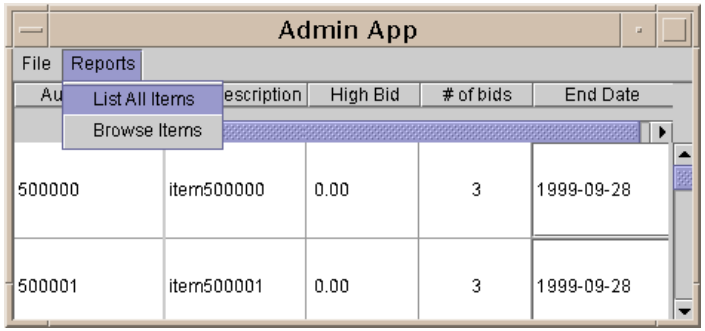
Tópicos	Sección

Componentes y Modelos de Datos	• Componentes de Peso Ligero • Ordenación de Componentes • Modelos de Datos • Dibujado Personalizado de Celdas • Edición de Celdas Personalizadas • Manejo de Eventos Especializado • Direcciones del Proyecto Swing

---

Componentes y Modelos de Datos

El programa AuctionClient es una sencilla aplicacion GUI que permite a los administradores de la casa de subastas listar y navegar por los ítems de la subasta, e imprime informes sobre estos ítems. Esta sección describe el código Swing de la aplicación que utiliza componentes de peso ligero y otras características Swing:



Componentes de Peso Ligero

Todos los componentes Swing, excepto **JApplet**, **JDialog**, **JFrame** y **JWindow** son componentes de peso ligero. Los componentes de peso ligero, al contrario que sus contraparte del AWT, no dependen del toolkit local del sistema.

Por ejemplo, un componente pesado **java.awt.Button** ejecutándose sobre la plataforma Java para Unix mapea el botón Motif real. En esta relación es botón Motif es llamado "par" del **java.awt.Button**. Si hemos creado dos **java.awt.Button** en una aplicación, también se crearán dos "pares" y dos botones Motif. La plataforma Java comunicac con los botones Motif usando el JNI. Para cada componente añadido a la aplicación, hay una pila adicional unida al sistema de ventanas local, que es por lo que estos componentes se llaman de peso pesado.

Los componentes de peso ligero no tiene "pares" y emulan a los componentes del sistema local de ventanas. Un botón de peso ligero está representado por un rectángulo con una etiqueta dentro que acepta eventos del ratón. Añadir más botones significa dibujar más rectángulos.

Un componente de peso ligero necesita dibujarse obre algo, y una aplicación escrita en Java necesita interactuar con el controlador de ventanas local para que la ventana principal de la aplicación pueda ser cerrada o minimizada. Esto es porque los componentes padres de nivel superior mencionados arriba (**JFrame**, **JApplet**, y otros) están implementado como componentes de peso pesado -- necesitan ser mapeados sobre un componente en el sistema local de ventanas.

Un **JButton** es una forma muy sencilla de dibujar. Para componentes más complejos, como **JList** o **JTable**, los elementos o celdas de la lista o la tabla son dibujadas por un objeto **CellRenderer**. Un objeto **CellRenderer** proporciona flexibilidad porque hace posible que cualquier tipo de objeto pueda ser mostrado en cualquier fila o columna.

Por ejemplo, un **JTable** puede usar un **CellRenderer** diferente para cada columna. Este segmento de código selscciona la segunda columna, que está referenciada como índice **1**, para usar un objeto **CustomRenderer** para crear las celdas de esa columna.

```
JTable scrollTable=new JTable(rm);
TableColumnModel scrollColumnModel =
              scrollTable.getColumnModel();
CustomRenderer custom = new CustomRenderer();
scrollColumnModel.getColumn(1).setCellRenderer(custom);
```

Ordenar Componente

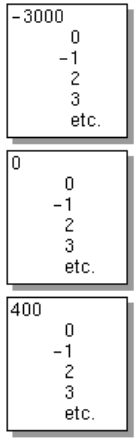
Cada aplicación o applet Swing necesita al menos un componente contenedor de peso pesado (un **JFrame**, **JWindow**, **JApplet**, o **JDialog**). Cada uno de estos componentes con la contraparte de **JFrame**: **JInternalFrame**, contiene un componente llamado **RootPane**. El **JRootPane** controla la adición de capas adicionales usadas en dicho contenedor como **JLayeredPane**, **JContentPane**, **GlassPane** y la opcional**JMenuBar**. También les permite a todos los componentes emulados (de peso ligero) interactuar con la cola de eventos AWT para enviar y recibir eventos. Al interactuar con la cola de eventos, todos los componentes emulados obteinen una interacción indirecta con el controlador de ventanas local.

JLayeredPane

El **JLayeredPane** se sitúa sobre el **JRootPane**, y como su nombre indica, controla las capas del componente contenidas dentro de los límites del contenedor de peso pesado. Los componentes no son añadidos al **JLayeredPane**, sino al **JContentPane**. El **JLayeredPane** determina el orden Z de los componentes del **JRootPane**. Se puede pensar en el orden Z como el orden de solapamiento de varios componentes. Si arrastramos y soltamos un componente o solicitamos un diálogo desplegable, queremos que el componente aparezca encima de todas las otras ventana de la aplicación. El **JLayeredPane** nos permite poner los componentes en capas.

El **JLayeredPane** divide la profundidad del contenedor en diferentes bandas que pueden usarsr para asignarle a un componente un tipo de nivel apropiado. La banda **DRAG_LAYER** , valor 400, aparece sobre todas las demás capas. El nivel más ingerior de **JLayeredpane**, la banda **DEFAULT_FRAME_LAYER**, tiene valor -3000 y y es el nivel de los contenedores de peso pesado, incluyendo el **MenuBar**. Las bandas son las siguientes:

Valor	**Nombre de Banda**	**Tipos de Componentes**
-3000	**DEFAULT_FRAME_LAYER**	**JMenubar**
0	**DEFAULT_LAYER**	**JButton, JTable, ..**
	PALETTE_LAYER	Componentes flotantes como un **JToolBar**
	MODAL_LAYER	Diálogos Modales
400	**DRAG_LAYER**	FONT FACE="Verdana, Arial, Helvetica, sans-serif">Arrastrar y Soltar sobre todas las capas



de estas bandas de profundidad generales, los componentes peuden estár organizados con un sistema de ordenación para ordenar los componentes dentro de una banda particular, pero este sistema invierte la prioridad de los números. Por ejemplo, en una banda especificada como **DEFAULT_LAYER**, los componentes con un valor, aparecen delante de los otros componentes de la banda; mientras, componentes con un número mayor o -1 aparecen por detrás de él. El número más alto en es esque de numeración es .1, por eso una forma de visualizarlo es un vector de componentes que pasa a través de dibujar primero los componentes con un número mayor terminando con el componente en la posición 0.

Por ejemplo, el siguiente código añade un **JButton** a la capa por defecto y especifica que aparezca encima de los otros componentes de esa misma capa:

```
JButton enterButton = new JButton("Enter");
layeredPane.add(enterButton,
                JLayeredPane.Default_Layer, 0);
```

Podemos conseguir el mismo efecto llamando al método **LayeredPane.moveToFont** dentro de una capa o usando el método **LayeredPane.setLayer** método para moverlo a una capa diferente.

JContentPane

El **JContentPane** controla la adición de componentes a los contenedores de peso pesado. Por eso, tenemos que llamar al método **getContentPane** para añadir un componente al **ContentPane** del **RootPane**. Por defecto, un **ContentPane** se inicializa con un controlador de distribución **BorderLayout**. Hay dos formas de cambiar el controlador de distribución. Podemos llamar al método **setLayout** de esta forma:

```
getContentPane()).setLayout(new BoxLayout())
```

O podemos reemplazar el **ContentPane** por defecto con nuestro propio **ContentPane**, como un **JPanel**, como este:

```
JPanel pane= new JPanel();
pane.setLayout(new BoxLayout());
setContentPane(pane);
```

GlassPane

El **GlassPane** normalmente es completamente transparente y solo actúa como una hoja de cristal delante de los componentes. Podemos implementar nuestro propio **GlassPane** usando un componente como **JPanel** e instalándolo como el **GlassPane** llamando al método **setGlassPane**. El **RootPane** se configura con un **GlassPane** que puede ser recuperado llamando a **getGlassPane**.

Una forma de usar un **GlassPane** es para implementar un componente que de forma invisble maneje todos los eventos de teclado y de ratón, bloqueando efectivamente la entrada del usuario hasta que se complete un evento. El **GlassPane** puede bloquear los eventos, pero realmente el cursor no volverá a su estado por defecto si tenermos seleccionar el cursor para que sea un cursor ocupado en el **GlassPane**. Se requiere un evento de ratón adicional para el refresco:

```
MyGlassPane  glassPane = new MyGlassPane();
setGlassPane(glassPane);
setGlassPane.setVisible(true); //before worker thread
..
setGlassPane.setVisible(false); //after worker thread

private class MyGlassPane extends JPanel {

  public MyGlassPane() {
    addKeyListener(new KeyAdapter() { });
    addMouseListener(new MouseAdapter() { });
    super.setCursor(
      Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
  }
```

```
}
```

Modelos de Datos

Se han combinado numerosos modelos de capas para formar las tablas del GUI **AuctionClient**. A un nivel fundacional, el interface **TableModel** y sus dos implementaciones **AbstractTableModel** y **DefaultTableModel** propocionan las opciones más básicas para almacenar, recupear y modificar los datos básicos.

El **TableModel** es responsable de definir y categorizar los datos por sus clases. También determina si el dato puede ser editado y cómo se agrupan los datos en columnas y filas. Sin embargo, es importante observar que mientras el interface **TableModel** se usa más frecuentemente en la construcción de un **JTable**, no está unido fundamentalmente a su apariencia en pantalla. Las implementaciones podría fácilmente formar la parte básica de la hoja de cálculo, o incluiso una clase no-GUI que pida la organización de los datos de una forma tabular.

La clase **ResultsModel** es el corazón de las tablas **AuctionClient**. Define una hoja de datos dimámica, dicta qué usuarios de la clase pueden editar los datos a través del método **ResultsModel.isCellEditable**, y proporciona el método **update** para mantener los datos actualizados. El modelo es la base de la tablas fijas y escrollables, y deja que las modificaciones se reflejen en cada vista.

A un alto nivel, y representado unca capa intermedia entre los datos y su representación gráfica esta el **TableColumnModel**. En este nivel los datos son agrupados por columnas en anticipación de su aparición gráfica en la tabla. La visibilidad y tamaño de las columnas, sus cabeceras, y los tipos de componentes de sus renderizadores de celdas y editores son todos manejados por la clase **TableColumnModel**.

Por ejemplo, congelar la columna más ala izquierda del GUI **AuctionClient** es posible porque los datos de la columna sin fácilmente intercambiables entre múltiples objetos **TableColumnModel** y **JTable**. Esto traduce los objetos **fixedTable** y **scrollTable** del programa **AuctionClient**.

Más alto tadavía se unen los distintos rederizadores, editores y componentes de cabecera cuya combinación define el aspecto y organización del componente **JTable**. Este nivel es onde se tomas las decisiones fundamentales sobre la distribución del **JTable**.

La creacción de las clases internas **CustomRenderer** y **CustomButtonRenderer** dentro de la aplicación **AuctionClient** permite a los usuarios de esas clases redefinir los componentes sobre los que se basa la apariencia de las celdas de la tabla. De igual forma, la clase **CustomButtonEditor** toma el lugar del editor por defecto de la tabla. De una forma verdaderamente orientada a ojetos, los editores por defecto y renderizadores son fácilmente reemplazados si afectar a los datos que ellos representan ni la función del componente en el que residen.

Finalmente, los distintos interfaces de los componente de usuario son responsavles de la apariencia última de la **JTable**. Esta es la representación específica del aspecto y comportamiento de las tablas **AuctionClient** y sus datos de una forma final. El resultado final es que añadir una parte final Swing a unos servicios existentes requiere muy código adicional. De hecho, la codificación del modelo es una de las tareas más sencillas al construir una aplicación Swing.

Modelo de la Tabla

La clase **JTable** tiene asociada una clase **DefaultTableModel** que internamente usa un vector para almacenar datos. Los datos de cada fila son almacenados en un objeto Vector **singl** mientras que otro objeto **Vector** almacena cada una de esas columnas y los elementos que las constituyen. El objeto **DefaultTableModel** puede ser inicializado con datos de varias formas diferentes. Este código muesta el **DefaultTableModel** creado con un array de dos dimensiones y un segundo array que representa las cabeceras de columnas. El **DefaultTableModel** convierte el array de **Object** en los objetos **Vector** apropidos:

```
Object[][] data = new Object[][]{ {"row 1 col1",
                                   "Row 1 col2" },
                                   {"row 2 col 1",
                                   "row 2 col 2"}
                                 };
Object[] headers = new Object[] {"first header",
                                  "second header"};
DefaultTableModel model = new DefaultTableModel(data,
                                 headers);

table = new JTable(model);
table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
```

Crear un modelo de tabla personalizado es tan cercano y sencillo como usar **DefaultTableModel**, y requiere muy poca codificación adicional. Podemos implementar un modelo de tabla implementando un método que devuelva el número de entradas del modelo, y un método que recupere un elemento en un posición específica de ese modelo. Por ejemplo, el modelo **JTable** puede ser implementado desde **javax.swing.table.AbstractTableModel** mediante la implementación de los métodos **getColumnCount**, **getRowCount** y **getValueAt** como se ve aquí:

```
final Object[][] data = new Object[][]{ {
                        "row 1 col1",
                        "row 1 col2" },
                        {"row 2 col 1",
                        "row 2 col 2"} };
final Object[] headers = new Object[] {
                        "first header",
                        "second header"};

TableModel model = new AbstractTableModel(){
  public int getColumnCount() {
```

```
      return data[0].length;
    }
    public int getRowCount() {
      return data.length;
    }
    public String getColumnName(int col) {
      return (String)headers[col];
    }

    public Object getValueAt(int row,int col) {
      return data[row][col];
    }
  };
  table = new JTable(model);
  table.setAutoResizeMode(
        JTable.AUTO_RESIZE_OFF);
```

Esta tabla es de sólo lectura y los valores de sus datos ya son conocidos. De hecho, incluso los datos son declarados **final** para que peudan ser recuperados por la clase interna **TableModel**. Esta no es la situación normal cuando trabajamos con datos vivos.

Podemos crear una tabla editable añadiendo el método de verificación **isCellEditable**, que es usado por el editor de celda por defecto, y el método **AbstractTableModel** para configurar un valor en una posición. Hasta este cambio, el **AbstractTableModel** ha estado manejando el redibujado y el redimensionado de la tabla disparando distintos eventos de cambio de tabla. como el **AbtractTableModel** no conoce nada de lo ocurrido a los datos de la tabla, necesitamos informarle llamando al método **fireTableCellUpdated**. Las siguientes líneas han añadido la clase interna **AbstractTableModel** para permitir la edición de los datos:

```
  public void setValueAt (Object value,
                          int row, int col) {
    data[row][col] = value;
    fireTableCellUpdated (row, col);
  }

  public boolean isCellEditable(int row,
                          int col) {
    return true;
  }
```

Más Modelos de Tablas

Un requerimiento común para mostrar datos tabulares es la inclusión de un columna no desplazable. Este columna proporcina una conjunto de datos anclados que permanecen estacionarios y visibles mientras que sus columnas vecinas son desplazadas horizontalmente (y frecuentemente fuera de la vista). Esto es importante en casos donde la fila de datos puede ser identificada por un único valor en la columna fijada, como un nombre o número identificador. el siguiente código de ejemplo usa una columna de tabla fijada para mostrar una lista de ítems de la subasta.

El modelo de tabla base de este ejemplo implementa la clase **AbstractTableModel**. Su método **update** rellena dinámicamente los datos de la tabla desde una llamada a la base de datos. Envían un evento de la tabla ha sido actualizada llamando al método **fireTableStructureChanged** para indicar el número de filas o columnas de la tabla que se han modificado.

```
package auction;

import javax.swing.table.AbstractTableModel;
import javax.swing.event.TableModelEvent;
import java.text.NumberFormat;
import java.util.*;
import java.awt.*;

public class ResultsModel extends AbstractTableModel{
  String[]  columnNames={};
  Vector rows = new Vector();

  public String getColumnName(int column) {
    if (columnNames[column] != null) {
      return columnNames[column];
    } else {
      return "";
    }
  }

  public boolean isCellEditable(int row, int column){
```

```
    return false;
  }

  public int getColumnCount() {
    return columnNames.length;
  }

  public int getRowCount() {
    return rows.size();
  }

  public Object getValueAt(int row, int column){
    Vector tmprow = (Vector)rows.elementAt(row);
    return tmprow.elementAt(column);
  }

  public  void update(Enumeration enum) {
    try {
      columnNames = new String[5];
      columnNames[0]=new String("Auction Id #");
      columnNames[1]=new String("Description");
      columnNames[2]=new String("High Bid");
      columnNames[3]=new String("# of bids");
      columnNames[4]=new String("End Date");
      while((enum !=null) &&
                (enum.hasMoreElements())) {
        while(enum.hasMoreElements()) {
          AuctionItem auctionItem=(
                AuctionItem)enum.nextElement();
          Vector items=new Vector();
          items.addElement(new Integer(
                auctionItem.getId()));
          items.addElement(
                auctionItem.getSummary());
          int bidcount= auctionItem.getBidCount();
          if(bidcount >0) {
            items.addElement(
                NumberFormat.getCurrencyInstance().
                format(auctionItem.getHighBid()));
          } else {
            items.addElement("-");
          }
          items.addElement(new Integer(bidcount));
          items.addElement(auctionItem.getEndDate());
          rows.addElement(items);
        }
      }

      fireTableStructureChanged();
    } catch (Exception e) {
      System.out.println("Exception e"+e);
    }
  }
}
```

La tabla es creada desde el modelo **ResultsModel**, Luego se elimina la primera columna de la tabla y se añade a una nueva tabla. Como ahora tenemos dos tablas, la única forma de que las selecciones estén sincronizadas es usar un objeto **ListSelectionModel** para configurar la selección sobre la fila de la tabla en la sotras tablas que no fueron seleccionadas llamando al método **setRowSelectionInterval**.

El ejemplo completo lo podemos encontrar en el ficheo fuente AuctionClient.java:

```
  private void listAllItems() throws IOException{
    ResultsModel rm=new ResultsModel();
    if (!standaloneMode) {
      try {
         BidderHome bhome=(BidderHome)
```

```
            ctx.lookup("bidder");
      Bidder bid=bhome.create();
      Enumeration enum=
            (Enumeration)bid.getItemList();
      if (enum != null) {
        rm.update(enum);
      }
    } catch (Exception e) {
      System.out.println(
            "AuctionServlet <list>:"+e);
    }
} else {
  TestData td= new TestData();
  rm.update(td.results());
}
scrollTable=new JTable(rm);
adjustColumnWidth(scrollTable.getColumn(
    "End Date"), 150);
adjustColumnWidth(scrollTable.getColumn(
    "Description"), 120);
scrollColumnModel = scrollTable.getColumnModel();
fixedColumnModel = new DefaultTableColumnModel();

TableColumn col = scrollColumnModel.getColumn(0);
scrollColumnModel.removeColumn(col);
fixedColumnModel.addColumn(col);

fixedTable = new JTable(rm,fixedColumnModel);
fixedTable.setRowHeight(scrollTable.getRowHeight());
headers = new JViewport();

ListSelectionModel fixedSelection =
            fixedTable.getSelectionModel();
fixedSelection.addListSelectionListener(
            new ListSelectionListener() {
  public void valueChanged(ListSelectionEvent e) {
    ListSelectionModel lsm = (
       ListSelectionModel)e.getSource();
    if (!lsm.isSelectionEmpty())  {
       setScrollableRow();
    }
  }
 });

 ListSelectionModel scrollSelection =
            scrollTable.getSelectionModel();
 scrollSelection.addListSelectionListener(
            new ListSelectionListener() {
   public void valueChanged(ListSelectionEvent e) {
     ListSelectionModel lsm =
            (ListSelectionModel)e.getSource();
     if (!lsm.isSelectionEmpty())  {
       setFixedRow();
     }
   }
 });

 CustomRenderer custom = new CustomRenderer();
 custom.setHorizontalAlignment(JLabel.CENTER);
 scrollColumnModel.getColumn(2).setCellRenderer(
            custom);
 scrollColumnModel.getColumn(3).setCellRenderer(
            new CustomButtonRenderer());

 CustomButtonEditor customEdit=new
```

```
                    CustomButtonEditor(frame);
scrollColumnModel.getColumn(3).setCellEditor(
            customEdit);

headers.add(scrollTable.getTableHeader());

JPanel topPanel = new JPanel();
topPanel.setLayout(new BoxLayout(topPanel,
            BoxLayout.X_AXIS));
adjustColumnWidth(
            fixedColumnModel.getColumn(0), 100);

JTableHeader fixedHeader=
                fixedTable.getTableHeader();
fixedHeader.setAlignmentY(Component.TOP_ALIGNMENT);
topPanel.add(fixedHeader);
topPanel.add(Box.createRigidArea(
                    new Dimension(2, 0)));
topPanel.setPreferredSize(new Dimension(400, 40));

JPanel headerPanel = new JPanel();
headerPanel.setAlignmentY(Component.TOP_ALIGNMENT);
headerPanel.setLayout(new BorderLayout());

JScrollPane scrollpane = new JScrollPane();
scrollBar = scrollpane.getHorizontalScrollBar();

headerPanel.add(headers, "North");
headerPanel.add(scrollBar, "South");
topPanel.add(headerPanel);

scrollTable.setPreferredScrollableViewportSize(
            new Dimension(300,180));
fixedTable.setPreferredScrollableViewportSize(
            new Dimension(100,180));
fixedTable.setPreferredSize(
            new Dimension(100,180));

innerPort = new JViewport();
innerPort.setView(scrollTable);
scrollpane.setViewport(innerPort);

scrollBar.getModel().addChangeListener(
            new ChangeListener()  {
  public void stateChanged(ChangeEvent e) {
    Point q = headers.getViewPosition();
    Point p = innerPort.getViewPosition();
    int val = scrollBar.getModel().getValue();
    p.x = val;
    q.x = val;
    headers.setViewPosition(p);
    headers.repaint(headers.getViewRect());
    innerPort.setViewPosition(p);
    innerPort.repaint(innerPort.getViewRect());
  }
});

scrollTable.getTableHeader(
              ).setUpdateTableInRealTime(
              false);

JPanel bottomPanel = new JPanel();
bottomPanel.setLayout(new BoxLayout(
            bottomPanel, BoxLayout.X_AXIS));
fixedTable.setAlignmentY(Component.TOP_ALIGNMENT);
```

```
    bottomPanel.add(fixedTable);
    bottomPanel.add(Box.createRigidArea(
                          new Dimension(2, 0)));
    innerPort.setAlignmentY(Component.TOP_ALIGNMENT);
    bottomPanel.add(innerPort);
    bottomPanel.add(Box.createRigidArea(
                          new Dimension(2, 0)));

    scrollPane= new JScrollPane(bottomPanel,
              JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
              JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
    JViewport outerPort = new JViewport();
    outerPort.add(bottomPanel);
    scrollPane.setColumnHeaderView(topPanel);
    scrollPane.setViewport(outerPort);

    scrollTable.setAutoResizeMode(
                  JTable.AUTO_RESIZE_OFF);
    frame.getContentPane().add(scrollPane);

    scrollTable.validate();
    frame.setSize(450,200);
  }

  void setFixedRow() {
    int index=scrollTable.getSelectedRow();
    fixedTable.setRowSelectionInterval(index, index);
  }

  void setScrollableRow() {
    int index=fixedTable.getSelectedRow();
    scrollTable.setRowSelectionInterval(index, index);
  }

  void adjustColumnWidth(TableColumn c, int size) {
    c.setPreferredWidth(size);
    c.setMaxWidth(size);
    c.setMinWidth(size);
  }
```

Modelo JList

El componente **JList** muestra una lista verticla de datos y usa un **ListModel** para contener y manipular los datos. También usa un objeto **ListSelectionModel** para permitir la selección y subsecuente recuperación de elementos de la lista.

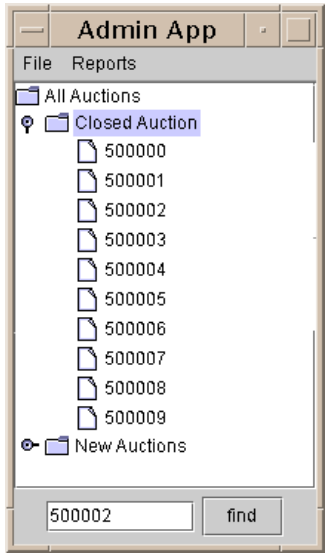
Las implementaciones por defecto de las clases **AbstractListModel** y **AbstractListSelectionModel** las proporciona el API Swing desde las clases **DefaultListModel** y **DefaultListSelectionModel**. Si usamos estos dos modelos por defecto y el renderizador de celdas por defecto, obtendremos una lista que muestra elementos modelo llamado al método **toString** sobre cada objeto. La lista usa el modelo **MULTIPLE_INTERVAL_SELECTION** de selección de lista para seleccionar cada elemento de la lista.

Hay disponibles tres modos de selección para **DefaultListSelectionModel**: **SINGLE_SELECTION**, donde sólo se puede seleccionar un ítem a la vez; **SINGLE_INTERVAL_SELECTION** en el que se puede seleccionar un rango de items secuenciales; y **MULTIPLE_INTERVAL_SELECTION**, en el que se permite que cualquir o todos los elementos sean seleccionados. El modo de selección puede cambiarse llamando al método **setSelectionMode** de clase **JList**.

```
  public SimpleList() {
     JList list;
     DefaultListModel deflist;
     deflist= new DefaultListModel();
     deflist.addElement("element 1");
     deflist.addElement("element 2");
     list = new JList(deflist);

     JScrollPane scroll = new JScrollPane(list);
     getContentPane().add(scroll, BorderLayout.CENTER);
   }
```

Modelo JTree

La clase **JTree** modela y muestra una lista vertical de elementos o nodos ordenados en una forma de árbol de herencia.



Un objeto **JTree** teine un nodo raíz y uno o más nodos hijos, que pueden contener más nodos hijos. Cada nodo padre puede expandirse para mostrar sus hijos de forma similar a los familiares árboles de directorios de los usuarios de Windows.

Como los componentes **JList** y **JTable**, el **JTree** consta de más de un modelo. El modo de selección es similar al detallado para el modelo **JList**. El modo de selección tiene estás ligeras diferencias en los nombres: **SINGLE_TREE_SELECTION**, **DISCONTIGUOUS_TREE_SELECTION**, y **CONTIGUOUS_TREE_SELECTION**.

Mientras que **DefaultTreeModel** mantiene los datos en un árbol y es responsable de añadir y eliminar nodos, es la clase **DefaultTreeMutableTreeNode** la que define los métodos usados para moverse por los nodos. El **DefaultTreeModel** se usa frecuentemente para implementar modelos personaloizados porque no hay un **AbstractTreeModel** en el paquete **JTree**. Sin embargo, si usamos objetos personalizados, debemos implementar **TreeModel**. Este código de ejemplo crea un **JTree** usando el **DefaultTreeModel**.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class SimpleTree extends JFrame {
  public SimpleTree() {
    String[] treelabels =   {
                             "All Auctions",
                             "Closed Auction",
                             "Open Auctions"};
    Integer[] closedItems = { new Integer(500144),
                              new Integer(500146),
                              new Integer(500147)};

    Integer[] openItems = { new Integer(500148),
                            new Integer(500149)};

    DefaultMutableTreeNode[] nodes = new
        DefaultMutableTreeNode[treelabels.length];
    DefaultMutableTreeNode[] closednodes = new
        DefaultMutableTreeNode[closedItems.length];
    DefaultMutableTreeNode[] opennodes = new
        DefaultMutableTreeNode[openItems.length];

    for (int i=0; i < treelabels.length; i++) {
      nodes[i] = new
        DefaultMutableTreeNode(treelabels[i]);
    }
    nodes[0].add(nodes[1]);
    nodes[0].add(nodes[2]);

    for (int i=0; i < closedItems.length; i++) {
       closednodes[i] = new
                DefaultMutableTreeNode(closedItems[i]);
       nodes[1].add(closednodes[i]);
    }

    for (int i=0; i < openItems.length; i++) {
       opennodes[i] = new
                DefaultMutableTreeNode(openItems[i]);
       nodes[2].add(opennodes[i]);
    }
    DefaultTreeModel model=new
                DefaultTreeModel(nodes[0]);

    JTree tree = new JTree(model);

    JScrollPane scroll = new JScrollPane(tree);
    getContentPane().add(scroll, BorderLayout.CENTER);
  }
```

```
  public static void main(String[] args) {
    SimpleTree frame = new SimpleTree();
    frame.addWindowListener( new WindowAdapter() {
      public void windowClosing( WindowEvent e ) {
        System.exit(0);
      }
    });
    frame.setVisible(true);
    frame.pack();
    frame.setSize(150,150);
  }
}
```

El método **toString** se usa para recuperar el valor de los objetos **Integer** en ek árbol. Y aunque se usa **DefaultTreeModel** para mantener los datos en el árbol y para añadir y eliminar nodos, la clase **DefaultMutableTreeNode** define los métodos usados para moverse a través de los nodos de un árbol.

con el método **depthFirstEnumeration** se consigue una búsqueda de nodos dentro de un **JTree**, que es el mismo que el método **postorderEnumeration** desde el punto final hasta el primer árbol. O podemos llamar al método **preorderEnumeration**, el inverso del método **postorderEnumeration**, que empieza desde la raíz y desciende cada rama por orden. O podemos llamar al método **breadthFirstEnumeration**, que empieza en la raíx y visita todos los nodos hijos en un nivel nates de visitar los nodos hijos de una profundidad inferior.

El siguiente código de ejemplo expande el nodo padre si conteine un nodo hijo que corresponda con el campo de búsqueda introducido. Usa una llamada a **Enumeration e = nodes[0].depthFirstEnumeration();** para devolver la lista de todos los nodos del árbol. Una vez que ha encontrado una correspondencia, construye el **TreePath** desde el nodo raíz hacia el nodo que concuerda con la cadena búsqueda pasada a **makeVisible** de la clase **JTree** que se asegura de que nodo se expandirá en el árbol.

```
import java.awt.*;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class SimpleSearchTree extends JFrame {
  JPanel findPanel;
  JTextField findField;
  JTree tree;
  JButton findButton;
  DefaultMutableTreeNode[] nodes;

  public SimpleSearchTree() {
    String[] treelabels =  { "All Auctions",
                             "Closed Auction",
                             "Open Auctions" };
    Integer[] closedItems = { new Integer(500144),
                              new Integer(500146),
                              new Integer(500147) };

    Integer[] openItems ={ new Integer(500148),
                           new Integer(500149)};

    nodes = new
      DefaultMutableTreeNode[treelabels.length];
    DefaultMutableTreeNode[] closednodes = new
            DefaultMutableTreeNode[closedItems.length];
    DefaultMutableTreeNode[] opennodes = new
            DefaultMutableTreeNode[openItems.length];
    for (int i=0; i < treelabels.length; i++) {
       nodes[i] = new
         DefaultMutableTreeNode(treelabels[i]);
    }
    nodes[0].add(nodes[1]);
    nodes[0].add(nodes[2]);

    for (int i=0; i < closedItems.length; i++) {
       closednodes[i] = new
                DefaultMutableTreeNode(closedItems[i]);
       nodes[1].add(closednodes[i]);
    }
```

```
    for (int i=0; i < openItems.length; i++) {
       opennodes[i] = new DefaultMutableTreeNode(
                           openItems[i]);
       nodes[2].add(opennodes[i]);
    }

    DefaultTreeModel model=new
                       DefaultTreeModel(nodes[0]);
    tree = new JTree(model);

    JScrollPane scroll = new JScrollPane(tree);
    getContentPane().add(scroll, BorderLayout.CENTER);
    findPanel= new JPanel();
    findField= new JTextField(10);
    findButton= new JButton("find");
    findButton.addActionListener (new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        String field=findField.getText();
        if (field != null) {
          findNode(findField.getText());
        } else {
          return;
        }
      }
    });
    findPanel.add(findField);
    findPanel.add(findButton);
    getContentPane().add(findPanel, BorderLayout.SOUTH);
  }
  public void findNode(String field) {
    Enumeration e = nodes[0].depthFirstEnumeration();
    Object currNode;
    while (e.hasMoreElements()) {
      currNode = e.nextElement();
      if (currNode.toString().equals(field)) {
        TreePath path=new TreePath(((
             DefaultMutableTreeNode)currNode).getPath());
        tree.makeVisible(path);
        tree.setSelectionRow(tree.getRowForPath(path));
        return;
      }
    }
  }

  public static void main(String[] args) {
    SimpleSearchTree frame = new SimpleSearchTree();
    frame.addWindowListener( new WindowAdapter() {
      public void windowClosing( WindowEvent e ) {
        System.exit(0);
      }
    });
    frame.setVisible(true);
    frame.pack();
    frame.setSize(300,150);
  }
}
```

JTree, **JTable** y **JList** probablemente son los modelos más comunes que querremos personalizar. Pero podemos usar modelos como **SingleSelectionModel** para manipulación de datos en general. Esta clase nos permite especificar como se seleccionan los datos en un componente.

Dibujo de Celdas Personalizado

Como hemos aprendido arriba, muchos componentes tienen un renderizador de celdas por defecto para dibujar cada elemento de la tabla, árbol o lista. El renderizador de celdas por defecto normalmente es un **JLabel** y muestra una representación **String** de los datos del elemento.

Un sencillo renderizador de celda personalizado puede extender la clase **DefaultXXXCellRenderer** para proporcionar personalización adicional en el **getXXXCellRenderer**. Los componentes **DefaultTableCellRenderer** y **DefaultTreeCellRenderer** usan un **JLabel** para dibujar la celda. Esto significa que cualquier personalización que pueda ser aplicada a un

JLabel también puede ser usada en una celda de **JTable** o de **JTree**.

Por ejemplo, el siguiente renderizador selecciona el color del fondo del componente si el ítem de la subasta ha recibido un alto número de pujas:

```
class CustomRenderer extends DefaultTableCellRenderer {
  public Component getTableCellRendererComponent(
                    JTable table,Object value,
                    boolean isSelected,
                    boolean hasFocus,
                    int row, int column) {

      Component comp =
                  super.getTableCellRendererComponent(
                    table,value,isSelected,hasFocus,
                    row,column);

      JLabel label = (JLabel)comp;

      if(((Integer)value).intValue() >= 30) {
          label.setIcon(new ImageIcon("Hot.gif"));
      } else {
          label.setIcon(new ImageIcon("Normal.gif"));
      }

      return label;
  }
}
```

El renderizador se selecciona sobre una columna de esta forma:

```
  CustomRenderer custom = new CustomRenderer();
  custom.setHorizontalAlignment(JLabel.CENTER);
  scrollColumnModel.getColumn(2).setCellRenderer(
                                          custom);
```

Si el componente que está siendo mostrado dentro de la columna **JTable** requiere más funcionalidad que la disponible usando un **JLabel**, podemos crear nuestro propio **TableCellRenderer**. Este código de ejemplo usa un **JButton** como renderizador de celdas:

```
class CustomButtonRenderer extends JButton
                implements TableCellRenderer {
   public CustomButtonRenderer() {
     setOpaque(true);
   }

   public Component getTableCellRendererComponent(
                     JTable table, Object value,
                     boolean isSelected,
                     boolean hasFocus, int row,
                     int column) {

      if (isSelected) {
         ((JButton)value).setForeground(
                table.getSelectionForeground());
         ((JButton)value).setBackground(
                table.getSelectionBackground());
      } else {
         ((JButton)value).setForeground(table.getForeground());
         ((JButton)value).setBackground(table.getBackground());
      }
      return (JButton)value;
    }
}
```

Al igual que el renderizador de celdas por defecto **JLabel**, esta clase trata con el componente principal (en este caso **JButton**) para hacer el dibujado. La selección de la celda cambia los colores del botón. Como antes, el renderizador de celdas está seguro sobre la columna apropiada de la tabla de subastas con el método **setCellRenderer**:

```
   scrollColumnModel.getColumn(3).setCellRenderer(
```

```
        new CustomButtonRenderer());
```

De forma alternativa, todos los componentes **JButton** pueden configurarse para usar el **CustomButtonRenderer** en la tabla con una llamada a **setDefaultRenderer** de esta forma:

```
table.setDefaultRenderer(
      JButton.class, new CustomButtonRenderer());
```

Editor de Celdas Personalizado

De la misma forma que podemos configurar como se dibujan las celdas en una **JTable** o en un **JTree**, también podemos configurar como una celda editable responde a la ediciones. Una diferencia entre usar editores y renderizadores de cledas es que hay un **DefaultCellEditor** para todos los componentes, pero no hay un **DefaultTableCellEditor** para celdas de tablas.

Mientras existen renderizadores separados para **JTree** y **JTable**, una sóla clase **DefaultCellEditor** implementa los dos interfaces **TableCellEditor** y **TreeCellEditor**. Sin embargo, la clase **DefaultCellEditor** sólo tiene constructores para los componentes **JComboBox**, **JCheckBox**, y **JTextField**. La clase **JButton** no se mapea con tinguno de estos constructores por eso se crea un **JCheckBox** inútil para satisfacer los requerimientos de la clase **DefaultCellEditor**.

El siguiente ejemplo usa un editor de botón personalizado que muestra el número de días que quedan de subasta cuando se hacer doble click sobre él. El doble click para disparar la acción se especifica seleccionando el valor **clickCountToStart** a dos. Una copia exacta del método **getTableCellEditorComponent** dibuja el botón en modo edición. Un componente **JDialog** que muestra el número de días que quedan aparecerá cuando se llame al método **getCellEditorValue**. El valor del número de días que quedan se calcula moviendo la fecha del calendario actual hasta la fecha final. La clase **Calendar** no tiene un método que exprese una diferencia entre dos fechas distinto a los milisegundos que haya entre esas dos fechas.

```
class CustomButtonEditor extends DefaultCellEditor {
  final JButton mybutton;
  JFrame frame;

  CustomButtonEditor(JFrame frame) {
    super(new JCheckBox());
    mybutton = new JButton();
    this.editorComponent = mybutton;
    this.clickCountToStart = 2;
    this.frame=frame;
    mybutton.setOpaque(true);
    mybutton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        fireEditingStopped();
      }
    });
  }

  protected void fireEditingStopped() {
    super.fireEditingStopped();
  }

  public Object getCellEditorValue() {
    JDialog jd= new JDialog(frame, "Time left");
    Calendar today=Calendar.getInstance();
    Calendar end=Calendar.getInstance();
    SimpleDateFormat in=new SimpleDateFormat("yyyy-MM-dd");
    try {
      end.setTime(in.parse(mybutton.getText()));
    } catch (Exception e){
      System.out.println("Error in date"+mybutton.getText()+e);
    }
    int days = 0;
    while(today.before(end)) {
    today.roll(Calendar.DATE,true);
    days++;
    }
    jd.setSize(200,100);
    if (today.after(end)) {
       jd.getContentPane().add(new JLabel("Auction completed"));
    } else {
      jd.getContentPane().add(new JLabel("Days left="+days));
    }
    jd.setVisible(true);
     return new String(mybutton.getText());
```

```
    }

    public Component getTableCellEditorComponent(JTable table,
                  Object value, boolean isSelected,
                  int row, int column) {

      ((JButton) editorComponent).setText(((
                  JButton)value).getText());
      if (isSelected) {
        ((JButton) editorComponent).setForeground(
                  table.getSelectionForeground());
        ((JButton) editorComponent).setBackground(
                  table.getSelectionBackground());
      } else {
        ((JButton) editorComponent).setForeground(
                  table.getForeground());
        ((JButton) editorComponent).setBackground(
                  table.getBackground());
      }
      return editorComponent;
      }
  }
```

Manejo de Eventos Especializados

Swing usa las clases de manejo de eventos disponibles en el API AWT desde el JDK 1.1. Sin embargo, algunos APIs nuevos están disponibles en la clase **SwingUtilities** que se usan para añadir más control sobre la cola de eventos. Los dos nuevos métodos manejadores de eventos son **invokeLater** y **invokeAndWait**. Este último espera a que el evento sea procesador en la cola de eventos.

Estos métodos se usan frecuentemente para solicitar el foco sobre un componente después de que otro evento haya ocurrido y que podría afectar al foco de componentes. Podemos devolver el foco llamando al método **invokeLater** y pasando un **Thread**:

```
JButton button =new JButton();
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    button.requestFocus();
  }
});
```

Direcciones Swing

Mientras que la arquitectura básica Swing ha permanecido estable a su diseño original, se han realizado muchas mejoras y optimizaciones sobre componentes como **JTable** y en áreas desplazables.

Sin embargo, como veremos en la sección Analizar un Programa, una simple tabla de 700x300 requiere casi medio megabyte de memoria cuando se usa doble buffer. La creacción de 10 tablas probablemente necesitaría el intercambio de memoria a disco, afectando severamenta al rendimiento en máquinas de bajo nivel.

El API de Impresión

El paquete **java.awt.print** de la plataforma Java 2 nos permite imprimir cualquier cosa que pueda ser renderizada a un contexto **Graphics** o **Graphics2D** — incluyendo componentes AWT, componentes Swing y gráficos 2D. El API de impresión es fácil de usar. Nuestra aplicación le dice al sistema de impresión qué imprimir, y el sistema de impresión determina cuando se renderiza cada página. Este *modelo de impresión por retrollamada* permite soporte de impresión en un amplio rango de impresoras y sistemas. El modelo de retrollamada también permite al usuario imprimir a una impresora de mapa de bits desde un ordenador que no tiene suficiente memoria o espacio en disc para contener el bitmap de una página completa.

Un contexto gráfico permite a un programa dibujar en un dispositivo de renderización como una pantalla, una impresora o una imagen fuera de pantalla. Como los componentes Swing se renderizan a través de un objeto **Graphics** usando el soporte de gráficos AWT, es fácil imprimir componentes Swing con el nuevo API de impresión. Sin embargo, los componentes AWT no se renderizan a un dispositivo gráfico, debemos extender la clase del componente AWT e implementar el método de dibujo del componente AWT.

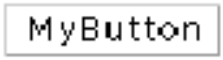
¿Qué hay en el Paquete?

El **java.awt.print** contiene los siguientes interfaces, clases y excepciones. Aquí podrás encontrar la Especificación del API.

- Interfaces
  - Pageable
  - Printable
  - PrinterGraphics
- Clases
  - Book
  - PageFormat
  - Paper

    - PrinterJob
- Excepciones
    - PrinterAbortException
    - PrinterException
    - PrinterIOException

Imprimir un Componente AWT



La aplicación printbutton.java muestra un panel con un *MyButton* sobre él. Cuando se pulsa el botón, la aplicación imprime el componente *MyButton*.

En el código, la clase Button se extiende para implementar **Printable** e incluye los método **paint** y **print**. Este último es necesario porque la clase implementa **Printable**, y el método **paint** es necesario porque describe como aparecen la forma del botón y la etiqueta de texto cuando se imprimen.

Para ver el botón, la contexto gráfico de impresión es trasladado a un área imaginable de la impresora, y para ver la etiqueta de texto, se selecciona una fuente en el contexto gráfico de impresión.

En este ejemplo, el botón se imprime a 164/72 pulgadas dentro del margen imaginable (hay 72 pixels por pulgada) y a 5/72 pulgadas del margen superior imaginado. Aquí es donde el botón es posicionado por el controlador de distribución y estos mismo número son devultos por las siguientes llamadas:

```
int X = (int)this.getLocation().getX();
int Y = (int)this.getLocation().getY();
```

Y aquí está el código de la clase **MyButton**:

```
class MyButton extends Button
                 implements Printable {

  public MyButton() {
    super("MyButton");
  }

  public void paint(Graphics g) {
  //To see the label text, you must specify a font for
  //the printer graphics context
    Font  f = new Font("Monospaced", Font.PLAIN,12);
    g2.setFont (f);

  //Using "g" render anything you want.
```

```
//Get the button's location, width, and height
  int X = (int)this.getLocation().getX();
  int Y = (int)this.getLocation().getY();
  int W = (int)this.getSize().getWidth();
  int H = (int)this.getSize().getHeight();

//Draw the button shape
  g.drawRect(X, Y, W, H);

//Draw the button label
//For simplicity code to center the label inside the
//button shape is replaced by integer offset values
  g.drawString(this.getLabel(), X+10, Y+15);

}

public int print(Graphics g,
                   PageFormat pf, int pi)
                     throws PrinterException {
  if (pi >= 1) {
    return Printable.NO_SUCH_PAGE;
  }

  Graphics2D g2 = (Graphics2D) g;

//To see the button on the printed page, you
//must translate the printer graphics context
//into the imageable area
  g2.translate(pf.getImageableX(), pf.getImageableY());
  g2.setColor(Color.black);
  paint(g2);
  return Printable.PAGE_EXISTS;
 }
```

Nota: La impresión Graphics2D está basada en la clase **BufferedImage** y algunas plataformas no permiten un color de fondo negro por defecto. Si este es nuestro caso tenemos que añadir **g2.setColor(Color.black)** al método **print** antes de la invocación a **paint**.

Imprimir un Componente Swing

Imprimir un componente Swing es casi lo mismo que imprimir un componente AWT, excepto que la clase **MyButton** no necesita una implementación del método **paint**. Sin embargo, si teine un método **print** que llama al método **paint** del componente. La implementación del método **paint** no es necesaria porque los componentes Swing

saben como dibujarse a sí mismos.

Aquí está el código fuente completo para la versión Swing de printbutton.java.

```
class MyButton extends JButton implements Printable {

  public MyButton() {
    super("MyButton");
  }

  public int print(Graphics g,
                   PageFormat pf, int pi)
                     throws PrinterException {
    if (pi >= 1) {
      return Printable.NO_SUCH_PAGE;
    }

    Graphics2D g2 = (Graphics2D) g;
    g2.translate(pf.getImageableX(),
                 pf.getImageableY());
    Font  f = new Font("Monospaced", Font.PLAIN,12);
    g2.setFont (f);
    paint(g2);
    return Printable.PAGE_EXISTS;
   }
```

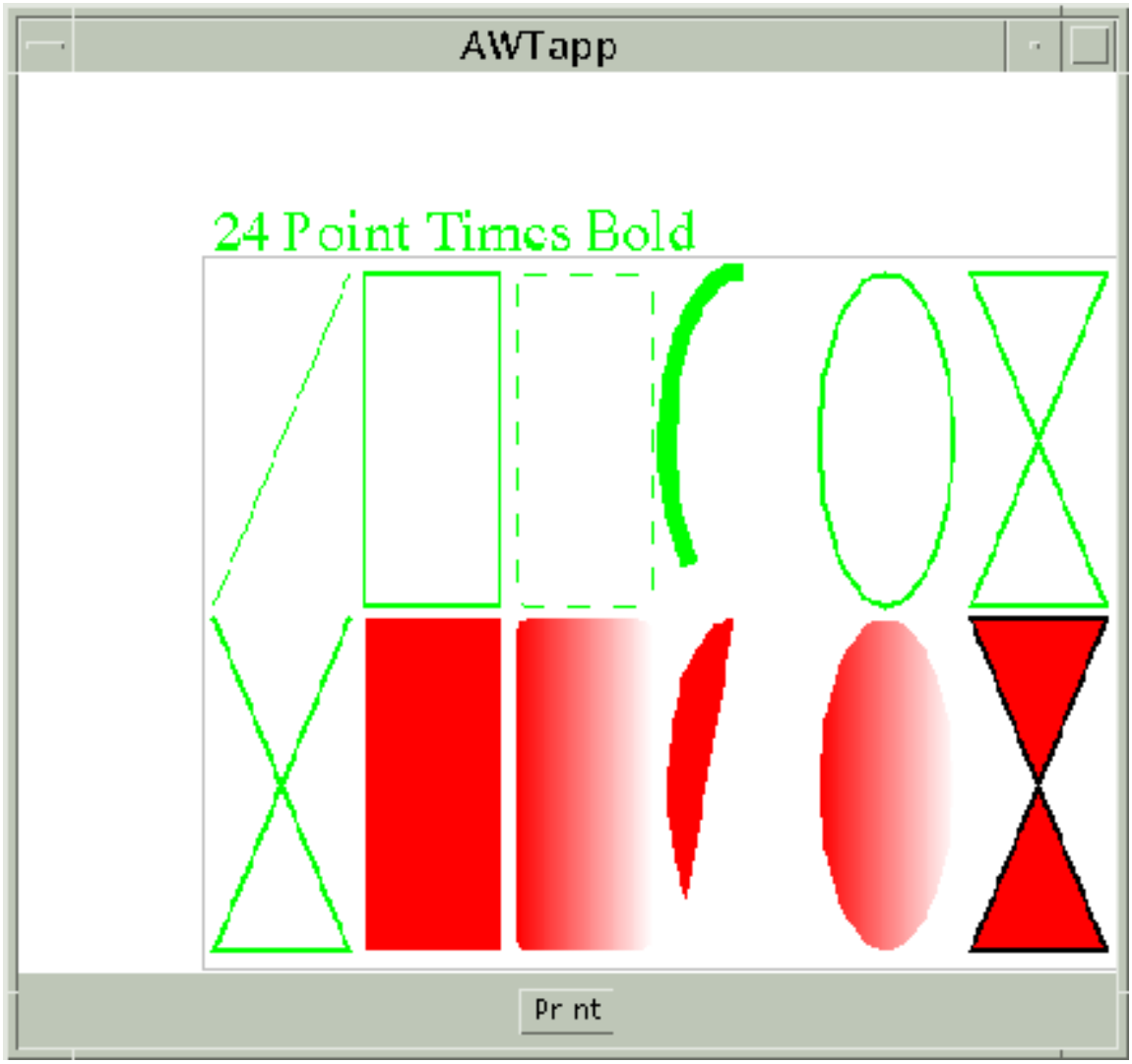
Si extendemos un **JPanel** e implementamos **Printable**, podemos imprimir un componente panel y todos sus componentes.

```
public class printpanel extends JPanel
                        implements ActionListener,
                        Printable {
```

Aquí está el código de printpanel.java que imprime un objeto **JPanel** y el **JButton** que contiene, y el código de ComponentPrinterFrame.java que imprime un ojeto **JFrame** y los componentes **JButton**, **JList**, **JCheckBox**, y **JComboBox** que contiene.

Imprimir Gráficos en Swing

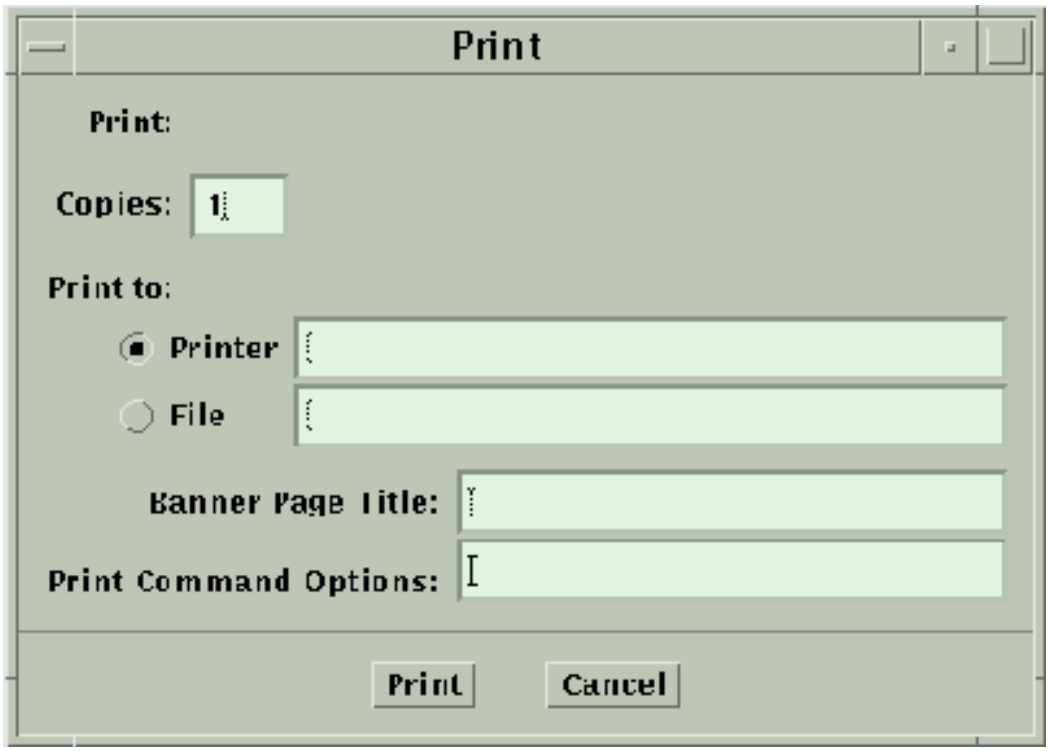
De la misma forma que el ejemplo AWT extiende un componente **Button** e implementa el método **paint** para dibujar un botón, podemos subclasificar componentes AWT y Swing e implementar el método **paint** para renderizar gráficos 2D en la pantalla o en la impresora. La aplicación Swing ShapesPrint.java muestra como se hace esto.



El método **paintComponent** llama al método **drawShapes** para renderizar gráficos 2D en la pantalla cuando arranca la aplicación. Cuando pulsamos sobre el botón, **Print**, se crea un contexto gráfico de impresión y es pasado al método **drawShapes** para el dibujado.

Diálogo de Impresión

Es fácil mostrar el Diálogo de Impresión para que el usuario final pueda intercambiar las propiedades del rabajo de impresión. El método **actionPerformed** del ejemplo Swing anterior modificado aquí hace justo esto:




```
public void actionPerformed(ActionEvent e) {
  PrinterJob printJob = PrinterJob.getPrinterJob();
  printJob.setPrintable((MyButton) e.getSource());
  if(printJob.printDialog()){
    try { printJob.print(); }
    catch (Exception PrinterExeption) { }
  }
}
```

Nota: En Swing, la sentencia **printJob.setPageable((MyButton) e.getSource());** puede escribirse como **printJob.setPrintable((MyButton) e.getSource());**. La diferencia es que **setPrintable** es para aplicaciones que no conocen el número de páginas que están imprimiendo. Si usamos **setPrintable**, necesitamos añadir **if(pi >= 1){ return Printable.NO_SUCH_PAGE: }** al principio del método **print**.

Diálogo de configuración de Página

Podemos añadir una línea de código que le dice al objeto **PrinterJob** que mueste el Diálogo de Configuración de Página para que el usuario final pueda modificar interactivamente el formato de la página para imprimir en vertical u horizontal, etc. El método **actionPerformed** ejemplo Swing acnterior está mostrado aquí para que muestre los diálogos de Impresión y Configuración de Página:

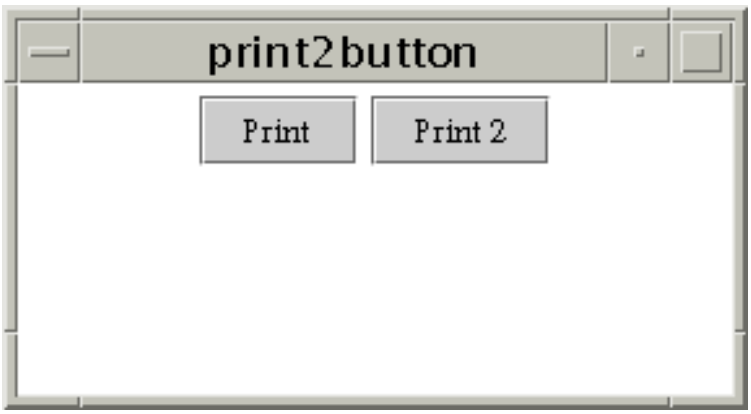
Nota: Algunas plataformas no soportan el diálogo de configuración de página. En estas plataformas, la llamada a **pageDialog** simplemente devuelven el objeto **PageFormat** que se les pasó y no muestran ningún

diálogo.

```
public void actionPerformed(ActionEvent e) {
  PrinterJob printJob = PrinterJob.getPrinterJob();
  printJob.setPrintable((MyButton) e.getSource());
  PageFormat pf = printJob.pageDialog(
                              printJob.defaultPage());
  if(printJob.printDialog()){
    try { printJob.print(); } catch (Exception ex) { }
  }
}
```

Imprimir una Colección de Páginas

Podemos usar la clase **Book** para imprimir una colección de páginas que añadimos al libro. Esta páginas pueden estár en cualquier orden y tener diferentes formatos.



El ejemplo print2button.java pone los botones **Print** y **Print 2** del tipo **MyButton** en un panel. Crea un libro que contiene las páginas para imprimir. Cuando pulsamos algun botón, el libro imprime una copia del botón **Print** en modo horizontal y dos copias del botón **Print 2** en modo vertical, como se especifica en la implementación del método **actionPerformed** mostrada aquí:

Nota: Actualmente un Bug restringe a la plataforma Solaris a imprimir sólo en vertical.

```
public void actionPerformed(ActionEvent e) {
  PrinterJob printJob = PrinterJob.getPrinterJob();

/* Set up Book */
  PageFormat landscape = printJob.defaultPage();
  PageFormat portrait = printJob.defaultPage();
  landscape.setOrientation(PageFormat.LANDSCAPE);
  portrait.setOrientation(PageFormat.PORTRAIT);
  Book bk = new Book();
  bk.append((Printable)b, landscape);
  bk.append((Printable)b2, portrait, 2);
  printJob.setPageable(bk);
```

```
    try { printJob.print(); } catch (Exception ex) { }
  }
```

Ozito

Impresión Avanzada

La sección anterior explicó cómo imprimir componentes sencillos y cubría las técnicas que se pueden usar para imprimir capturas de pantalla. Sin embargo, si queremos imprimir más que un componente por cada página, o su nuestro componentes es mayor que el tamaño de una página, necesitamos hacer algún trabajo adicional dentro del método **print**. Esta sección explica qué necesitamos hacer y concluye con un ejemplo de cómo imprimir los contenidos de un componente **JTable**.

Varios Componentes por Página

Hay veces cuando imprimimos un componente en una página que no se cubre las necesidades de impresión que queremos. Por ejemplo, podríamos queren incluir una cabecera o un pie en cada página de impresión con un número de página -- algo que no es necesario mostrar en la pantalla.

Desafortunadamente, imprimir múltiples componentes sobre una página no es tán sencillo como añadir llamadas a **paint** porque cada llamada sobreescribe la salida de la llamada anterior.

La clave para imprimir más de un componente en un página, es usar los métodos **translate(double, double)** y **setClip** de la clase **Graphics2D**.

El método **translate** mueve un lápiz imaginario a la siguiente posición de la salida de impresión donde el componente puede ser dibujado y luego imprimido. Hay dos métodos **translate** en la clase **Graphics2D**. Para imprimir múltiples componentes necesitamos el que toma dos argumentos **double** porque este método permite posiconamiento relativo. Debemos asegurarnos de forzar cualquier valor entero a double o float. El posicionamiento relativo en este contexto significa que las llamadas anteriores a **translate** son tenidas en cuenta cuando se calqula el nuevo punto de traslado.

El método **setClip** se usa para restringir que el componente sea pintado, y por lo tanto, imprimido, en el área especificada. Esto nos permite imprimir múltiples componentes en una página moviendo el lápiz imaginario a diferentes puntos de la página y luego pintándo cada componente en el área recortada.

Ejemplo

Podemos reemplazar el método **print** de los ejemplos **printbutton.java** Abstract Window Toolkit (AWT) y Swing con el siguiente código para añadir un mensaje en el pie de página de *Company Confidential*.

```
public int print(Graphics g, PageFormat pf, int pi)
                throws PrinterException {

    if (pi >= 1) {
       return Printable.NO_SUCH_PAGE;
    }

    Graphics2D g2 = (Graphics2D) g;
    Font f= Font.getFont("Courier");
    double height=pf.getImageableHeight();
    double width=pf.getImageableWidth();

    g2.translate(pf.getImageableX(),
                 pf.getImageableY());
    g2.setColor(Color.black);
    g2.drawString("Company Confidential", (int)width/2,
        (int)height-g2.getFontMetrics().getHeight());
    g2.translate(0f,0f);
    g2.setClip(0,0,(int)width,
        (int)(height-g2.getFontMetrics().getHeight()*2));
    paint (g2);
    return Printable.PAGE_EXISTS;
}
```

En el nuevo método **print**, el contexto **Graphics2D** es recortado antes de llamar al método **paint** del padre **JButton**. Esto evita que el método **JButton paint** sobreescriba el botón de la página. El método **translate** se usa para apuntan el método **JButton paint** a que empieza el **paint** con un desplazamiento de 0,0 desde la parte visible de la página. el área visible ya está calculada mediante una llamada anterior a **translate**:

```
g2.translate(pf.getImageableX(), pf.getImageableY());
```

Para más componentes, podríamos necesitar configurar el color de fondo para ver los resultados. En este ejemplo el color de texto se imprimió en negro.

Métodos Útiles para Llamar en el Método print

Los siguientes métodos son útiles para calcular el número de páginas requeridas y para hacer que un componente se reduzca hasta entrar en una página:

Métodos PageFormat:

getImageableHeight()
devuelve la altura de la página que podemos usar para imprimir la salida.

getImageableWidth()
devuelve la anchura de la página que podemos usar para imprimir la salida.

Método Graphics2D:

scale(xratio, yratio)
escala el conexto gráfico 2D a este tamaño. Un ratío de uno mantiene el tamaño, menos de uno reduce el tamaño del contexto gráfico.

Componentes Mayores de una Página

El API de impresión de Java " tiene un API **Book** que proporciona el concepto de páginas. Sin embargo, este API sólo añade objetos printables a una colecciónde objetos printables. No calcula las rupturas de página ni expande componentes sobre múltiples páginas

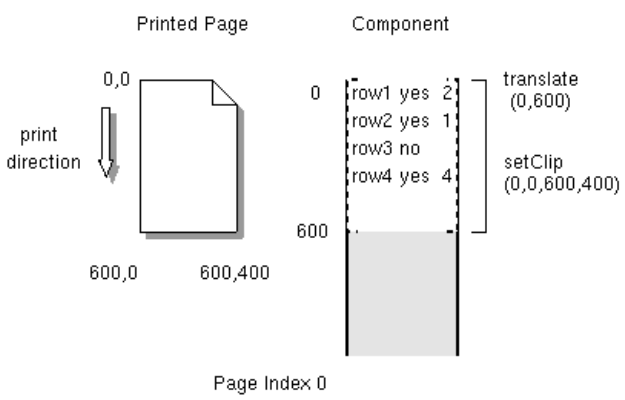
Cuando imprimimos un sólo componente en una página, sólo tenemos que chequear que el valor del índice es mayor o igual que uno y devolver **NO_SUCH_PAGE** cuando se alcanza este valor.

Para imprimir multiples páginas, tenemos que calcular el número de páginas necesarias para contener el componente. Podemos calcular el número total de páginas necesarias dividiendo el espacio ocupado por el componente por el valor devuelto por el método **getImageableHeight**. Una vez calculado el número total de páginas, podemos ejecutar el siguiente chequeo dentro del método **print**:

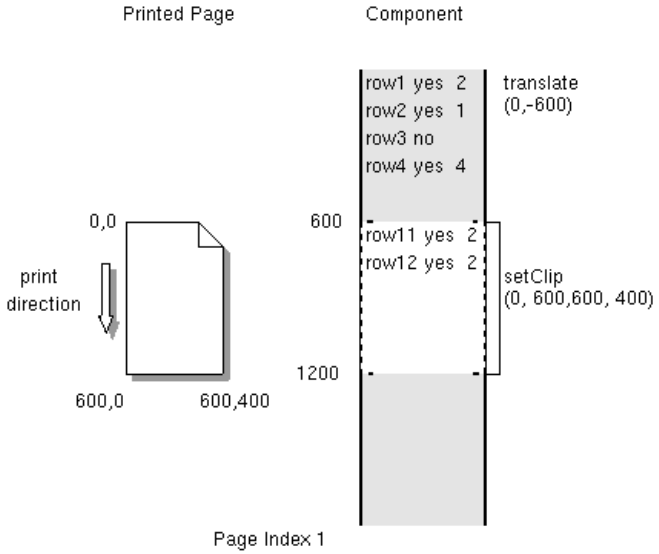
```
if (pageIndex >=TotalPages) {
      return NO_SUCH_PAGE;
}
```

El marco de trabajo de impresión llama al método **print** multiples veces hasta que **pageIndex** sea menor o igual que **TotalPages**. Todo lo que necesitamos hacer es crear una nueva página para del mismo componente encada bucle **print**. Esto se puede hacer tratando la página impresa como una ventana deslizante sobre el componente. La parte del componente que se está imprimiendo es seleccionada por una llamada a **translate** para marcar la parte superior de la página y una llama a **setClip** para marcar la parte inferior de la página. el siguiente diagrama ilustra este proceso.

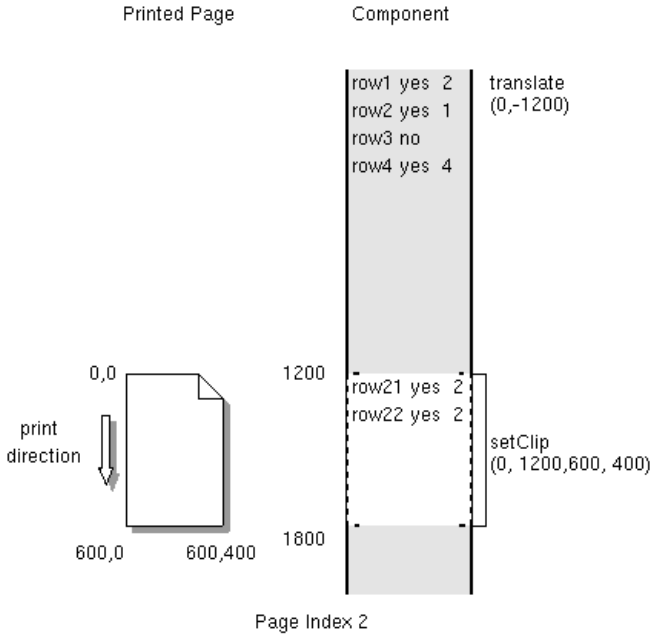
El lado izquierdo del diagrama representa la página enviada a la impresora. El lado LEFT contiene la longitud del componente que está siendo imprimido en el método **print**. La primera página puede ser representada de esta forma:


Printed Page
Component
0,0
print
direction
600,0
600,400
0
600
row1 yes 2
row2 yes 1
row3 no
row4 yes 4
translate
(0,600)
setClip
(0,0,600,400)
Page Index 0

Luego la ventana de la página impresa se desliza a lo largo del componente para imprimir la segunda página, con el índice uno.



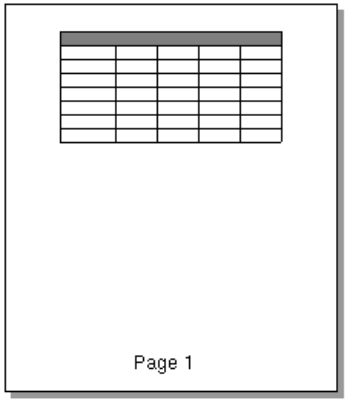
Este proceso continúa hasta que se alcanza la última página.



Imprimir un Componente JTable

La clase Report.java usa muchos de técnicas avanzadas cubiertas en esta sección para imprimir los datos y la cabecera de un componente **JTable** que expande muchas páginas. La salida de impresión también inlcuye un pié de página con el número de ésta.

Este diagrama muestra como sería la impresión:



```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.print.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.Dimension;

public class Report implements Printable{
  JFrame frame;
  JTable tableView;

  public Report() {
    frame = new JFrame("Sales Report");
    frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);}});

    final String[] headers = {"Description", "open price",
        "latest price", "End Date", "Quantity"};
    final Object[][] data = {
        {"Box of Biros", "1.00", "4.99", new Date(),
          new Integer(2)},
        {"Blue Biro", "0.10", "0.14", new Date(),
          new Integer(1)},
        {"legal pad", "1.00", "2.49", new Date(),
          new Integer(1)},
        {"tape", "1.00", "1.49", new Date(),
          new Integer(1)},
        {"stapler", "4.00", "4.49", new Date(),
          new Integer(1)},
        {"legal pad", "1.00", "2.29", new Date(),
          new Integer(5)}
```

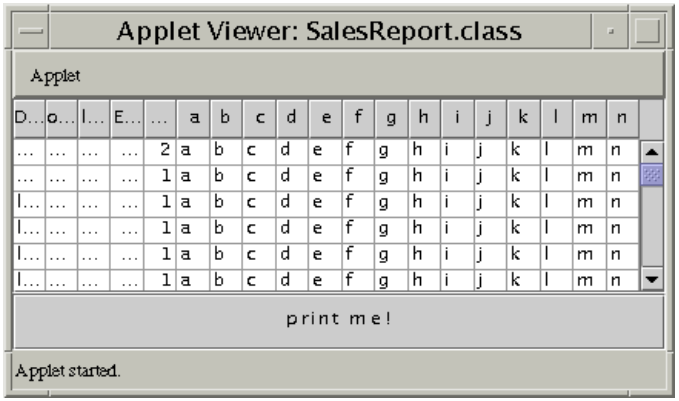
```
};

TableModel dataModel = new AbstractTableModel() {
    public int getColumnCount() {
      return headers.length; }
    public int getRowCount() { return data.length;}
    public Object getValueAt(int row, int col) {
            return data[row][col];}
    public String getColumnName(int column) {
            return headers[column];}
    public Class getColumnClass(int col) {
            return getValueAt(0,col).getClass();}
    public boolean isCellEditable(int row, int col) {
            return (col==1);}
    public void setValueAt(Object aValue, int row,
                  int column) {
            data[row][column] = aValue;
  }
};

tableView = new JTable(dataModel);
JScrollPane scrollpane = new JScrollPane(tableView);

scrollpane.setPreferredSize(new Dimension(500, 80));
frame.getContentPane().setLayout(
          new BorderLayout());
frame.getContentPane().add(
          BorderLayout.CENTER,scrollpane);
frame.pack();
JButton printButton= new JButton();

printButton.setText("print me!");

frame.getContentPane().add(
          BorderLayout.SOUTH,printButton);

// for faster printing turn double buffering off

RepaintManager.currentManager(
   frame).setDoubleBufferingEnabled(false);

printButton.addActionListener( new ActionListener(){
   public void actionPerformed(ActionEvent evt) {
     PrinterJob pj=PrinterJob.getPrinterJob();
     pj.setPrintable(Report.this);
     pj.printDialog();
     try{
       pj.print();
     }catch (Exception PrintException) {}
     }
   });

   frame.setVisible(true);
}

public int print(Graphics g, PageFormat pageFormat,
   int pageIndex) throws PrinterException {
   Graphics2D  g2 = (Graphics2D) g;
   g2.setColor(Color.black);
   int fontHeight=g2.getFontMetrics().getHeight();
   int fontDesent=g2.getFontMetrics().getDescent();

   //leave room for page number
```

```
        double pageHeight =
          pageFormat.getImageableHeight()-fontHeight;
        double pageWidth =
          pageFormat.getImageableWidth();
        double tableWidth = (double)
          tableView.getColumnModel(
          ).getTotalColumnWidth();
        double scale = 1;
        if (tableWidth >= pageWidth) {
                scale =  pageWidth / tableWidth;
        }

        double headerHeightOnPage=
                 tableView.getTableHeader(
                 ).getHeight()*scale;
        double tableWidthOnPage=tableWidth*scale;

        double oneRowHeight=(tableView.getRowHeight()+
                      tableView.getRowMargin())*scale;
        int numRowsOnAPage=
              (int)((pageHeight-headerHeightOnPage)/
                                  oneRowHeight);
        double pageHeightForTable=oneRowHeight*
                                    numRowsOnAPage;
        int totalNumPages=
              (int)Math.ceil((
                (double)tableView.getRowCount())/
                                    numRowsOnAPage);
        if(pageIndex>=totalNumPages) {
                      return NO_SUCH_PAGE;
        }

        g2.translate(pageFormat.getImageableX(),
                       pageFormat.getImageableY());
//bottom center
        g2.drawString("Page: "+(pageIndex+1),
            (int)pageWidth/2-35, (int)(pageHeight
            +fontHeight-fontDesent));

        g2.translate(0f,headerHeightOnPage);
        g2.translate(0f,-pageIndex*pageHeightForTable);

        //If this piece of the table is smaller
        //than the size available,
        //clip to the appropriate bounds.
        if (pageIndex + 1 == totalNumPages) {
           int lastRowPrinted =
                 numRowsOnAPage * pageIndex;
           int numRowsLeft =
                 tableView.getRowCount()
                 - lastRowPrinted;
           g2.setClip(0,
             (int)(pageHeightForTable * pageIndex),
             (int) Math.ceil(tableWidthOnPage),
             (int) Math.ceil(oneRowHeight *
                               numRowsLeft));
        }
        //else clip to the entire area available.
        else{
             g2.setClip(0,
             (int)(pageHeightForTable*pageIndex),
             (int) Math.ceil(tableWidthOnPage),
             (int) Math.ceil(pageHeightForTable));
```

```
            }

            g2.scale(scale,scale);
            tableView.paint(g2);
            g2.scale(1/scale,1/scale);
            g2.translate(0f,pageIndex*pageHeightForTable);
            g2.translate(0f, -headerHeightOnPage);
            g2.setClip(0, 0,
              (int) Math.ceil(tableWidthOnPage),
              (int)Math.ceil(headerHeightOnPage));
            g2.scale(scale,scale);
            tableView.getTableHeader().paint(g2);
            //paint header at top

            return Printable.PAGE_EXISTS;
        }

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

Imprimir un Informe de Ventas

La clase **Applet** SalesReport.java imprime un informe de ventas con filas que expánden sobre múltiples páginas con números en la parte inferior de cada página. Aquí se vé la aplicación cuando se lanza:



Necesitamos este fichero de policía para lanzar el applet:

```
grant {
  permission java.lang.RuntimePermission
                          "queuePrintJob";
};
```

Para lanzar el applet asumiendo un fichero de policía llamado **printpol** y una página HTML llamada **SalesReport.html**, teclearemos:

```
  appletviewer -J-Djava.security.policy=
                 printpol SalesReport.html
```

El diagrama muestra cómo se verá la impresión del informe:


1-1
2-1
3-1

Depurar Applets, Aplicaciones y Servlets

Una ley no escrita de la programación sentencia que gastatemos el 10 por cien de nuestro tiempo en el primer 90 por ciento de un proyecto, y el otro 90 por ciento de nuestro tiempo en el 10 por cierto restante. Esto suena igual que cualquiera de nuestros proyectos, probablemente estamosgastando el último 10 por ciento en depuración e integración. Mientras que hay cantidad de libros y de gente ayudándonos a empezar un progyecto, hay muy pocos recursor disponibles para ayudarnos a finalizarlo.

La buena noticia es que este capítulo se enfoca completamente en la depuración, y en evitar que nuestro proyecto se pase de tiempo. Usa ejemplos del mundo real para pasear a través de pasos sencillos para depurar y fijar nuestros programas. Cuando terminemos, deberemos ser unos expertos en el seguimiento de problemas en programas escritos en Java -- applets, aplicaciones y servlets -- de todas las formas y tamaños.

¿Tienes Prisa?

Si tienes un problema que te presiona y necesitas una respuesta ahora mismo, esta tabla podría ayudarte. Nos dice dónde encontrar las respuestas a los problemas más comunes a las que podemos acudir directamente.

Problema	Sección
El programa se cuelga o bloquea	Análisis y Seguimiento de Pila
Problemas en la ejecución del programa	Ir detrás de la silla con jdb
Problemas con Java Web Server™	Depurador de Servlets y Analizar y seguir Pistas

---

Recolección de Evidencias

El primer paso para intentar resolver cualquier problema es obtener tanta información como sea posible. Si podemos imagninarnos la escena de un crimen,sabemos que todo está chequeado, catalogado y analizado antes de alcanzar cualquier conclusión. Cuando se depura un programa, no tenemos armas, muestras de pelo, ni huellas dactilares, pero existen cantidad de evidencias que podemos obtener y que podrían contener o apuntar a la solución última. Esta sección explíca como recoger esas evidencias.

Instalación y Entorno

La plataforma Java ™ es una tecnología cambiante y de rápido movimiento. Podríamos tener más de una versión instalada en nuestro sistema, y esas versiones podrían haber sido instaladas como parte de la instalación de otros productos. En un entorno con versiones mezcladas, un programa puede experimentar problemas debido a los cambios de la plataforma en las nuevas versiones.

Por ejemplo, si las clases, las librerías, o las entradas de registro de Window de instalaciones anteriores permanecen en nuenstro sistema después de una actualización, hay una oportunidad de que la mezcla del nuevl software sea la causante de nuestros problemas y necesita ser investigada y eliminada. Las oportunidades para los problemas relacionados con la mezcla de versiones de software se ha incrementado con el uso de diferentes versiones de herramientas para desarrollar software de la plataforma Java.

La sección sobre Problemas con Versiones al final de este capítulo proporciona una lista completa de las principales versiones de la plataforma Java para ayudarnos a resolver nuestros problemas con versiones de software.

Path de Clases

En la plataforma Java 2, la variable de entorno **CLASSPATH** es necesaria para especificar a la propia aplicación dónde están sus clases, y no las clases de la plataforma Java como en versiones anteriores. Por eso es posible que nuestro **CLASSPATH** apunte a las clases de la plataforma Java desde versiones anteriores y nos cause problemas.

Para examinar el **CLASSPATH**, tecleamos esto en la línea de comando:

Windows 95/98/NT:
echo %CLASSPATH%

Unix:
echo $CLASSPATH

Las clases Java se cargan en primer lugar, primera forma básica de la lista **CLASSPATH**. Si la variable **CLASSPATH** contiene una referencia a un fichero **lib/classes.zip**, que apunta a una instalación diferente de la plataforma Java, esto peude causar que se cargen clases incomplatibles.

Nota: En la plataforma Java 2, las clases del sistema se eligen antes de cualquier clases de la lista **CLASSPATH** para minimizar de que se caeguen clases Java anteriores a la clase Java 2 del mismo nombre.

La variable **CLASSPATH** puede obtener su configuración desde la línea de comandos o desde las selecciones de configuración como aquellas especificadas en el Entorno de Usuario sobre Windows NT, un fichero **autoexec.bat**, o un fichero de arranque del shell **.cshrc** sobre Unix.

Podemos controlar las clases de la Máquina Virtual Java usadas para compilar nuestros programas con una opción especial de la línea de comandos que nos permite suministrar el **CLASSPATH** que querramos. La opción y parámetro de la plataforma Java 2 **-Xbootclasspath classpath**, y las versiones anteriores usan **-classpath classpath** y **-sysclasspath classpath**. Sin importar la versión que estamos ejecutando, el parámetro **classpath** especifica el classpath del sistema y del usuario, y los ficheros zip o JAR a usar en la compilación.

Para compilar y ejecutar el programa **Myapp.java** con un **CLASSPATH** suministrado en la línea de comandos, usamos las siguientes instrucciones:

Windows 95/98/NT:

En este ejemplo, la plataforma Java está instalada en el directorio **C:\java**. Tecleamos los siguiente en una sóla línea:

```
javac -J-Xbootclasspath:c\java\lib\tools.jar;c:
\java\jre\lib\rt.jar;c:\java\jre\lib\i18n.jar;.
  Myapp.java
```

No necesitamos la bandera **-J** para ejecutar el programa **Myapp** compilado, sólo tecleamos esto en una sóla línea:

```
java -Xbootclasspath:c:\java\jre\lib\rt.jar;c:
\java\jre\lib\i18n.jar;.  Myapp
```

Sistemas Unix:

En este ejemplo, la plataforma Java está instalada en el directorio **/usr/local/java**. Tecleamos todo en una sóla línea:

```
javac -J-Xbootclasspath:/usr/local/java/lib/tools.jar:
/usr/local/java/jre/lib/rt.jar:
/usr/local/java/jre/lib/i18n.jar:. Myapp.java
```

No necesitamos la bandera **-J** para ejecutar el programa **Myapp** compilado, sólo tecleamos esto en un sóla línea:

```
java -Xbootclasspath:/usr/local/java/jre/lib/rt.jar:
/usr/local/java/jre/lib/i18n.jar:. Myapp
```

Carga de Clases

Otra forma de analizar problemas con el **CLASSPATH** es localizar desde dónde está cargando las clases nuestra aplicación. La opción **-verbose** del comando **java** muestra de donde vienen los ficheros **.zip** o **.jar** cuando se carga. De esta forma, podremos decir si vienen del fichero zip de la plataforma Java o desde algún fichero JAR de la aplicación.

Por ejemplo, una aplicación podría estar usando la clase **Password** que escribimos para ella o podría estar cargando la clase **Password** desde la herramienta IDE instalado.

Deberíasmos ver cada nombre de fichero zip o Jar como se vé aquí:

```
$ java -verbose SalesReport
[Opened /usr/local/java/jdk1.2/solaris/jre/lib/rt.jar
        in 498 ms]
[Opened /usr/local/java/jdk1.2/solaris/jre/lib/i18n.jar
        in 60 ms]
[Loaded java.lang.NoClassDefFoundError from
        /usr/local/java/jdk1.2/solaris/jre/lib/rt.jar]
[Loaded java.lang.Class from
        /usr/local/java/jdk1.2/solaris/jre/lib/rt.jar]
[Loaded java.lang.Object from
        /usr/local/java/jdk1.2/solaris/jre/lib/rt.jar]
```

Incluir Código de Depurado

Una forma común de añadir código de diagnóstico a una aplicación es usa sentencias **System.out.println** en posiciones estratégicas de la aplicación. Esta técnica está bien durante el desarrollo, pero debemos acordarnos de eliminarlas

todas antes de liberar nuestro producto. Sin embargo hay otras aproximaciones que son tan sencillas y que no afectan al rendimiento de nuestra aplicación, y no muestra mensajes que no queremos que vea el cliente.

Activar la Información de Depuración en Tiempo de Ejecución

La primera alternativa a las clásicas sentencias de depuración **println** es activar la información de depuración en el momento de la ejecución. Una ventaja de esto es que no necesitamos recompilar ningún código si aparecen problemas mientras hacemos pruebase en la oficina del cliente.

Otra ventaja es que algunas veces los problemas de software pueden ser atribuidos a condiciones de carrera donde el mismo segmento de código se convierte en impredecible debido al tiempo entre cada iteracción del programa. Si controlamos el código de operación desde la línea de comandos en lugar de añadir sentencias de depuración **println**, podemos arreglar la secuencia de problemas que causa las condiciones de carrera que vienen desde el código **println**. Esta técnica también nos evita tener que añadir y eliminar las sentencias **println** y tener que recompilar nuestro código.

Esta técnica requiere que usemos una propiedad del sistema como bandera de depurado y que incluyamos código en la aplicación para comprobar que el valor de esta propiedad del sistema. Para activar la información de depuración desde la línea de comandos en el momento de la ejecución, arrancamos la aplicación y seleccionamos la propiedad del sistema debug a **true** de esta forma:

java -Ddebug=true TestRuntime

El código fuente que necesita la clase **TestRuntime** para examinar esta propiedad y configurar la bandera booleana debug de es el siguiente:

```
public class TestRuntime {
    boolean debugmode; //global flag that we test

    public TestRuntime () {

       String dprop=System.getProperty("debug");

       if ((dprop !=null) && (dprop.equals("yes"))){
            debugmode=true;
       }

       if (debugmode) {
           System.err.println("debug mode!");
       }
    }
}
```

Crear Versiones de Depuración y Producción en Tiempo de Compilación

Como se mencionó antes, un problem con la adición de sentencias **System.out.println** para depurar nuesto código es que debemos eliminarlas antes de liberar nuestro producto. Además de añadir código innecesario, las sentecias de depuración **println** pueden contener información que no queremos que vea el cliente.

Una forma de eliminar las sentencias de depuración **System.out.println** de nuestro código es usar la siguiente optimización del compilador para eleminar los corchetes pre-determinados de nuestos código en el momento de la compilazión y activar alguna algo similar a un depurador pre-procesador.

Este ejemplo usa una bandera booleana estática **dmode** que cuando se selecciona a **false** resulta en la eliminación el código de depuración y de las sentencias de depuración. Cuando el valor de **dmode** se selecciona a **true**, el código es incluido en el fichero class compilado y está disponible en la aplicación para propósitos de depuración.

```
class Debug {

  //set dmode to false to compile out debug code
  public static final boolean dmode=true;
}

public class TestCompiletime {

  if (Debug.dmode) {                        // These
    System.err.println("Debug message");    // are
    }                                       // removed
}
```

Usar Métodos de Diagnósticos

Podemos usar métodos de diagnóstico para solicitar información de depuración desde la máquina virtual Java (JVM). Los dos siguientes métodos de la clase **Runtime** siguel las llamadas a métodos y los bytes codes de la JVM que usa nuestra aplicación. Como estos dos métodos producen cantidad de informacióne s mejor seguir pequeñas cantidades de código, incluso tan pequeñas como una línea a la vez.

Para permitie seguir las llamadas, tenemos que arrancan la JVM con los comandos del intérprete **java_g** o **java -Xdebug**.

Para listar cada método cuando es invocado durante la ejecución, añadimos la

siguiente línea antes del código donde queremos empezar a seguir la pista y añadimos la correspondiente línea **traceMethodCalls** con el argumento seleccionado a false para desactivar el seguimiento. La información de seguimiento se muestra en la salida estándard.

```
// set boolean argument to false to disable
Runtime.getRuntime().traceMethodCalls(true);
callMyCode();
Runtime.getRuntime().traceMethodCalls(false);
```

Para ver cada línea en bytecodes cuando se ejecutan, añadimos la siguiente línea al código de nuestra aplicación:

```
// set boolean argument to false to disable
Runtime.getRuntime().traceInstructions(true);
callMyCode();
Runtime.getRuntime().traceInstructions(false);
```

También podemos añadir la siguiente línea para que nuestra aplicación vuelque la pila usando el método **dumpStack** de la clase **Thread**. La salida de este volcado de pila se explica en Análisis y Seguimiento de la Pila, pero ahora podemos pensar en la pila como un apunte de los threads que se están ejecutando en la JVM.

```
Thread.currentThread().dumpStack();
```

Añadir Información de Depurado

La información de variables locales no está incluida en el corazón de las clases del sistema de la plataforma Java. Por eso, si usamos una herramienta de depuración para listar variables lcoales para clases del sistema donde coloquemos comandos **stop** , obtendremos la siguiente salida, incluso cuando compilemos con la bandera **-g** como sugiere la salida. Esta salida es de una sesión **jdb**:

```
main[1] locals
No local variables: try compiling with -g
```

Para obtener acceso a la información de variables lcoales, tenemos que obtener el fuente (**src.zip** o **src.jar**) y recompilarlo con una bandera **debug**. Podemos obtener el fuente de la mayoría de las clases java.* classes con la descarga de los binarios desde java.sun.com.

Una vez hayamos descargado el fichero **src.zip** o **src.jar**, extraemos sólo los ficheros que necesitamos. Por ejemplo, para extraer la clase **String**, tecleamos esto en la línea de comandos:

```
unzip /tmp/src.zip src/java/lang/String.java
```

o

```
jar -xf /tmp/src.jar src/java/lang/String.java
```

Recompilamos la clase o clases extraidas con la opción **-g**. También podemos añadir nuestros propios diagnósticos adicionales sobre el fichero fuente en este momento.

```
javac -g src/java/lang/String.java
```

El compilador Java 2 **javac** ofrece más opciones que sólo la opción original **-g** para código de depuración, y podemos reducir el tamaño de nuestras clases usando **-g:none**, que nos ofrece una reducción de un 10% del tamaño.

Para ejecutar la aplicación con las nuevas clases compiladas, necesitamos usar la opcioón **bootclasspath** para que esas clases se utilicen en primer lugar.

Tecleamos lo siguiente en una sóla línea con espacio antes de **myapp**.

Plataforma Java 2 Win95/NT:

Este ejemplo asume que la plataforma Java está instalada en **c:\java**, y los ficheros fuente están en **c:\java\src**:

```
jdb -Xbootclasspath:c:\java\src;c:\java\jre\lib\rt.jar;c:
\java\jre\i18n.jar;.  myapp
```

Sistemas Unix:

Este ejemplo asume que la plataforma Java está instalada en **/usr/local/java**, y los ficheros fuente están en **/usr/local/java/src**.

```
jdb -Xbootclasspath:/usr/java/src;
/usr/java/jre/lib/rt.jar;
/usr/java/jre/i18n.jar;.  myapp
```

La siguiente vez que ejecutemos el comando **locals** veremos los campos internos de la clase que deseamos analizar.

Ejecutar Tests y Analizar

Si todavía tenemos problemas incluso después de haber revisado los problemas de instalación y de entorno y haber incluido código de depuración, es el momento de usar herramientas para probar y analizar nuestro programa.

Trabajar Detrás de la Silla con jdb

Aunque hay algunas muy buenas herramientas IDE en el mercado, la herramienta de depuración Java™, **jdb** y sus sucesores tienen un papel importante que jugar en la prueba y depuración de programa. algunas ventajas de **jdb** sobre los IDE es que es gratis, es independiente de la plataforma (algunos IDE no lo son), y se ejecuta como un proceso separado al programa que está depurando. El beneficio de ejecutar **jdb** como un proceso separado es que podemos añadir una sesión de depurado a un programa que está ejecutándose.

El lado negativo de usar **jdb** es que sólo hay un interface de línea de comandos, y trata con el mismo código que estámos tratando de depurar. Esto significa que si hay un bug enla máquina virtual Java, **jdb** se podría equivocar al intentar diagnisticar el mismo bug!

La nueva arquitectura **JBUG** se creó para resolver estos problemas en el **jdb**. **JBUG**, entre otras cosas, proporciona un API de ayuda de depuración en la máquina virtual Java llamado "Java VM Debug Interface" (JVMDI). Este ayudante se comunica con el depurador desde el final usando el "Java Debug Wire Protocol" (JDWP). La depuración desde el final usa el interface remoto "Java Debug Interface" (JDI) para enviar y recibir comando sobre el protocolo JDWP. JBug está disponible para la plataforma Java 2, y tiene un estilo **jdb** que aprenderemos más adelante.

Prueba Sencilla con jdb

De vuelta a la clásica herramienta **jdb**. Aquí tenemos uno sencillos pasos para analizar un programa usando **jdb**. Este primer ejemplo depura un programa de la aplicación startup. El ejemplo Depuración Remota muestra como conectarlo con una aplicación que se está ejecutando.

Arrancar la Sesión

Para empezar una sesión de depurado, compilamos el programa SimpleJdbTest.java con información completa de depurado usando **javac** y la bandera -g. En este ejemplo, el programa **SimpleJdbTest.java** es una aplicación pero también podría ser un applet. Los procedimientos para depurar aplicaciones son iguales que para depurar applets una que se ha empezado la sesión de depurado.

```
javac -g SimpleJdbTest.java
```

Luego arrancamos la herramienta **jdb** con el nombre de la clase del programa como parámetro:

```
jdb SimpleJdbTest
Initializing jdb...
0xad:class(SimpleJdbTest)
```

Para depurar un applet en el **appletviewer** usamos el parámetro **-debug** como en este ejemplo:

```
$ appletviewer -debug MyApplet.html
Initializing jdb...
0xee2f9808:class(sun.applet.AppletViewer)
>
```

Seleccionar un método de ruptura y métodos de listado

En este punto, sólo se ha cargado la clase **SimpleJdbTest**; no se ha llamado al constructor de la clase. Para hacer que el **jdb** se pare cuando el programa se inicializa por primera vez, ponemos un stop, o punto de ruptura, en el constructor usando el comando **stop in**. Cuando se seleccionan puntos de ruptura, instuirmos al **jdb** a ejecutar nuestro programa usando el comando **run** de esta forma:

```
stop in SimpleJdbTest.<init>
Breakpoint set in SimpleJdbTest.<init>
run
run SimpleJdbTest
running ...
main[1]
Breakpoint hit: SimpleJdbTest.<init>
                  (SimpleJdbTest:10)
```

La herramienta **jdb** se para en la primera línea del constructor. Para listar los método que fueron llamados hasta llegar a este punto de ruptura, introducimos el comando **where**:

```
main[1] where
```

```
[1] SimpleJdbTest.<init> (SimpleJdbTest:10)
[2] SimpleJdbTest.main (SimpleJdbTest:29)
```

Los métodos numerados de la lista es el último marco de pila que ha alcanzado la JVM. En este caso el último marco de pula es el constructor **SimpleJdbTest** que fue llamado desde el **SimpleJdbTest** main.

Siempre que se llama a un nuevo método, se sitúa en esta lista de pila. La tecnología Hotspot consigue alguna de sus ganancias de velocidad elimando un nuevo marco de pila cuando se llama a un nuevo método. Para obtener una apreciación general de dónde se paró el código, introducimos el comando **list**.

```
main[1] list
6               Panel p;
7               Button b;
8               int counter=0;
9
10              SimpleJdbTest() {
11                      setSize(100,200);
12                      setup();
13              }
14              void setup (){
```

Localizar la Fuente

Si el fuente del fichero class parado no está disponible en el path actual, podemos decirle a **jdb** donde encontrar el fuente con el comando **use** dándole el directorio fuente como un parámetro. En el siguiente ejemplo el fuente está un subdirectorio o carpeta llamado **book**.

```
main[1] list
Unable to find SimpleJdbTest.java
main[1] use book
main[1] list
6               Panel p;
7               Button b[];
8               int counter=0;
9
10      =>      SimpleJdbTest() {
```

Buscar un Método

Para ver que sucede en el método **setup** de **SimpleJdbText**, usamos el comando **step** para pasar a través de sus 4 líneas y ver lo que pasa.

```
main[1] step
```

```
main[1]
Breakpoint hit: java.awt.Frame.<init> (Frame:222)
```

Pero espera un minuto! Este es ahora el constructor de la clase **Frame**! Si lo seguimos pasaremos a través del constructor de la clase **Frame** y no el de la clase **SimpleJdbText**. Porque **SimpleJdbTest** desciende de la clase **Frame**, el constructor padre, que en este caso es **Frame**, es llamado sin avisarnos.

El comando step up

Podríamos continuar pasando y eventualmente volveríamos al constructor de **SimpleJdbTest**, pero para retornar inmediatamente podemos usar el comando **step up** para volver al constructor de **SimpleJdbTest**.

```
main[1] step up
main[1]
Breakpoint hit: SimpleJdbTest.<init>
                  (SimpleJdbTest:8)
```

El comando next

También podemos usar el comando **next** para obtener el método **setup**. En este siguiente ejemplo, la herramienta **jdb** ha aproximado que el fichero fuente está fuera del constructor cuando procesó el último comando **step up**. Para volver al constructor, usamos otro comando **step**, y para obtener el método **setup**, usamos un comando **next**. Para depurar el método **setup**, podemos **step** (pasar) a través del método **setup**.

```
main[1] step
Breakpoint hit: SimpleJdbTest.<init>
                  (SimpleJdbTest:11)
main[1] list
7               Button b[]=new Button[2];
8               int counter=0;
9
10              SimpleJdbTest() {
11                      setSize(100,200);<
12                      setup();
13              }
14              void setup (){
15                      p=new Panel();
16              }
main[1] next
Breakpoint hit: SimpleJdbTest.<init>
                  (SimpleJdbTest:12)
main[1] step
```

```
Breakpoint hit: SimpleJdbTest.setup (SimpleJdbTest:15)
```

El comando stop in

Otra forma de obtener el método **setup** es usar el comando **stop in SimpleJdbTest.setup**. Podemos listar el fuente de nuevo para saber donde estamos:

```
main[1] list
11                      setSize(100,200);
12                      setup();
13              }
14              void setup (){
15      =>              p=new Panel();
16                        b[0]= new Button("press");
17                        p.add(b[0]);
18                        add(p);
19
```

El comando print

Lo primero que hace el método **setup** es crear un **Panel p**. Si intentamos mostrar el valor de **p** con el comando **print p**, veremos que este valor es **null**.

```
main[1] print p
p = null
```

Esto ocurre porque la línea aún no se ha ejecutado y por lo tanto al campo **p** no se le ha asignado ningún valor. Necesitamos pasar sobre la sentencia de asignación con el comando **next** y luego usar de nuevo el comando **print p**.

```
main[1] next

Breakpoint hit: SimpleJdbTest.setup (SimpleJdbTest:16)
main[1] print p
p = java.awt.Panel[panel0,0,0,0x0,invalid,
                      layout=java.awt.FlowLayout]
```

Seleccionar Puntos de Ruptura en Métodos Sobrecargado

Aunque pasar a través de clases pequeñas es rápido, como regla general en grandes aplicaciones, es más rápido usar puntos de ruptura. Esto es así porque **jdb** tiene un conjunto de comandos muy simples y no teine atajos, por eso cada comando teine que ser pegado o tecleado por completo.

Para seleccionar un punto de ruptura en la clase **Button**, usamos **stop in java.awt.Button.<init>**

```
main[1] stop in java.awt.Button.<init>
java.awt.Button.<init> is overloaded,
          use one of the following:
void <init>
void <init>java.lang.String)
```

El mensaje explica porque **jdb** no puede parar en este método sin más información, pero el mensaje nos explica que sólo necesitamos ser explícitos en el tipo de retorno para los métodos sobrecargados en los que queremos parar. Para parar en el constructor de **Button** que crea este **Button**, usamos **stop in java.awt.Button.<init>java.lang.String)**.

El comando cont

Para continuar la sesión **jdb**, usamos el comando **cont** . La siguiente vez que el programa cree un **Button** con un constructor **String**, **jdb** se parará para que podamos examinar la salida.

```
main[1] cont
main[1]
Breakpoint hit: java.awt.Button.<init>
                            (Button:130)
```

Si la clase **Button** no ha sido compilada con información de depurado como se describió antes, no veremos los campos internos desde el comando **print**.

Limpiar Puntos de Ruptura

Para limpiar este punto de ruptura y que no pare cada vez que se cree un **Button** se usa el comando **clear**. Este ejemplo usa el comando **clear** sin argumentos para mostrar la lista de puntos de ruptura actuales, y el comando **clear** con el argumento **java.awt.Button:130.** para borrar el punto de ruptura **java.awt.Button:130.**.

```
main[1] clear
Current breakpoints set:
SimpleJdbTest:10
java.awt.Button:130
main[1] clear java.awt.Button:130
Breakpoint cleared at java.awt.Button: 130
```

Mostrar Detalles del Objeto

Para mostrar los detalles de un objeto, usamos el comando **print** para llamar al método **toString** del objeto, o usar el comando **dump** para mostrar los campos y valores del objeto.

Este ejemplo pone un punto de ruptura en la línea 17 y usa los comandos **print** y **dump** para imprimir y volcar el primer objeto **Button** del array de objetos **Button**. La salica del comando The **dump** ha sido abreviada.

```
main[1] stop at SimpleJdbTest:17
Breakpoint set at SimpleJdbTest:17
main[1] cont
main[1]
Breakpoint hit: SimpleJdbTest.setup (SimpleJdbTest:17)

main[1] print b[0]
b[0] = java.awt.Button[button1,0,0,0x0,invalid,
                                        label=press]
main[1] dump b[0]
b[0] = (java.awt.Button)0x163 {
private int componentSerializedDataVersion = 2
boolean isPacked = false
private java.beans.PropertyChangeSupport
                        changeSupport = null
long eventMask = 4096
transient java.awt.event.InputMethodListener
                        inputMethodListener = null
....
java.lang.String actionCommand = null
java.lang.String label = press
}
```

Finalizar la Sesión

Esto finaliza el sencillo ejemplo **jdb**. Para terminar una sesión **jdb**, se usa el comando **quit**:

```
0xee2f9820:class(SimpleJdbTest)
> quit
```

Depuración Remota

El **jdb** es un proceso de depuración externo, lo que significa que depura el programa enviándole mensajes hacia y desde el ayudante de la máquina virtual Java. Esto hacer muy fácil la depuración de un programa en ejecución, y nos ayuda a depurar un programa que interactua con el usuario final. Una sesión de depuración remota desde la línea de comandos no interfiere con la operación normal de la aplicación.

Arrancar la Sesión

Antes de la versión Java 2, lo único que se requería para permitir la depuración remota era arrancar el programa con la bandera **-debug** como primer argumento, y si la aplicación usa librerías nativas, terminanos el nombre de la librería con una **_g**. Por ejemplo, necesitaríamos una copia de la librería **nativelib.dll** como **nativelib_g.dll** para depurar con esta librería.

En Java 2, las cosas son un poco más complicada. Necesitamos decirla a la JVM dónde está el ficheo **tools.jar** usando la variable **CLASSPATH**. El fichero **tools.jar** normalmente se encuentra en el directorio **lib** de la instalación de la plataforma Java.

También necesitamos desactivar el compilador "Just In Time" (JIT) si existe. Este compilador se desactiva seleccionado la propiedad **java.compiler** a **NONE** o a una cadena vacía. Finalmente, como la opción **-classpath** sobreescribe cualquier classpath seleccionado por el usuario, también necesitamos añadir el **CLASSPATH** necesario para nuestra aplicación.

Poniéndo todo esto junto, aquí está línea de comandos necesaria para arrancar un programa en modo de depuración remoto. Se pone todo en una sóla línea e incluimos todas las clases necesarias en la línea de comandos.

```
Windows:

$ java -debug -classpath C:\java\lib\tools.jar;.
-Djava.compiler=NONE SimpleJdbTest
Agent password=4gk5hm

Unix:

$ java -debug -classpath /usr/java/lib/tools.jar:.
-Djava.compiler=NONE SimpleJdbTest
Agent password=5ufhic
```

La salida es el password del agente (en este caso, **4gk5hm**) si el programa se arranca de forma satisfactoria. La password de agente se suministra cuando se arranca **jdb** para que éste peuda encontrar la aplicación arrancada correspondiente en modo depuración en esa máquina.

Para arrancar **jdb** en modo depuración remoto, suministramos un nombre de host, que puede ser la misma máquina donde se está ejecutando el programa o **localhost** si estámos depurando en la misma máquina que el programa remoto, y la password de agente.

```
jdb -host localhost -password 4gk5hm
```

Listar Threads

Una vez dentro de la sesión **jdb**, podemos listar los threads activos actualmente, con el comando **threads**, y usar el comando **thread <threadnumber>**, por ejemplo, **thread 7** para seleccionar un thread para analizarlo. Una vez seleccionado un thread, usamos el comando **where** para ver los métodos que han sido llamados por este thread.

```
$ jdb -host arsenal -password 5ufhic
Initializing jdb...
> threads
Group system:
1. (java.lang.Thread)0x9        Signal dispatcher
                                cond. waiting
2. (java.lang.ref.Reference     0xb Reference Handler
      $ReferenceHandler)        cond. waiting
3. (java.lang.ref.              Finalizer
      Finalizer                 cond. waiting
      $FinalizerThread)0xd

4. (java.lang.Thread)0xe        Debugger agent
                                running
5. (sun.tools.agent.            Breakpoint handler
      Handler)0x10              cond. waiting
6. (sun.tools.agent.            Step handler
      StepHandler)0x12          cond. waiting
Group main:
7. (java.awt.                   AWT-EventQueue-0
       EventDispatchThread)     cond. waiting
       0x19
8. (sun.awt.                    PostEventQueue-0
      PostEventQueue)0x1b       cond. waiting
9. (java.lang.Thread)0x1c       AWT-Motif
                                running
10. (java.lang.Thread)0x1d      TimerQueue
                                cond. waiting
11. (sun.awt.                   Screen Updater
       ScreenUpdater)0x1f       cond. waiting
12. (java.lang.Thread)0x20      Thread-0
                                cond. waiting
> thread 7
AWT-EventQueue-0[1] where
  [1] java.lang.Object.wait (native method)
  [2] java.lang.Object.wait (Object:424)
  [3] java.awt.EventQueue.getNextEvent
                            (EventQueue:179)
  [4] java.awt.EventDispatchThread.run
```

```
                    (EventDispatchThread:67)
```

Listar el Fuente

Para listar el fuente, el thread necesita ser suspendido usando el comando **suspend**. Para permitir que un thread continúe usamos el comando **resume**. El ejemplo usa **resume 7.**

```
AWT-EventQueue-0[1] suspend 7
AWT-EventQueue-0[1] list
Current method is native
AWT-EventQueue-0[1] where
  [1] java.lang.Object.wait (native method)
  [2] java.lang.Object.wait (Object:424)
  [3] java.awt.EventQueue.getNextEvent
                            (EventQueue:179)
  [4] java.awt.EventDispatchThread.run
                (EventDispatchThread:67)
AWT-EventQueue-0[1] resume 7
```

Finalizar la Sesión

Cuando finalizamos de depurar remotamente este programa, eliminamos cualquier punto de ruptura restante antes de salir de la sesión de depuración. Para obtener una lista de estos puntos de ruptura usamos el comando **clear**, y para eliminarlos introducimos el comando **clear class:linenumber** de esta forma:

```
main[1] clear
Current breakpoints set:
SimpleJdbTest:10

main[1] clear SimpleJdbTest:10
main[1] quit
```

Usar el Piloto Automático

Un truco poco conocido del **jdb** es el fichero de arranque **jdb**. **jdb** automáticamente busca un fichero llamado **jdb.ini** en el directorio **user.home**. Si tenemos varios proyecto, es una buena idea seleccionar una propiedad **user.home** diferente para cada proyecto cuando arranquemos **jdb**. Para arrancar **jdb** con un fichero **jdb.ini** en el directorio actual, tecleamos esto:

```
jdb -J-Duser.home=.
```

El fichero **jdb.ini** nos permite seleccionar los comandos de configuración de **jdb**, como **use**, sin tener que introducir los detalles cada vez que ejecutamos **jdb**. El

siguiente fichero de ejemplo **jdb.ini** empieza una sesión **jdb** para la clase **FacTest**. Incluye los fuentes de la plataforma Java en el path de fuentes y le pasa el parámetro número 6 al programa. Se ejecuta y para en la línea 13, muestra la memoria libre, y espera una entrada posterior.

```
load FacTest
stop at FacTest:13
use /home/calvin/java:/home/calvin/jdk/src/
run FacTest 6
memory
```

Aquí está salida de la ejecución del fichero **jdb.ini**:

```
$ jdb -J-Duser.home=/home/calvin/java
Initializing jdb...
0xad:class(FacTest)
Breakpoint set at FacTest:13
running ...
Free: 662384, total: 1048568
main[1]
Breakpoint hit: FacTest.compute (FacTest:13)
main[1]
```

Podríamos peguntarnos si los ficheros **jdb.ini** pueden usarse para controlar una sesión **jdb** completa. Desafortunadamente, los comandos en un fichero **jdb.ini** se ejecutan de forma síncrona, y **jdb** no espera hasta que se llegue a un punto de ruptuira para ejecutar el siguiente comando. Podemos añadir retardos artificiales con comandos **help** repetidos, pero no hay garantía de que el thread se suspenda cuando necesitamos que lo haga.

Crear un Diálogo de Sesión

Podemos usar una característica poco conocida de **jdb** para obtener un registro de nuestra sesión de depuración. La salida es similar a la que veríamos si ejecutáramos **jdb -dbgtrace**.

Para permitir el diario **jdb**, creamos un fichero llamado **.agentLog** en el directorio donde estámos ejecutando **jdb** o **java -debug**. En el fichero **.agentLog**, ponemos el nombre del fichero en el que se escriba la información de la sesión en la primera línea.Por ejemplo, un fichero **.agentLog** podría tener estos contenidos:

```
jdblog
```

Cuando luego ejecutamos **jdb** o **java -debug**, veremos que la información de sesión **jdb** se muestra de esta forma. Podemos usar esta información para recuperar los puntos de ruptura y los comandos introducidos por si necesitamos reproducir esta sesión de depuración.

```
---- debug agent message log ----
[debug agent: adding Debugger agent to
system thread list]
[debug agent: adding Breakpoint handler
to system thread list]
[debug agent: adding Step handler to
system thread list]
[debug agent: adding Finalizer to
system thread list]
[debug agent: adding Reference Handler to
system thread list]
[debug agent: adding Signal dispatcher to
system thread list]
[debug agent: Awaiting new step request]
[debug agent: cmd socket:
Socket[addr=localhost/127.0.0.1,
port=38986,localport=3 8985]]
[debug agent: connection accepted]
[debug agent: dumpClasses()]
[debug agent: no such class: HelloWorldApp.main]
[debug agent: Adding breakpoint bkpt:main(0)]
[debug agent: no last suspended to resume]
[debug agent: Getting threads for HelloWorldApp.main]
```

Depurar Servlets

Podemo depurar servlets con los mismos comandos **jdb** usados para depurar un applet o una aplicación. JSDK "Java™ Servlet Development Kit" proporciona una programa llamado **servletrunner** que nos permite ejecutar un servlet sin un navegador web. En la mayoría de los sistemas, este programa simplemente ejecuta el comando **java sun.servlet.http.HttpServer**. Por lo tanto, podemos arrancar la sesión **jdb** con la clase **HttpServer**.

Un punto importante a recordar cuando depuramos servlets es que el servidor Web Java y **servletrunner** realizan la carga y descargas de servlets, pero no incluyen el directorio **servlets** en el **CLASSPATH**. Esto significa que los servlets se cargan usando un cargador de clases personalizado y no por el cargador de clases por defecto del sistema.

Ejecutar servletrunner en Modo Depuración

En este ejemplo, se incluye el directorio de ejemplos **servlets** en el **CLASSPATH**. Configuramos el CLASSPATH en modo depuración de esta forma:

```
Unix

$ export CLASSPATH=./lib/jsdk.jar:./examples:$CLASSPATH

Windows

$ set CLASSPATH=lib\jsdk.jar;examples;%classpath%
```

Para arrancar el programa servletrunner, podemos ejecutar el script de arranque suministrado llamado **servletrunner** o simplemente suministramos las clases servletrunner como parámetros de **jdb**. Este ejemplo usa el parámetro servletrunner.

```
$ jdb sun.servlet.http.HttpServer
Initializing jdb...
0xee2fa2f8:class(sun.servlet.http.HttpServer)
> stop in SnoopServlet.doGet
Breakpoint set in SnoopServlet.doGet
> run
run sun.servlet.http.HttpServer
running ...
main[1] servletrunner starting with settings:
```

```
port = 8080
backlog = 50
max handlers = 100
timeout = 5000
servlet dir = ./examples
document dir = ./examples
servlet propfile = ./examples/servlet.properties
```

Para ejecutar **SnoopServlet** en modo depuración, introducimos la siguiente URL donde **yourmachine** es la máquina donde arrancamos el servletrunner y **8080** es el número d puerto mostrado en las selecciones de salida.

```
http://yourmachine:8080/servlet/SnoopServlet
```

En este ejemplo **jdb** para en la primera línea del método **doGet** del servlet. El navegador espera una respuesta de nuestro servlet hasta que se pase el timeout.

```
main[1] SnoopServlet: init

Breakpoint hit: SnoopServlet.doGet (SnoopServlet:45)
Thread-105[1]
```

Podemo usar el comando **list** para saber dónde se ha parado **jdb** en el fuente.

```
Thread-105[1] list
41        throws ServletException, IOException
42        {
43      PrintWriter     out;
44
45 =>   res.setContentType("text/html");
46      out = res.getWriter ();
47
48      out.println("<html>");
49      out.println("<head>
                    <title>Snoop Servlet
                    </title></head>");
Thread-105[1]
```

El servlet puede continuar usando el comando **cont**.

```
Thread-105[1] cont
```

Ejecutar el Java Web Server en Modo Depuración

La versión JSDK no contiena las clases disponibles en el Java Web Server y también tiene su propia configuración servlet especial. Si no podemos ejecutar nuestro servlet desde **servletrunner**, otra opción puede ser ejecutar el servidor

web Java en modo depuración.

Para hacer esto añadimos la bandera **-debug** como el primer parámetro después del programa **java**. Por ejemplo en el script **bin/js** cambiamos la línea Java para que se parezca a esto. En versiones anteriores de la plataforma java 2, también tendremos que cambiar el puntero del programa a la variable $JAVA a **java_g** en vez de a **java**.

```
Antes:

exec $JAVA $THREADS $JITCOMPILER $COMPILER $MS $MX \

Depués:

exec $JAVA -debug $THREADS $JITCOMPILER
                                    $COMPILER $MS $MX \
```

Aquí está como conectar remotamente con el Java Web Server. La password de agente es generada sobre la slaida estandard desde el Java Web Server pero puede ser redirigida a un fichero en cualquier lugar. Podemos encontrar dónde chequeando los scripts de arranque del Java Web Server.

```
jdb -host localhost -password <the agent password>
```

Los servlets se cargan por un cargador de clases separado si están contenidos en el directorio **servlets**, que no está en el **CLASSPATH** usado cuando se arrancó el Java Web server. Desafortunadamente, cuando depuramos en modo remoto con **jdb**, no podemos controlar el cargador de clases personalizado y solicitarle que cargue el servlet, por eso tenemos que incluir el directorio **servlets** en el **CLASSPATH** para depurar o cargar el servlet requiriéndolo a través de un navegador y luego situando un punto de ruptura una vez que el servlet está ejecutando.

En este siguiente ejemplo, se incluye el **jdc.WebServer.PasswordServlet** en el **CLASSPATH** cuando se arranca el Java Web server. El ejemplo selecciona un punto de ruptura para parar el método **service** de este servlet, que es el método de proceso principal.

La salida estándard del Java Web Server standard produce este mensaje, que nos permite seguir con la sesión remota de **jdb**:

```
Agent password=3yg23k

$ jdb -host localhost -password 3yg23k
Initializing jdb...
> stop in jdc.WebServer.PasswordServlet:service
```

```
Breakpoint set in jdc.WebServer.PasswordServlet.service
> stop
Current breakpoints set:
        jdc.WebServer.PasswordServlet:111
```

El segundo **stop** lista los puntos de ruptura actuales en esta sesión y muestra el número de línea donde se encuentan. Ahora podemos llamar al servlet a través de nuestra página HTML. En este ejemplo, el servlet está ejecutando una operación POST:

```
<FORM METHOD="post" action="/servlet/PasswordServlet">
<INPUT TYPE=TEXT SIZE=15 Name="user" Value="">
<INPUT TYPE=SUBMIT Name="Submit" Value="Submit">
</FORM>
```

Obtenemos el control del thread del Java Web Server cuando se alcanza el punto de ruptura, y podemos continuar depurando usando las mismas técnicas que se usarón en la sección Depuración Remota.

```
Breakpoint hit: jdc.WebServer.PasswordServlet.service
(PasswordServlet:111) webpageservice Handler[1] where
[1] jdc.WebServer.PasswordServlet.service
                                (PasswordServlet:111)
[2] javax.servlet.http.HttpServlet.service
                                (HttpServlet:588)
[3] com.sun.server.ServletState.callService
                                (ServletState:204)
[4] com.sun.server.ServletManager.callServletService
                                (ServletManager:940)
[5] com.sun.server.http.InvokerServlet.service
                                (InvokerServlet:101)
```

Un problema común cuando se usan el Java WebServer y otros entornos de servlets es que se lanzan excepiones pero son capturadas y manejadas desde fuera del ámbito del servlet. El comando **catch** nos permite atrapar todas estas excepciones.

```
webpageservice Handler[1] catch java.io.IOException
webpageservice Handler[1]
Exception: java.io.FileNotFoundException
 at com.sun.server.http.FileServlet.sendResponse(
                                 FileServlet.java:153)
 at com.sun.server.http.FileServlet.service(
                                 FileServlet.java:114)
 at com.sun.server.webserver.FileServlet.service(
                                 FileServlet.java:202)
 at javax.servlet.http.HttpServlet.service(
```

```
                                    HttpServlet.java:588)
    at com.sun.server.ServletManager.callServletService(
                                    ServletManager.java:936)
    at com.sun.server.webserver.HttpServiceHandler
              .handleRequest(HttpServiceHandler.java:416)
    at com.sun.server.webserver.HttpServiceHandler
              .handleRequest(HttpServiceHandler.java:246)
    at com.sun.server.HandlerThread.run(
                                    HandlerThread.java:154)
```

Este sencillo ejemplo fue generado cuando los ficheros no se encontraban pero esta técnica puede usarse para problemas con datos posteados. Recordamos usar **cont** para permitir que el servidor web continúe. Para limpiar está trampa usamos el comando **ignore**.

```
webpageservice Handler[1] ignore java.io.IOException
webpageservice Handler[1] catch
webpageservice Handler[1]
```

Ozito

Depurar Eventos AWT

Antes del nuevo mecanismo de eventos del SWT presentado en el JDK 1.1 los eventos eran recibidos por un componente como un **TextField**, y propagado hacia arriba a sus componentes padre. Esto significa que podría simplemente añadir algún código de diagnóstico a los método **handleEvent** o **action** del componente para monitorizar los eventos que le han llegado.

Con la presentación del JDK 1.1 y el nuevo sistema de la cola de eventos, los eventos son enviados a una cola de eventos en lugar de al propio componente. Los eventos son despachados desde la cola de Eventos del Sistema a los oyentes de eventos que se han registrado para ser notificados cuando se despache un evento para ese objeto.

Usar AWTEventListener

Podemo suar un **AWTEventListener** para monitorizar los eventos AWT desde la cola de eventos del sistema. Este oyente toma una máscada de evento construida desde una operación **OR** de los **AWTEvent** que queremos monitorizar. Para obtener una simple lista de los eventos **AWTEvent**, usamos el comando **javap -public java.awt.AWTEvent**. Este ejemplo sigue la pista a los eventos de foco y del ratón.

Nota: No se debe utilizar **AWTEventListener** en un producto para la venta, ya que degrada el rendimiento del sistema.

```
//EventTest.java
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class EventTest extends JFrame {

  public EventTest() {
    JButton jb1=new JButton("hello");
    getContentPane().add(jb1);

    //AWTEventListener
    getToolkit().addAWTEventListener(
      new AWTEventListener() {
        public void eventDispatched(AWTEvent e) {
          System.out.println(e+"\n");
        }
      }, AWTEvent.MOUSE_EVENT_MASK |
```

```
            AWTEvent.FOCUS_EVENT_MASK
        );
    }

    public static void main (String args[]) {

      EventTest et=new EventTest();
      et.setSize(300,300);
      et.pack();
      et.show();
    }
}
```

Ozito

Analizar la Pila

Los desarrolladores siempre han considerado un misterio el seguimiento de pila. Hay muy poca o ninguna documentación disponible, y cuando obtenenos una, o necesitamos generar una, el tiempo lo prohibe. La siguiente sección descubre los secretos de la depuración con seguimiento de la pila, y al final, podremos considerar el seguimiento de pila como una herramienta útil para analizar otros programas -- no sólo los que no funcionan!

¿Qué es un seguimiento de pila producido por la plataforma ™? Es una imagen amigable para el usuario de los threads y monitores en la máquina virtual Java. Dependiendo de lo compleja que sea nuestra aplicación o applet, un seguimiento de pila puede tener un rango desde las cincuenta líneas hasta los cientos de líneas de diagnóstico.

Sin importar el tamaño del seguimiento de pila, hay unas pocas cosas importantes que nos pueden ayudar a diagnosticar la mayoría de los problemas de software sin importar si somos expertos o nuevos en la plataforma Java.

Hay tres formas populares para generar un seguimiento de pila: enviar una señal a la Máquina Virtual Java (JVM); la máquina virtual java genera un seguimiento de pila por nosotros; o usar herramientas de depuración o llamadas al API.

Enviar una Señal a la JVM

En plataformas UNIX podemos enviar una señal al programa con el comando **kill**. Está es la señal de salida, que es manejada por la máquina virtual Java.

Sistemas Unix:

Por ejemplo, en la plataforma Solaris™, podemos usar el comando **kill -QUIT**

process_id, donde **process_id** es el número de proceso de nuestro programa.

De forma alternativa podemos introducir la secuencia clave **<ctrl>** en la ventana donde se arrancó el programa.

El envío de esta señal instruye a un manejador de señal de la JVM a que imprima recursivamente toda la información de los threads y monitores que hay dentro de la JVM.

Windows 95/NT:

Para generar un seguimiento de pila en plataformas Windows 95 o Windows NT, introducimos esta secuencia **<ctrl><break>** en la ventana donde se está ejecutando el programa.

La JVM genera un Seguimiento de Pila

Si la JVM experimentó un error intermo como una violación de segmento o una fallo de página ilegal, llama a su propio manejador de señales para imprimir información sobre los threads y monitores.

Usar Herramientas de Depuración o Llamadas al API

Podemos generar un seguimiento parcial de la pila, (que en este caso es sólo información de los threads) usando el método **Thread.dumpStack**, o el método **printStackTrace** de la clase Throwable.

También podemos obtener información similar introduciendo el comando **where** dentro del depurador Java.

Si tenemos éxito al generar un seguimiento de pila, podremos ver algo similar a esto seguimiento de pila.

```
strings core | grep JAVA_HOME
```

En la versiones Java 2, los threads que llaman a métodos que resultan en una llamada a código nativo son indicados en el seguimiento de pila.

¿Qué Versión Genera el Seguimiento de Pila?

En la versión Java 2 el seguimiento de pila contiene la versión del JVM, la misma informació que veriámos usando el parámetro**-version**.

Sin embargo si no hay string de versión, podemos obtener una idea sobre de qué versión proviene este seguimiento de pila. Obviamente, si nosotros mismos hemos generado este seguimiento de pila no debe ser un problema, pero podríamos estar viendo un seguimiento de pila posteado en un grupo de noticias o en un artículo por e-mail.

Primero identificaremos donde está la sección "Registered Monitor Dump" en el seguimiento de pila:

- Si vemos un **utf8 hash table lock** en el "Registered Monitor Dump", esto es un seguimiento de pila de la plataforma Java 2. La versión final de la plataforma Java 2 también contiene un string de versión, por eso si no hay string de versión podría tratarse de una versión Beta de Java 2.
- Si vemos un **JNI pinning lock** y no vemos **utf8 hash lock**, esto es una versión JDK 1.1+.

Si no aparece ninguna de las cosas anteriores en el "Registered Monitor Dump", probablemente será una versión JDK 1.0.2.

¿Qué Plataforma Genera el Seguimiento de Pila?

También podemos saber si el seguimiento de pila viene de una máquina Windows 95, una NT, o UNIX buscando los threads que está esperadno. En una máquina Unix los threads que están esperando se llaman explícitamente. En una máquina Windows 95, o NT sólo se muestra un contador de los threads que están esperando:

- **Windows 95/NT:** Finalize me queue lock: <unowned> Writer: 1
- **UNIX:** Finalize me queue lock: <unowned>
  waiting to be notified "Finalizer Thread"

¿Qué Paquete Thread fue Utilizado?

Las JVMs de Windows 95 y Windows NT son por defecto threadas nativos del JVM. En UNIX las JVMs son por defectos, threads verdes de la JVM, usan una pseudo-implementación thread. Para hacer que la JVM use threads nativos necesitamos suministrar el parámetro **-native**, por ejemplo, **java -native MyClass**.

Verificando la existencia de un **Alarm monitor** en la salida del seguimiento de pila podemos identificar que este seguimiento de pila viene de un thread verde la JVM.

¿Qué son los Estados de Threads?

Veremos muchos threads diferentes en muy diferentes estados en una imagen del seguimiento de pila de JVM. Esta tabla descfribe varias claves y sus significados.

Clave	Significado
R	Thread runnable o ejecutándose
S	Thread suspendido
CW	Thread esperando en un condición variable
MW	Thread esperando un bloqueo de monitor
MS	Thread suspendido esperando un bloqueo de monitor

Normalmente, sólo los threadas en estados **R**, **S**, **CW** o **MW** deberían aparecer en el seguimiento de pila.

Los monitores se usan para controlar el acceso a código que sólo debería ser ejecutado por un sólo thread a la vez. Monitores se cubren en más detalles en la siguiente sección. Los otros dos estados de threads comunes que podríamos ver son R, threads ejecutables y CW, threads en una condición de estado de espera. Los threadas ejecutables son por definición threads que podrían ser ejecutados o estar ejecutándose en ese momento. En una máquina multi-procesador ejecutándo un sistema operativo realmente multi-procesador es posible que todos los threads ejecutables se estén ejecutando en el mismo momento. Sin embargo es más probable que otros threads ejecutables estén esperando un programador de threads para tener su turno de ejecución.

Podríamos pensar en los threads en una condición de estado de espera como esperando a que ocurra un evento. Frecuentemente un thread aparecerá en el estado CW si está en un **Thread.sleep** o en una espera sincronizada. En nuestro anterior seguimiento de pila el método main estaba esperando a que un thread se completara y se notificara su finalización. En el seguimiento de pila esto aparecerá como:

```
"main" (TID:0xebc981e0, sys_thread_t:0x26bb0,
                                  state:CW) prio=5
 at java.lang.Object.wait(Native Method)
 at java.lang.Object.wait(Object.java:424)
 at HangingProgram.main(HangingProgram.java:33)
```

El código que creó este seguimiento de fila es este:

```
synchronized(t1) {
  try {
    t1.wait();    //line 33
  }catch (InterruptedException e){}
}
```

En la versión Java 2 las operaciones de monitores, incluyendo nuestra espera aquí, son manejadas por la máquina virtual Java a través de una llamada JNI a sysMonitor. La condición de espera de un thread se mantiene en una cola de espera de monitor especial del objeto que está esperando. Esto explica porqué aunque seamos los únicos esperando por un objeto el código todavía necesita estar sincronizado con el objeto como si estuviera utilizano de hecho el monitor de ese objeto.

Examinar Monitores

Esto nos trae la otra parte del seguimiento de pila: el volcado de monitores. Si consideramos que la sección de threads de un seguimiento de pila identifica la parte multi-thread de nuestra aplicación, entonces la sección de monitores representa las partes de nuestra aplicación que usan un sólo thread.

Podría ser sencillo imaginar un monitor como un lavadero de coches. En muchos lavaderos de coches, sólo se puede lavar un coche a la vez. En nuestro código Java sólo un thread a la vez puede tener el bloqueo sobre una pieza sincronizada de código. Todos los demás threads esperan en la cola para entrar al código sincronizado como lo hacen los coches para entrar en el lavadero de coches.

Se puede pensar en un monitor como un bloqueo para un objeto, y cada objeto tiene un monitor. Cuando generamos un seguimiento de pila, los monitores se listan como registrados o no registrados. En la mayoría de los casos estos monitores registrados, o monitores del sistema, no deberían ser la causa de nuestro problema de software, pero nos ayudarán a entenderlos y reconocerlos. La siguiente tabla describe los monitores registrados mas comunes:

Monitor	Descripción
utf8 hash table	Bloquea el hashtable de Strings i18N definidos que fueron cargados desde la clase constant pool.
JNI pinning lock	Protege las copias de bloques de array a código de métodos nativos.
JNI global reference lock	¡Bloquea la tabla de referencias globales que contiene los valores que necesitan ser liberado explícitamete, y sobrevivirá al tiempo de vida de la llamada del método nativo.
BinClass lock	Bloquea el acceso a la lista de clases cargadas y resueltas. La tabla global de lista de clases.
Class linking lock	Protege datos de clases cuando se cargan librerías nativas para resolver referencias simbólicas
System class loader lock	Asegura que sólo un thread se carga en una clase del sistema a la vez.
Code rewrite lock	Protege el código cuando se intenta una optimización.
Heap lock	Protege la pila Java durante el manejo de memoria de la pila.
Monitor cache lock	Sólo un thread puede tener acceso al monitor cache a la vez este bloqueo asegura la integridad del monitor cache.

Dynamic loading lock	Protege los threads verdes de la JVM Unix de la carga de librería compartida stub libdl.so más de uno a la vez.
Monitor IO lock	Protege I/O física por ejemplo, abrir y leer.
User signal monitor	Controla el acceso al controlador de señal si hay una señal de usuario en un thread verde de la JVM.
Child death monitor	Controla accesos al proceso de información de espera cuando usamos llamadas al sistema de ejecución para ejecutar comandos locales en un thread verde de la JVM.
I/O Monitor	Controla accesos al fichero descriptor de threadas para eventos poll/select.
Alarm Monitor	Controla accesos a un controlador de reloj usado en threads verdes de la JVM para manejar timeouts
Thread queue lock	Protege la cola de threads activos.
Monitor registry	Sólo un thread puede tener acceso al registro de monitores al mismo tiempo que este bloqueo asegura la integridad de este registro.
Has finalization queue lock *	Protege la lista de objetos bloqueados que han sido recolectadas para la basura, y considera la finalización necesaria. Son copiados a la cola Finalize.
Finalize me queue lock *	Protege una lista de objetos que pueden ser finalizados por desocupados.
Name and type hash table lock *	Protege las tablas de constantes de las JVMs y sus tipos.
String intern lock *	Bloquea la hashtable de Strings definidos que fueron cargadas desde la clase constant pool
Class loading lock *	Asegura que sólo un thread carga una clase a la vez.
Java stack lock *	Protege la lista de segmentos libres de la pila

Nota: * bloqueo aparecidos sólo en los seguimientos de pre-Java 2.

El propio registro de monitores está protegido por un monitor. Esto significa que el thread al que pertenece un bloqueo es el último thread en usar un monitor. Es como decir que este thread es el thread actual. Como sólo un thread pueden entrar en un bloque sincronizado a la vez, otros threads se ponen a la cola para entrar en el código sincronizado y aparecen con el estado **MW**. En el volcado del caché de monitores, se denotan como threads "esperando para entrar". En el código de

usuario un monitor es llamado a acción siempre que se usa un bloque o método sincronizado.

Cualquier código que espere un objeto o un evento (método que espera) también tiene que estar dentro de un bloque sincronizado. Sin emabrgo, una vez que se llama a este método, se entrega el bloqueo sobre el objeto sincronizado.

Cuando el thread en estado de espera es notificado de un evento hacia el objeto, teine la competencia del acceso exclusivo a ese objeto, y tiene que obtener el monitor. Incluso cuando un thread a enviado un "notify event" a los threads que están esperando, ninguno de estos threads puede obtener realmente le control del monitor bloqueado hasta que el thread notificado haya abandonado el bloque de código sincronizado

Poner los Pasos en Práctica

Ejemplo 1

Consideremos un problema de la vida real como por ejemplo el Bug ID 4098756. Podemos encontrar más detalles sobre este bus en el JDC Bug Parade. Este bug documenta un problema que ocurre cuando usamos un componente **Choice** sobre Windows 95.

Cuando el usuario selecciona una de las opciones desde el componente **Choice** usando el ratón, todo va bien. Sin embargo, cuando el usuario intenta usar una tecla de fleca paramover la lista de opciones, la aplicación Java se congela.

Afortunadamente, este problema es reproducible y había un seguimiento de pila Java para ayudar a corregir el problem. El seguimiento de pila completo está en la página del bug, pero sólo necesitamos enfocarnos en estos dos threads claves:

```
"AWT-Windows" (TID:0xf54b70,
sys_thread_t:0x875a80,Win32ID:0x67,
state:MW) prio=5
java.awt.Choice.select(Choice.java:293)
sun.awt.windows.WChoicePeer.handleAction(
                              WChoicePeer.java:86)

"AWT-EventQueue-0" (TID:0xf54a98,sys_thread_t:0x875c20,
Win32ID:0x8f, state:R) prio=5
java.awt.Choice.remove(Choice.java:228)
java.awt.Choice.removeAll(Choice.java:246)
```

El thread **AWT-EventQueue-0** está en estado ejecutable dentro del método **remove**. **Remove** está sincronizado, lo que explíca por qué el thread **AWT-Windows** no puede entrar al método **select**. El thread **AWT-Windows** está en estado **MW** (monitor wait); sin embargo, sin embargo si seguimos el seguimiento de pila, esta situación no cambia aunque el interface gráfico de

usuario (GUI) parezca estár congelado.

Esto indica que la llamada a **remove** nunca retornó. Siguiendo el camino del código hacia la clase **ChoicePeer**, podemos ver que se está haciendo a un llamada al **MFC** nativo que no retorna, Es aquí donde está el problema real y es un bug de las clases corazón Java. El código del usuario esta bien.

Ejemplo 2

En este segundo ejemplo investigaremos un bug que al principio parece ser un fallo de Swing pero descubriremos que es debido al hecho que Swing no es seguro ante los threads.

El informa de bug también está disponible en la site JDCm el número del bug es 4098525.

Aquí tenemos un ejemplo del código usado para reproducir este problem. El díalogo modal se crea desde dentro del método **JPanel paint**.

```
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import javax.swing.*;

class MyDialog extends Dialog
                        implements ActionListener {

    MyDialog(Frame parent) {
        super(parent, "My Dialog", true);
        Button okButton = new Button("OK");
        okButton.addActionListener(this);
        add(okButton);
        pack();
    }

    public void actionPerformed(ActionEvent event) {
         dispose();
    }
}

public class Tester extends JPanel {

    MyDialog myDialog;
    boolean firstTime = true;

    public Tester (JFrame frame) throws Exception {
        super();
```

```
            myDialog = new MyDialog(frame);
        }

        void showDialogs() {
            myDialog.show();
        }

        public void paint(Graphics g) {
            super.paint(g);
            if (firstTime) {
               firstTime = false;
               showDialogs();
            }
        }

        public static void main(String args[])
                                  throws Exception {

           JFrame frame = new JFrame ("Test");
           Tester gui = new Tester(frame);
           frame.getContentPane().add(gui);
           frame.setSize(800, 600);
           frame.pack();
           frame.setVisible(true);
        }
    }
```

Cuando ejecutamos este programa encontramos que se bloquea al principio. Haciendo un seguimiento de pila podremos ver estos threads claves.

El seguimiento de pista que tenemos aquí es ligeramente diferente al que aparece en el informe del bug, pero tienen el mismo efecto. También usamos la versión Java 2 para generar el seguimiento y suministrar la opción **-Djava.compiler=NONE** cuando ejecutamos el programa para que podams ver los números de línea del fuent. El thread a buscar es el que tiene el estado MW, monitor de espaera que en este caso es el thread**AWT-EventQueue-1**

```
    "AWT-EventQueue-1" (
           TID:0xebca8c20, sys_thread_t:0x376660,
                                      state:MW) prio=6
     at java.awt.Component.invalidate(Component.java:1664)
     at java.awt.Container.invalidate(Container.java:507)
     t java.awt.Window.dispatchEventImpl(Window.java:696)
     at java.awt.Component.dispatchEvent(
                               Component.java:2289)
     at java.awt.EventQueue.dispatchEvent(
```

```
                                EventQueue.java:258)
    at java.awt.EventDispatchThread.run(
                                EventDispatchThread.java:68)
```

Si buscamos está línea en el fichero **java/awt/Component.java** que está contenido en el archivo **src.jar**, veremos esto:

```
    public void invalidate() {
        synchronized (getTreeLock()) { //line 1664
```

Es aquío donde nuestra aplicaciónse bloquea, está esperando a que el monitor **getTreeLock** se libere. La siguiente tarea es encontrar elthread que tiene bloqueado este monitos.

Para ver quién está bloqueando este monitor buscamos en el volcado del cache de Monitores y en este ejemplo podemos ver lo siguiente:

```
Monitor Cache Dump:
  java.awt.Component$AWTTreeLock@EBC9C228/EBCF2408:
        owner "AWT-EventQueue-0" ( 0x263850) 3 entries
  Waiting to enter:
    "AWT-EventQueue-1" (0x376660)
```

El monitor **getTreeLock** está actualmente bloqueado en un objeto de una clase interna creada especialmente **AWTTreeLock**. Este es el código para crear ese bloqueo en el fichero **Component.java**.

```
    static final Object LOCK = new AWTTreeLock();
    static class AWTTreeLock {}
```

El propietario actual es **AWT-EventQueue-0**. El thread llamó a nuestro método **paint** para crear nuesto **Dialog** modal mediante una llamada a **paintComponent**. El propio **paintComponent** fue llamado desde una llamada a **update** del **JFrame**.

¿Pero dónde se originó el bloqueo? Bien, ni hay una forma sencilla de encontrar qué parte del marco tiene el bloqueo pero una simple búsqueda de **javax.swing.JComponent** podremos ver que **getTreeLock** es llamado dentro del método **paintChildren** que dejamos en la línea 388.

```
at Tester.paint(Tester.java:39)
at javax.swing.JComponent.paintChildren(
                              JComponent.java:388)
```

El resto del puzzle se coloca junto analizando el método **MDialogPeer show**. El código del diálogo crea un nuevo ModalThread que es por lo que hemos visto un thread **AWT-Modal** en la salida del seguimiento de pila, este thread es usado para postear el diálogo. Es cuando el evento de despacha usando **AWT-EventQueue-1** que es usado para ser el proxy de despacho de eventos de AWT y es necesario un acceso al monitor **getTreeLock** y es aquí donde tenemos el bloqueo.

Desafortunadamente el código Swing no está diseñado para ser seguro con los threads por eso la solución en este ejemplo es no crear diálogos modales desde dentro de método paint de Swing. Ya que Swing tiene que hacer cantidad de bloqueos y cálculos; que las partes de un componente ligero que necesitan ser dibujadas deben estar fuertemente advertidas de que no incluyan código sincronizado o código que puede resultar en una llamada sincronizadac como en un diálogo modal, dentro del método **paint**.

Esto completa la teoria del seguimiento de pila Java, y ahora deberíamos saber qué busar la siguiente vez que veamos un seguimiento de pila. Para ahorrar tiempo, deberíamos hacer uso de la búsqueda de Bugs del JDC para ver si nuestro problema ha sido reportado por alguien más.

Lista de chequeo del Experto

Para sumarizar, estos son los pasos a tomar la proxima vez que nos crucemos con un problema en un programa Java:

- **Programas Colgados, bloqueados o congelados:** Si pensamos que nuestro programa está colgado, generamos un seguimiento de pila. Examinamos los threads en estados **MW** o **CW**. Si el programa está bloqueado, algunos threads del sistema se nos mostrarán como el thread actual porque la JVM no tendrá nada más que hacer
- **Programas Cascados o Abortados:** Sobre Unix buscaremos por un fichero corazón. Podemos analizar este fichero en una herramienta de depuración nativa como **gdb** o **dbx**. Buscamos los threads que hayan sido llamados por método nativos. Como la tecnología Java usa un modelo de memoria seguro, cualquier posible corrupción habrá ocurrido en el método nativo. Recordamos que la JVM también usa código nativo por lo que bien podría no ser un bug de nuestra aplicación.
- **Programas ocupados:** El mejor curso de acción que podemos tomar para los programas ocupados es generar frecuentes seguimientos de pila. Esto nos apuntará hacia el código que está causando los errores, y podrmos empezar nuestra investigación desde aquí.

Ozito

Problemas de Versiones

Esta sección proporciona una tabla que sumariza los problemas y soluciones relacionados con la tenencia de distintas versiones de la plataforma Java™ instalados en nuesto sistema.

Producto	Desarrollo
JDK 1.0.2	Utiliza **CLASSPATH** para encontrar y cargar las clases corazón del sistema. En Windows 95: **CLASSPATH=/usr/java/lib/classes.zip:.** En Unix: **CLASSPATH=c:\java\lib\classes.zip** Las librerías dinámicas Unix, los ficheros **.dll**, los objetos compartidos y fichero **.so** están localizados en la variable PATH. **Efectos laterales:** El fichero **Autoexec.bat** de Win95 contiene una variable **CLASSPATH** caducada seleccionad por el usuario o la instalaciónde otras aplicaciones. El Entorno de usuario de WinNt contiene un vieja variable **CLASSPATH**. Los scripts Unix **.cshrc**, **.profile**, o **.login** contiene un **CLASSPATH** erróneo. La variáble de entorno **JAVA_HOME** también es usada por programas para comprobar si no está seleccionada. Podemos borrar este campo en el shel Bourne (sh) de esta forma: **unset JAVA_HOME** **Diagnósticos:** Usamos la opción **-classpath** para forzar a la máquina virtual Java a que use sólo la línea de comandos. Sólo **CLASSPATH**: **java -classpath c:\java\lib\classes.zip;. myapp**

Producto	Desarrollo

JDK 1.1	Usa paths relativos para encontrar el fichero **classes.zip** desde la instalación de la plataforma Java. La variable de entorno **CLASSPATH** se usa para cargar las clases de la aplicación. **Efectos laterales:** Otras versiones Java encontradad en el path de la aplicación podrían ser cargadas si el directorio **bin** del JDK no se selecciona explítamente delante de la variable de entorno **PATH**. **Diagnósticos:** Usamos la opción **-classpath** para forzar a la máquina virtual Java a que use sólo la línea de comandos. Sólo **CLASSPATH**: **java -classpath c:\java\lib\classes.zip;. myapp**

Producto	**Desarrollo**
Plataforma Java 2	La plataforma está dividida en un Entorno de Ejecución Java (JRE) y un compilador Java. El JRE está incluido como un subdirectorio de la versión, y los tradiciones programas **java** y **javac** del directorio **bin** llaman directamente el programa real en el directorio **jre/bin**. Los archivos JAR que contienen las clases del sistema de la plataforma Java, **rt.jar** y **i18.jar**, están localizados en el directorio **jre/lib** con un path de búsqueda relativo. **Efectos Laterales:** Si las aplicaciones anteriores usaban el fichero **classes.zip** para cargar las clases del sistema de la plataforma Java, podría intentar cargar erróneamente un conjunto de clases adicionales. **Diagnósticos:** Usamos la opción **-Xbootclasspath** para forzar al máquina virtual Java a usar el **CLASSPATH** sumnistrado en la línea de comandos:**java -Xbootclasspath:c:\java\jre\lib\rt.jar; c:\java\jre\lib\i18n.jar;. myapp** Podríamos necesitar suministrar esto como una opción de la línea de comandos de esta forma: **javac -J-Xbootclasspath:c\java\lib\tools.jar;c: \java\jre\lib\rt.jar;c:\java\jre\lib\i18n.jar;. myapp.java**

Producto	**Desarrollo**

Java Plug-In

Sobre Windows 95 y Windows NT usamos el registro para encontrar plug-in de la plataforma Java instalados.

Efectos Laterales:
El registro podría estar corrompido, o el plug-in eliminado físicamente pero no del registro.

Diagnósticos:
Mostrar las propiedades **java.version** y **java.class.path** en nuesto código y verlo en la Consola del Java Plug-in Console

```
System.out.println("version="+System.getProperty(
  "java.version"
  ));
System.out.println("class path="+System.getProperty(
  "java.class.path"
  ));
```

Si hay un conflicto, chequeamos el registro con el comando **regedit**, buscamos la palabra *VM*, y si existe la borramos y reinstalamos el plug-in.

Producto	**Desarrollo**
Netscape	usa ficheros **.jar** como **java40.jar** del directorio **netscape**.

Efectos Laterales:
No todas las versiones de Netscape son totalmente compatibles con JDK 1.1. Podemos obtener actualizaciones en http://www.netscape.com.

Diagnósticos:
Arrancamos el navegador desde la línea de comandos con la opción **-classes**.

Producto	**Desarrollo**

Internet Explorer	Usa ficheros **.cab** para contener las clases del sistema. También usa el registro del sistema sobre Windows 95/NT. **Efectos Laterales:** Usamos el comando **regedit** para buscar la palabra *VM*. Esa es la entrada **CLASSPATH** donde podemos añadir nuestras propias clases. **Diagnósticos:** El registro puede corromperse. Buscamos CLASSPATH usando el programa regedit y editamos el valor al que apunta CLASSPATH.

Mejorar el Rendimiento por Diseño

Las restricciones del ancho de banda en las redes alrededor del mundo hacen de las operaciones basadas en red potenciales cuellos de botella que pueden tener un importante impacto en el rendimiento de las aplicaciones. Muchas aplicaciones de red están disañadas para usar almacenes de conexiones y por ello pueden reutilizar conexiones de red existentes y ahorrar el tiempo y la sobrecarga que conllevan el abrir y cerrar conexiones de red.

Junto con el almacen de conexiones, hay otras características que podemos diseñar dentro de nuestros programas para mejorar el rendimiento. Este capítulo explica cómo podemos diseñar un applet para que descargue ficheros y recursos de forma más eficiente, o diseñar un programa basado en threads para usar un almacen de threads para ahorrarnos el costoso proceso de arrancar threads.

Mejorar la Velocidad de Descarga de un Applet

El rendimiento de descarga de un applet se refiere al tiempo que tarda el navegador en descargar todos los ficheros y recursos que necesita para arrancar el applet. Un factor importante que afecta al rendimiento de la descarga del applet es el número de veces que tiene que solicitar datos al servidor. Podemos reducir el número de peticiones empaquetando las imagenes del applet en un fichero class, o usando un archivo JAR.

Empaquetar Imágenes en un Clase

Normalmente, si un applet tiene seis imágenes de botones se traducen en seis solicitudes adicionales al servidor para cargar esos ficheros de imágenes. Seis solicitudes adicionales podrían no parecer demasiadas en una red interna, pero en las conexiones de baja velocidad y eficiencia, esas solicitudes adicionales pueden tener un impacto muy negativo en el rendimiento. Por eso, nuestro último objetivo será cargar el applet tan rápido como sea posible.

Una forma de almacenar imágenes en un fichero class es usar un esquema de codificación ASCII como X-PixMap (XPM). De esta forma, en vez de mantener la imágenes en ficheros GIF en el servidor, los ficheros son codificados como un **Strings** y son almacenados en un sólo fichero class.

Este código de ejemplo usa páquetes del ganador de la JavaCup del JavaOne 1996, que contenía las clases **XImageSource** y **XpmParser**. Estas clases proporciona todo los necesario para leer un fichero **XPM**. Podemos ver esto ficheros en SunSite.

Para el proceso inicial de codificación, hay un número de herramientas gráficas que

podemos usar para crear fichero **XPM**. En Solaris podemos usar **ImageTool** o una variedad de otros GNU image packages. Podemos ir a la web site Download.com para obtener software de codificación para las plataformas Windows.

El siguiente código extraido del ejemplo de código MyApplet que carga imágenes. Podemos ver el **String** codificado en la definición XPM de imágenes.

La clase **Toolkit** crea un objeto **Image** para cada imagen desde el objeto fuente XPM Image.

```
Toolkit kit = Toolkit.getDefaultToolkit();
  Image image;
  image = kit.createImage (new XImageSource (_reply));
  image = kit.createImage (new XImageSource (_post));
  image = kit.createImage (new XImageSource (_reload));
  image = kit.createImage (new XImageSource (_catchup));
  image = kit.createImage (new XImageSource (_back10));
  image = kit.createImage (new XImageSource (_reset));
  image = kit.createImage (new XImageSource (_faq));
```

La alternativa técnica de abajo usa ficheros GIF. Requiere una petición al servidor para cada imagen cargada.

```
Image image;
  image = getImage ("reply.gif");
  image = getImage ("post.gif");
  image = getImage ("reload.gif");
  image = getImage ("catchup.gif");
  image = getImage ("back10.gif");
  image = getImage ("reset.gif");
  image = getImage ("faq.gif");
```

Esta técnica reduce el trafico de la red porque todas las imágenes están disponibles en un sólo fichero class.

- Usar imágenes XPM codificadas hace más grande el fichero de la clase, pero el número de peticiones de red es menor.
- Al hacer que las definiciones de imágenes XPM formen parte del fichero class, hacemos que el proceso de carga de imágenes sea parte de la carga normal del fichero class del applet sin clases extras.

Una vez cargado, podemos usar las imágenes para crear botones u otros

componentes del interface de usuario. El siguiente segmento de código muestra cómo usar la imágenes con la clase **javax.swing.JButton**.

```
ImageIcon icon = new ImageIcon (
                   kit.createImage (
                     new XImageSource (_reply)));
JButton button = new JButton (icon, "Reply");
```

Usar Ficheros JAR

Cuando un applet consta de más de un fichero, podemos mejorar el rendimiento de la descarga con ficheros JAR. Un fichero JAR contiene todos los ficheros del applet en un sólo fichero más rápido de dsacargar. Mucha parte del tiempo ahorrado viene de la reducción del número de conexiones HTTP que el navegador tiene que hacer.

El capítulo: Desarrollar Nuestra Aplicación tiene información sobre cómo crear y firmar ficheros JAR.

El código HTML de abajo usa la etiqueta **CODE** para especificar el ejecutable del applet **MyApplet**, y la etiqueta **ARCHIVE** especifica el fichero JAR que contiene todos los ficheros relacionados con **MyApplet**. El ejecutable especificado por la etiqueta **CODE** algunas veces es llamado **code base**.

Por razones de seguridas los ficheros JAR listados por el parámetro **archive** deben estar en el mismo directorio o subdirectorio que el **codebase** del applet. Si no se suministra el parámetro **codebase** el directorio de donde se cargó el applet se usa como el **codebase**.

El siguiente ejemplo especifica **jarfile** como el fichero JAR que contiene todos los ficheros relacionados para el ejecutable **MyApplet.class**.

<APPLET CODE="MyApplet.class" ARCHIVE="jarfile" WIDTH="100" HEIGHT="200"> </APPLET>

Si la descarga del applet usa múltiples ficheros JAR como se muestra en el siguiente segmento HTML, el **ClassLoader** carga cada fichero JAR cuando el applet arranca. Por eso, si nuestro applet usa algunos ficheros de recursos de forma infrecuente, el fichero JAR que contiene esos ficheros es descargado sin importar si los recursos van a ser usados durante la sesión o no.

<APPLET CODE="MyApplet.class" ARCHIVE="jarfile1, jarfile2" WIDTH="100" HEIGHT="200"> </APPLET>

Para mejorar el rendimiento cuando se descargan fichero no usados de forma frecuente, ponemos los ficheros usados más frecuentemente dentro de un fichero JAR y los ficheros menos usados en el directorio de la clase del applet. Los ficheros usados poco frecuentemente son localizados y descargados sólo cuando el

navegador los necesita.

Almacen de Threads

El servidor de applets Java Developer Connection℠ (JDC) y el Java Web Server™ hacen un uso extensivo del almacen de threads para mejorar el rendimiento. El almacen de threads es crear un suministro de threads durmientes al principio de la ejecución. Como el proceso de arranque de un thread es muy caro en términos de recursos del sistema, el almacen de threads hace el proceso de arrancada un poco más lento, pero aumenta el rendimiento en tiempo de ejecución porque los threads durmientes (o suspendidos) sólo se despiertan cuando cuando son necesarios para realizar nuevas tareas.

Este código de ejemplo tomado de la clase Pool.java muestra una forma de implementar la fusión de threads, En el constructor de la fusión (mostrado abajo), se inicializan y arrancan los **WorkerThreads**. La llamada al método **start** ejecuta el método **run** del **WorkerThread**, y la llamada a **wait** suspende el **Thread** mientras el **Thread** espera a que llegue un trabajo. La última línea del constructor empuja el **Thread** durmiente hacia la pila.

```
public Pool (int max, Class workerClass)
                            throws Exception {

    _max = max;
    _waiting = new Stack();
    _workerClass = workerClass;
    Worker worker;
    WorkerThread w;
    for ( int i = 0; i < _max; i++ ) {
        worker = (Worker)_workerClass.newInstance();
        w = new WorkerThread ("Worker#"+i, worker);
        w.start();
        _waiting.push (w);
    }
}
```

Junto al método **run**, la clase **WorkerThread** tiene un método **wake**. Cuando viene el trabajo, se llama al método **wake**, que asigna los datos y notifica al **WorkerThread** durmiente (el inicializado por el **Pool**) para recuperar la ejecución. El método **wake** llama a **notify** hace que el **WorkerThread** bloqueado salga del estado de espera, y se ejecuta el método **run** de la clase HttpServerWorker. Una vez realizado el trabajo, el **WorkerThread** se pone de nuevo en el **Stack** (asumiento que el **Pool** de threads no está lleno) o termina.

```
synchronized void wake (Object data) {
        _data = data;
```

```
        notify();
    }

    synchronized public void run(){
      boolean stop = false;
      while (!stop){
          if ( _data == null ){
            try{
               wait();
            }catch (InterruptedException e){
                  e.printStackTrace();
                  continue;
            }
          }

          if ( _data != null ){
            _worker.run(_data);
          }

          _data = null;
          stop = !(_push (this));
      }
    }
```

En este alto nivel, el trabajo entrante es manejado por el método **performWork** en la clase **Pool**. Cuando viene el trabajo, se saca de la pila un **WorkerThread** existente (o se crea uno nuevo si el **Pool** está vacío). El **WorkerThread** durmiente es activado mendiate una llamada a su método **wake**.

```
    public void performWork (Object data)
                    throws InstantiationException{
      WorkerThread w = null;
      synchronized (_waiting){
         if ( _waiting.empty() ){
            try{
              w = new WorkerThread ("additional worker",
              (Worker)_workerClass.newInstance());
              w.start();
            }catch (Exception e){
              throw new InstantiationException (
                          "Problem creating
                          instance of Worker.class: "
                          + e.getMessage());
            }
          }else{
            w = (WorkerThread)_waiting.pop();
```

```
        }
      }
      w.wake (data);
    }
```

El constructor de la clase HttpServer.java crea un nuevo ejemplar **Pool** para servir ejemplares de la clase HttpServerWorker. Los ejemplares **HttpServerWorker** se crean y almacenan como parte de los datos **WorkerThread**. Cuando se activa un **WorkerThread** mediante una llamada a su método **wake**, el ejemplar **HttpServerWorker** es invocado mediante su método **run**.

```
try{
    _pool = new Pool (poolSize,
                  HttpServerWorker.class);
  }catch (Exception e){
    e.printStackTrace();
    throw new InternalError (e.getMessage());
  }
```

Este código está en el método **run** de la clase HttpServer.java. Cada vec que viene una petición, el dato es inicializado y el **Thread** empieza el trabajo.

Nota: Si creamos un nuevo **Hashtable** por cada **WorkerThread** provocamos demasiada sobrecarga, sólo modificamos el código para que no use la abstración **Worker**.

```
try{
        Socket s = _serverSocket.accept();
        Hashtable data = new Hashtable();
        data.put ("Socket", s);
        data.put ("HttpServer", this);
        _pool.performWork (data);
  }catch (Exception e){
        e.printStackTrace();
  }
```

El almacen de threads es una técnica efectiva de ajuste de rendimiento que coloca el caro proceso de arranque de threads en la arrancada de la aplicación. De esta forma, el impacto negativo en el rendimiento ocurre sólo una vez durante el arrancada del programa donde se nota menos.

Ozito

Almacen de Conexiones

Si hemos usado SQL u otra herramienta similar para conectarnos con una base de datos y actúar sobre los datos, probablemente habremos notado que la obteneción de la conexión y el login es la parte que tarda más tiempo. Una aplicación puede fácilmente tardar varios segundos cada vez que necesita establecer una conexión.

El varsiones anteriores a JDBC™ 2.0 cada sesión de base de datos requería una nueva conexión y un login incluso si la conexión anterior usaba la misma tabla y cuenta de usuario. Si estámos usando versioens anteriores al JDBC 2.0 y queremos mejorar el rendimiento, podemos cachear las conexiones JDBC.

Las conexiones cacheadas se mantienen un objeto pool en tiempo de ejecución y pueden ser utilizadas y reutilizadas cuando las necesite la aplicación. Una forma de implementar un objeto pool es hacer una una simple hashtable de objetos conection. Sin embargo, una forma más sencilla de hacerlo es escribir un **driver** JDBC envuelto que es un intermediario entre la aplicación y la base de datos.

La envoltura trabaja particulamente en los Beans de Enterprise que san persistencia manejada por el Bean por dos razones: 1) Sólo se carga una clase **Driver** por cada Bean, y 2) los detalles específicos de la conexión se manejan fuera del Bea.

Esta sección explica cómo escribir una clase **Driver** JDBC envuelta

---

Clases Wrapper

El **Driver** JDBC envuelto creado para estos ejemplos consta de las siguientes clases:

- **JDCConnectionDriver**
- **JDCConnectionPool**
- **JDCConnection**

Driver de Conexión

La clase JDCConnectionDriver.java implementa el interface **java.sql.Driver**, que proporciona método para cargar drivers y crear nuevas conexiones a bases de datos.

Un objeto **JDCConnectionManager** es creado por una aplicación que pretende una conexión con una base de datos. La aplicación proprociona el ULR para la base de datos, el ID del usuario y la password.

El constructor **JDCConnectionManager** hace esto:

- Registra el objeto **JDCConnectionManager** con **DriverManager**.
- Carga la clase **Driver** pasada al constructor por el programa llamante.
- Inicializa un objeto **JDCConnectionPool** para las conexiones con la URL de la base de datos, el ID y el password del usuario pasados al constructor por el programa llamante.

```
public JDCConnectionDriver(String driver,
                String url,
                String user,
                String password)
        throws  ClassNotFoundException,
                InstantiationException,
                IllegalAccessException,
                SQLException {

  DriverManager.registerDriver(this);
  Class.forName(driver).newInstance();
  pool = new JDCConnectionPool(url, user, password);
}
```

Cuando el programa llamante necesita una conexión con la base de datos, llama al método **JDCConnectionDriver.connect**, que a su vez, llama al método **JDCConnectionPool.getConnection**.

Almacen de Conexiones

La clase JDCConnectionPool.java tiene conexiones disponibles para el programa llamando en su método **getConnection**. Este método busca una conexión disponible en el almacen de conexiones. Si no hay ninguna disponible, crea una nueva conexión. Si hay una conexión disponible en el almacen, el método **getConnection** alquila la conexión y la devuelve al programa llamante.

```
public synchronized Connection getConnection()
        throws SQLException {
```

```
    JDCConnection c;
    for(int i = 0; i < connections.size(); i++) {
       c = (JDCConnection)connections.elementAt(i);
       if (c.lease()) {
          return c;
       }
    }

    Connection conn = DriverManager.getConnection(
                          url, user, password);
    c = new JDCConnection(conn, this);
    c.lease();
    connections.addElement(c);
    return c;
  }
```

La clase JDCConnection.java representa una conexión JDBC en el almacen de conexiones, y esencialmente es una envoltura alrededor de un conexión real JDBC. El objeto **JDCConnection** mantiene una bandera de estado para indicar si la conexión está en uso y el momento en que la conexión se sacó del almacen. Este tiempo es usado por la clase **ConnectionReaper.java** para identificar las conexiones colgadas.

Bloqueos y Cuelgues

Mientras que muchos clientes y servidores de bases de datos tiene formas de manejar los bloqueos y los cuelgues y no tenemos que preocuparnos de escribir código para manejar estas situaciones, muchos de los nuevos modelos de base de datos ligeros distribuidos no están tan bien equipados. La clase conection pool proporciona una cosechador de conexiones muerta para manejar dichas situacciones.

La clase ConnectionReaper decide que una clase está muerta cuando se cumplen las siguientes condiciones:

- La conexión está marcada como que está en uso.
- La conexión es más vieja que tiempo de timeout preseleccionado.
- La conexión falla en un chequeo de validación.

El chequeo de validación ejecuta una simple consulta SQL sobre la conexión para ver si lanza una excepción. En este ejemplo, el método de validación solicita una descripción de alto nivel de las tablas de la base de datos. Si una conexión falla el test de validación, se cierra, se inicia una nueva conexión con la base de datos y se añade al almacen de conexiones.

```
  public boolean validate() {
```

```
    try {
       conn.getMetaData();
    }catch (Exception e) {
       return false;
    }
    return true;
  }
```

Cerrar Conexiones

La conexiónes devuelta al almacen de conexiones cuando el programa llamante llama al método **JDCConnection.close** en su claúsula**finally**.

```
  public void close() throws SQLException {
    pool.returnConnection(this);
  }
```

Aplicación de Ejemplo

Usamos un almacen de conexiones en una aplicación de forma similar a como usaríamos cualquiere otro driver JDBC. Aquí está el código de un RegistrationBean controlado por el Bean. Este **RegistrationBean** se ha adaptado desde la casa de subastas de JavaBeans enterprise descrito en los coaítulo 1 -3.

Cuando se crea el primer objeto **RegistrationBean**, crea un ejemplar estático de la clase **JDCConnectionDriver**. Este objeto driver estático se registra a sí mismo con el **DriverManager** en el constructor **JDCConnectionDriver** poniendo disponibles la solicitudes de conexiones para todos los objetos **RegistrationBean** creados por la aplicación cliente.

Pasar la URL como **jdbc:jdc:jdcpool** en el método **getConnection** permite que el **DriverManager** corresponda la **getConnection** solicitada al driver registrado. El **DriverManager** usa un sencillo **String** para encontrar un driver disponible que pueda manejar URLs en ese formato.

```
  public class RegistrationBean implements EntityBean{

    private transient EntityContext ctx;
    public String theuser, password;
    public String creditcard, emailaddress;
    public double balance;

  //Static class instantiation
    static {
          try{
          new pool.JDCConnectionDriver(
```

```
                    "COM.cloudscape.core.JDBCDriver",
                    "jdbc:cloudscape:ejbdemo",
                    "none", "none");
            }catch(Exception e){}
    }

    public Connection getConnection()
                        throws SQLException{
            return DriverManager.getConnection(
                                    "jdbc:jdc:jdcpool");
    }
}
```

Ozito

Características y Herramientas de Rendimiento

La nueva Máquina Virtual Java™ (JVM) tiene características para mejorar el rendimiento, y podemos usar un número de herramientas para incrementar el rendimiento de la aplicación o reducir el tamaño de los ficheros Class generados. Por eso las características y herramientas mejoran el rendimiento de nuestra aplicación con muy pocos o casi ningún cambio en en nuestra aplicación.

Caracterísitcas de la Máquina Virtual Java (JVM)

La plataforma Java® 2 ha presentamo muchas mejoras de rendimiento sobre versiones anteriores, incluyendo asignación más rápida de memoria, reducción del tamaño de las clases, mejorar la recolección de basura, monitores lineales y un JIT interno como estándard. Cuando usamo la nueva JVM de Java 2 nada más sacarla de la caja veremos una mejora, sin embargo para entendiendo como funciona el aumento de velocidad podemos ajustar nuestra aplicación para exprimir hasta el último bit de rendimiento.

Métodos en Línea

La versión Java 2 de la JVM automáticamente alinea métodos sencillo en el momento de la ejecución. En una JVM sin optimizar, cada vez que se llama a un método, se crea un nuevo marco de pila. La creacción de un nuevo marco de pila requiere recursos adicionales así como algún re-mapeo de la pila, el resultado final crear nuevos marcos de pila incurre en una pequeña sobrecarga.

Los métodos en línea aumenta el rendimiento reduciendo el número de llamadas a métodos que hace nuestro programa. La JVM alínea métodos que devuelven constantes o sólo acceden a campos internos. Para tomar ventaja de los métodos en línea podemos hacer una de estas dos cosas. Podemos hacer que un método aparezca atractivo para que la JVM lo ponga en línea o ponerlo manualmente en línea si no rompe nuestro modelo de objetos. La alineación manual en este contexto sólo significa poner el código de un método dentro del método que lo ha llamado.

El alineamiento automático de la JVM se ilustra con este pequeño ejemplo:

```
public class InlineMe {

    int counter=0;
```

```
    public void method1() {
        for(int i=0;i<1000;i++)
        addCount();
        System.out.println("counter="+counter);
    }

    public int addCount() {
        counter=counter+1;
        return counter;
    }

    public static void main(String args[]) {
        InlineMe im=new InlineMe();
        im.method1();
        }
}
```

En el estado actual, el método **addCount** no parece muy atractivo para el detector en línea de la JVM porque el método **addCount** devuelve un valor. Para ver si éste método está en línea compilamos el ejemplo con este perfil activado:

```
java -Xrunhprof:cpu=times InlineMe
```

Esto genera un fichero de salida **java.hprof.txt**. Los 10 primeros métodos se parecerán a esto:

```
CPU TIME (ms) BEGIN (total = 510)
                        Thu Jan 28 16:56:15 1999
rank self accum  count trace method
 1  5.88%  5.88%    1  25 java/lang/Character.
                          <clinit>
 2  3.92%  9.80% 5808  13 java/lang/String.charAt
 3  3.92% 13.73%    1  33 sun/misc/
                          Launcher$AppClassLoader.
                          getPermissions
 4  3.92% 17.65%    3  31 sun/misc/
                          URLClassPath.getLoader
 5  1.96% 19.61%    1  39 java/net/
                          URLClassLoader.access$1
 6  1.96% 21.57% 1000  46 InlineMe.addCount
 7  1.96% 23.53%    1  21 sun/io/
                          Converters.newConverter
 8  1.96% 25.49%    1  17 sun/misc/
                          Launcher$ExtClassLoader.
                          getExtDirs
 9  1.96% 27.45%    1  49 java/util/Stack.peek
```

```
   10  1.96% 29.41%    1  24 sun/misc/Launcher.<init>
```

Si cambiamos el método **addCount** para que no devuelva ningún valor, la JVM lo pondrá en línea durante la ejecución. Para amigable el código en línea reemplazamos el método **addCount** con esto:

```
public void addCount() {
        counter=counter+1;
}
```

Y ejecutamos el perfil de nuevo:

```
java -Xrunhprof:cpu=times InlineMe
```

Esta vez el fichero de salida **java.hprof.txt** debería parecer diferente. El método **addCount** se ha ido. Ha sido puesto en línea!

```
CPU TIME (ms) BEGIN (total = 560)
                        Thu Jan 28 16:57:02 1999
rank self  accum  count trace method
 1  5.36%  5.36%    1  27 java/lang/
                          Character.<clinit>
 2  3.57%  8.93%    1  23 java/lang/
                          System.initializeSystemClass
 3  3.57% 12.50%    2  47 java/io/PrintStream.<init>
 4  3.57% 16.07% 5808  15 java/lang/String.charAt
 5  3.57% 19.64%    1  42 sun/net/www/protocol/file/
                          Handler.openConnection
 6  1.79% 21.43%    2  21 java/io/InputStreamReader.fill
 7  1.79% 23.21%    1  54 java/lang/Thread.<init>
 8  1.79% 25.00%    1  39 java/io/PrintStream.write
 9  1.79% 26.79%    1  40 java/util/jar/
                          JarFile.getJarEntry
10  1.79% 28.57%    1  38 java/lang/Class.forName0
```

Sincronización

Los métodos y objetos sincronizados en Java han tenido un punto de rendimiento adicional como el mecanismo utilizado para implementar el bloqueo de este código usando un registro de monitor glogal que sólo fue enhebrado en algunas áreas como la búsqueda de monitores existentes. En la versión Java 2, cada thread tiene un registro de monitor y por eso se han eliminado mucho de esos cuellos de botellas.

Si hemos usado préviamente otros mecanimos de bloqueos porque el punto de rendimiento con los métodos sincronizados merece la pena re-visitar ese código y incorporarle los bloqueos en línea de Java 2.

En el siguiente ejemplo que está creando monitores para el bloque sincronizado podemos alcanzar un 40% de aumento de velocidad. El tiempo empleado fue 14ms usando JDK 1.1.7 y sólo 10ms con Java 2 en una máquina Sun Ultra 1.

```
class MyLock {

  static Integer count=new Integer(5);
  int test=0;

  public void letslock() {
     synchronized(count) {
        test++;
     }
  }
}

public class LockTest {

  public static void main(String args[]) {

     MyLock ml=new MyLock();
     long time = System.currentTimeMillis();

     for(int i=0;i<5000;i++ ) {
      ml.letslock();
     }
     System.out.println("Time taken="+
      (System.currentTimeMillis()-time));
  }
}
```

Java Hotspot

La máquina virtual Java HotSpot™ es la siguiente generación de implementaciones de la máquina virtual de Sun Microsystem. La Java HotSpot VM se adhiere a la misma especificación que la JVM de Java 2, y ejecuta los mismos bytecodes, pero ha sido rediseñada para lanzar nuevas tecnologías como los modelos de la optimización adaptativa y de recolección de basura mejorada para mejorar dramáticamente la velocidad del JVM.

Optimización Adaptativa

El Java Hotspot no incluye un compilador interno JIT pero en su lugar compila y pone métodos en línea que parecen ser los más utilizados en la aplicación. Esto significa que en el primer paso por los bytescodes Java son interpretados como si ni tubieramos un compilador JIT. Si el código aparece como un punto caliente de

nuestra aplicación el compilador Hotspot compilará los bytecodes a código nativo que es almacenado en un caché y los métodos en línea al mismo tiempo.

Una ventaja de la compilazión selectiva sobre un compilador JIT es que el compilador de bytes puede gastar más tiempo en generar alta optimización para áreas que podrían provocar la mayor optimización. el compilador también puede compiladr código que podría ejecutarse mejor en modo intérprete.

En el versiones anteriores de la Java HotSpot VM donde no era posible optimizar código que no estába actualmente en uso. El lado negativo de esto es que la aplicación estaba en una enorme bucle y el optimizador no podía compilar el código del área hasta que el bucle finalizara. Posteriores versiones de la Java Hotspot VM, usa un reemplazamiento en la pila, significando que el código puede ser compilado en código nativo incluso si está siendo utilizado por el intérprete.

Recolección de Basura Mejorada

El recolector de basura usado en el la Java HotSpot VM presenta varias mejoras sobre los recolectores de basura existentes. El primero es que el recolector se ha convertido en un recolector de basura totalmente seguro. Lo que esto significa es que el recoelcto sabe exactamente qué es una referencia y qué son sólo datos. El uso de referencias directas a objetos en el heap en una Java HotSpot VM en lugar de usar manejadores de objetos. Este incremento del conocimiento significa que la fragmentación de memoria puede reducirse con un resultado de una huella de memoria más compacta.

La segunda mejora es el uso de cópiado generacional. Java crea un gran número de objetos en la pila y frecuentemente estos objetos tenían una vida muy corta. Reemplazado los objetos creados recientemente por un cubo de memoria, esperando a que el cubo se lene y luego sólo copiando los objetos vivos restantes a una nuevo área del bloque de memoria que el cubo puede liberar en un bloque. Esto significa que la JVM no tiene que buscar un hueco para colocar cada nuevo objeto en la pila y significa que se necesita manejar secciones de memoria más pequeñas de una vez.

Para objetos viejos el recolector de basura hace un barrido a través del hepa y compacta los huecos de los objetos muertos directamente, eliminando la necesidad de una lista libre usada en algoritmos de recolección de basura anteriores.

El tercer área de mejora es eliminar la percepción de pausar en la recolección de basura escalonando la compactaciónde grandes objetos liberados en pequeños grupos y compactándolos de forma incremental.

Sincronización Rápida de Threads

La Java HotSpot VM also mejora el código de sincronización existente. Los bloques y métodos sincronizados siempren representan una sobrecarga cuando se ejecutan en una JVM. El Java HotSpot implementa los propios puntos de entrada y salida del monitor de sincroniación y no dependen del Sistema Operativo local para

proporcionar esta sincronización. Este resultado es un gran aumento de la velocidad especialmente en las frecuentes aplicaciones GUI sincronizadas.

Compiladores Just-In-Time

La herramienta más sencilla para mejorar el rendimiento de nuestra aplicación el compilador Just-In-Time (JIT). Un JIT es un generador de código que convierte los bytecodes Java en código nativo de la máquina. Los programas Java invocados con un JIT generalmente se ejecutan más rápidos que cuando se ejecutan en bytecodes por el intérprete. La Java Hotspot VM elimina la necesidad de un compilador JIT en muchos casos, sin embargo podrían utilizar el compilador JIT en versiones anteriores.

El compilador JIT se puso disponible como una actualización de rendimiento en la versión Java Development Kit (JDK™) 1.1.6 y ahora es una herramienta estándard invocada siempre qu eusamos el intérprete **java** en la versión de la plataforma Java 2. Podemos desactivar el uso del compilador JIT usando la opción **-Djava.compiler=NONE** en la JVM.

¿Cómo Funcionan los Compiladores JIT?

Los compiladores JIT se suministran como librerías nativas dependientes de la plataforma. Si xiste la librería del compilador JIT, la JVM inicializa el JNI (Java Native Interface) para llamar a las funciones JIT disponibles en la librería en lugar de su función equivalente del intérprete.

Se usa la clase **java.lang.Compiler** para cargar la librería nativa y empezar la inicialización dentro del compilador JIT.

Cuando la JVM llama a un método Java, usa un método llamante como especificado en el bloque método del objeto class cargado. La JVM tiene varios métodos llamantes, por ejemplo, se utiliza un llamante diferente si el método es sincronizado o si es un método nativo.

El compilador JIT usa su propio llamante. Las versión de Sun chequean el bit de aceso al método por un valor **ACC_MACHINE_COMPILED** para notificarle al intérprete que el código de esté método ya está compilado y almacenado en las clases cargadas.

¿Cuando el se compilado el código JIT?

Cuando se llama a un método por primera vez el compilador JIT compilad el bloque del método a código nativo y lo almacena en un bloque de código.

Una vez que el código ha sido compilado se activa el bit **ACC_MACHINE_COMPILED**, que es usado en la plataforma Sun.

¿Cómo puedo ver lo que está haciendo el compilador JIT?

La variable de entorno **JIT_ARGS** permite un control sencillo sobre el compilador JIT en Sun Solaris. Hay dos valores útiles **trace** y **exclude(list)**. Para excluir los métodos del ejemplo **InlineMe** un mostrar un seguimiennto seleccionamos **JIT_ARGS** de esta forma:

```
Unix:
export JIT_ARGS="trace exclude(InlineMe.addCount
                               InlineMe.method1)"

$ java InlineMe
Initializing the JIT library ...
DYNAMICALLY COMPILING java/lang/System.getProperty
                                    mb=0x63e74
DYNAMICALLY COMPILING java/util/Properties.getProperty
                                    mb=0x6de74
DYNAMICALLY COMPILING java/util/Hashtable.get
                                    mb=0x714ec
DYNAMICALLY COMPILING java/lang/String.hashCode
                                    mb=0x44aec
DYNAMICALLY COMPILING java/lang/String.equals
                                    mb=0x447f8
DYNAMICALLY COMPILING java/lang/String.valueOf
                                    mb=0x454c4
DYNAMICALLY COMPILING java/lang/String.toString
                                    mb=0x451d0
DYNAMICALLY COMPILING java/lang/StringBuffer.<init>
                                    mb=0x7d690
 <<<< Inlined java/lang/String.length (4)
```

Observa que los métodos en línea como **String.length** está exentos. El metodo **String.length** también es un método especial y es normalmente compilado en un atajo de bytecodes interno para el intérprete java. Cuando usamos el compilador JIT estás optimizaciones proporcionadas por el intérprete Java son desactivadas para activar el compilador JIT para entender qué método está siendo llamado.

¿Cómo Aprovechar la Ventaja del Compilador JIT?

Lo primero a recordar es que el compilador JIT consigue la mayoría del aumento de velocidad la segunda vez que llama a un método. El compilador JIT compila el método completo en lugar de intérpretarlo línea por línea que también puede ser una ganancia de rendimiento cuando se ejecuta una aplicación el JIT activado. Esto significa que si el código sólo se llama una vez no veremos una ganancia de rendimiento significante. El compilador JIT también ignora los constructores de las clases por eso si es posible debemos mantener al mínimo el código en los constructores.

El compilador JIT también consigue una ganancias menores de rendimiento al no prechequear ciertas condiciones Java como punteros **Null** o excepciones de array fuera de límites. La única forma de que el compilador JIT conozca una excepción de puntero null es mediante una señal lanzada por el sistema operativo. Como la señal viene del sistema operativo y no de la JVM, nuestro programa mejora su rendimiento. Para asegurarnos el mejor rendimiento cuando se ejecuta una aplicación con el JIT, debemos asegurarnos de que nuestro código está muy limpio y sin errores como excepciones de punteros null o arrays fuera de límites.

Podríamos querer desactivar el compilador JIT su estámos ejecutando la JVM en modo de depuración remoto, o si queremos ver los números de líneas en vez de la etiqueta **(Compiled Code)** en nuestos seguimientos de pila. Para desactivar el compilador JIT, suministramos un nombre no válido o un nombre en blanco para el compilador JIT cuando invoquemos al intérprete. Los siguientes ejemplos muestran el comando **javac** para compilar el código fuente en bytecodes, y dos formas del comando **java** para invocar al intérprete sin el compilador JIT:

```
javac MyClass.java
java -Djava.compiler=NONE MyClass
```

o

```
javac MyClass.java
java -Djava.compiler="" MyClass
```

Herramientas de Terceras Partes

Hay otras herramientas disponibles incluidas aquellas que reducen el tamaño de los ficheros class generados. El fichero class Java contiene un área llamada almacen de constantes. Este almacen de constantes mantiene una lista de strings y otra información del fichero class para referencias. Unas de las piezas de información disponibles en el almacen de constantes son los nombres de los métodos y campos.

El fichero class se refiere a un campo de la clase como a una referencia a un entrada en el almacen de constantes. Esto significa que mientras las referencias permanezcan iguales no importa los valores almacenados en el almacen de constantes. Este conocimiento es explotado por varias herramientas que reescriben los nombres de los campos y de los métodos en el almacen de constantes con nombres recortardos. Esta técnica puede reducir el tamaño del fichero class en un porcentaje significante con el beneficio de que un fichero class más pequeño significa un tiempo de descarga menor.

Análisis de Rendimiento

Otra forma de aumentar el rendimiento es con ánalisis de rendimiento. Los análisis de rendimientos buscan las ejecución del programa apuntar donde podrían estar los cuellos de botella y otros problemas de rendimiento como los picos de memoria. Una vez que sables donde están los puntos de problemas potenciales podemos cambiar nuestro código para eliminar o reducir su impacto.

Perfiles

Las Máquinas Vituales Java™ (JVMs) han tenido la habilidad de proporcionar sencillos informes de perfiles desde Java Development Kit (JDK™) 1.0.2. Sin embargo, la información que ellos proporcionaban estaban limitadas a una lista de los métodos que un programa había llamado.

La plataforma Java® 2 proporciona muchas más capacidades de perfilado que las anteriormente disponibles y el análisis de estos datos generado se ha hecho más fácil por la emergencia de un "Heap Analysis Tool" (HAT). Esta herramienta, como implica su nombre, nos permite analizar los informes de perfiles del heap. El heap es un bloque de memoria que la JVM usa cuando se está ejecutando. La herramienta de análisis de heap nos permite generar informes de objetos que fueron usado al ejecutar nuestra aplicación. No sólo podemos obtener un listado de los métodos llamados más frecuentemente y la memoria usada en llamar a esos métodos, pero también podemos seguir los picos de memeoria. Los picos de memoria pueden tener un significante impacto en el rendimiento.

Analizar un Programa

Para analizar el programa **TableExample3** incluido en el directorio **demo/jfc/Table** de la plataforma Java 2, necesitamos generar un informe de perfil. El informa más sencillo de generar es un perfil de texto. Para generarlo, ejecutamos la aplicación el parámetro **-Xhprof**. En la versión final de la plataforma Java 2, esta opción fue renombrada como **-Xrunhprof**. Para ver una lista de opciones actualmente disponibles ejecutamos el comando:

```
java -Xrunhprof:help
Hprof usage: -Xrunhprof[:help]|[<option>=<value>, ...]
```

Nombre de Opción y Valor	Descripción	Por Defecto
---	---	---

heap=dump|sites|all	**heap profiling**	**all**
cpu=samples|times|old	**CPU usage**	**off**
monitor=y|n	**monitor contention**	**n**
format=a|b	**ascii or binary output**	**a**
file=<file>	**write data to file**	**java.hprof(.txt for ascii)**
net=<host>:<port>	**send data over a socket**	**write to file**
depth=<size>	**stack trace depth**	**4**
cutoff=<value>	**output cutoff point**	**0.0001**
lineno=y|n	**line number in traces**	**y**
thread=y|n	**thread in traces?**	**n**
doe=y|n	**dump on exit?**	**y**

```
Example: java -Xrunhprof:cpu=samples,file=log.txt,
                                depth=3 FooClass
```

La siguiente invocación crea una fichero de texto que podemos ver sin la herramienta de análisis de heap llamado **java.hprof.txt** cuando el programa genera un seguimiento de pila o sale. Se utiliza una invocación diferente para crear un fichero binario para usarlo con la herramienta de análisis de heap:

```
java -Xrunhprof TableExample3

d:\jdk12\demo\jfc\Table> java -Xrunhprof TableExample3
Dumping Java heap ... allocation sites ... done.
```

La opción de perfil literalmente hace un diario de cada objeto creado en el heap, por incluso cuando arrancamos y paramos el pequeño progeama **TableExample3** resulta un ficheo de informe de cuatro megabytes. Aunque la herramienta de análisis de heap usa una versión binaria de este fichero y proporciona un sumario, hay algunas cosas rápidas y fáciles que podemos aprender desde el fichero de texto sin usar la herramienta de análisis de heap.

Nota: Para listar todas las opciones disponibles, usamos **java -Xrunhprof:help**

Ver el Fichero de Texto

Elegimos un fichero que pueda manejar grandes ficheros y vamos hasta el final del fichero. Podría haber cientos o miles de líneas, por eso un atajo es buscar las palabras **SITES BEGIN**. Veríamos una lista de línea que empezarían un tango creciente de números seguido por dos números de porcentaje. La primera entrada en la lista sería similar a este ejemplo:

```
SITES BEGIN (ordered by live bytes)
                  Sun Dec 20 16:33:28 1998
```

rank	percent self	percent accum	live bytes	live objs	alloc'ed bytes	alloc'ed objs	stack trace	class name
1	55.86%	55.86%	826516	5	826516	5	3981	[S

La notación **[S** al final de la última línea indica que es la primera entrada de un array de **short**, un tipo primitivo. Esto es lo esperado con aplicaciones (AWT). Los primeros cinco contadores bajo la cabecera **objs** significa que actualmente hay cinco de esos arrays, y sólo ha habido cinco durante el tiempo de vida de esta aplicación, y han ocupado 826516 bytes. La referencia clase de este objeto es el valor listado bajp **stack trace**. Para encontrar donde se creo esté objeto en este ejmplo, buscamos **TRACE 3981**. Veremos esto:

```
TRACE 3981:
java/awt/image/DataBufferUShort.<init>(
                            DataBufferUShort.java:50)
java/awt/image/Raster.createPackedRaster(
                                  Raster.java:400)
java/awt/image/DirectColorModel.
      createCompatibleWritableRaster(
                              DirectColorModel.java:641)
sun/awt/windows/WComponentPeer.createImage(
                              WComponentPeer.java:186)
```

El código **TableExample3** selecciona un **scrollpane** de 700 por 300. Cuando miramos el fuente de **Raster.java**, qu está en el fichero **src.jar**, encontraremos estas sentencias en la línea 400:

```
case DataBuffer.TYPE_USHORT:
       d = new DataBufferUShort(w*h);
       break;
```

Los valores **w** y **h** son la anchura y altura de la llamada a **createImage** que arranca en **TRACE 3981**. El constructor **DataBufferUShort** crea un array de **shorts**:

```
data = new short[size];
```

donde **size** es **w*h**. Por eso, en teoría debería hacer una entrada en el array para 210000 elementos. Buscamos una enteada por cada ejemplarización de esta clase buscando por **trace=3981**. Una de las cinco entradas se parecerá a esto:

```
OBJ 5ca1fc0 (sz=28, trace=3979,
      class=java/awt/image/DataBufferUShort@9a2570)
 data 5ca1670
```

```
bankdata 5ca1f90
offsets 5ca1340
ARR 5ca1340 (sz=4, trace=3980, nelems=1,
                              elem type=int)
ARR 5ca1670 (sz=420004, trace=3981, nelems=210000,
                              elem type=short)
ARR 5ca1f90 (sz=12, trace=3982, nelems=1,
                              elem type=[S@9a2d90)
[0] 5ca1670
```

Podemos ver que los valores de los datos de estas referencias de imagen en un array **5ca1670** que devuelve un alista de 210000 elementos **short** de tamaño 2. Esto significa qu este array usa 420004 bytes de memoria.

De este dato podemos concluir que el programa **TableExample3** usa cerca de 0.5Mb para mapear cada tabal. Si la aplicación de ejemplo se ejecuta en una máquina con poca memoria, debemos asegurarnos de que no mantenemos referencias a objetos geandes o a imágenes que fueron construidas con el método **createImage**.

La Herramienta de Análisis de Heap

Esta herramienta puede analizar los mismos datos que nosotros, pero requere un fichero de informe binario como entrada. Podemos generar un fichero de informa binario de esta forma:

```
java -Xrunhprof:file=TableExample3.hprof,format=b
                                        TableExample3
```

Para generar el informe binario, cerramos la ventana **TableExample3**. El fichero de informe binario **TableExample3.hprof** se crea al salir del programa. La Herramienta de Análisis de Heap arranca un servidor HTTP que analiza el fichero de perfil binario y muestra el resultado en un HTML que podemos ver en un navegador.

Podemos obtener una copia de la Herramienta de Análisis de Heap de la site java.sun.com. Una vez instalado, ejecutamos los scripts shell y batch en el directorio **bin** instalado para poder ejecutar el servidor de la Herramienta de Análisis de Heap de esta forma:

```
>hat TableExample3.hprof
Started HCODEP server on port 7000
Reading from /tmp/TableExample3.hprof...
Dump file created Tue Jan 05 13:28:59 PST 1999
Snapshot read, resolving...
Resolving 17854 objects...
Chasing references,
```

```
          expect 35 dots.....................
Eliminating duplicate
              references.....................
Snapshot resolved.
Server is ready.
```

La salida de arriba nos dice que nuestro servidor HTTP se ha arrancado en el puerto 7000. Para ver este informe introducimos la URL http://localhost:7000 o http://your_machine_name:7000 en nuestro navegador Web. Si tenemos problema en arrancar el servidor usando el script, podemos alternativamente ejecutar la aplicación incluyendo el fichero de clases **hat.zip** en nuestro **CLASSPATH** y usar el siguiente comando:

```
java hat.Main TableExample3.hprof
```

La vista del informe por defecto contiene una lista de todas las clases. En la parte de abajo de está página inicial están las dos opciones básicas de informes:

```
Show all members of the rootset
Show instance counts for all classes
```

Si seleccionamos el enlace **Show all members of the rootset**, veremos un alista de las siguientes referencias porque estas referencias apuntan a picos potenciales de memoria.

```
Java Static References
Busy Monitor References
JNI Global References
JNI Local References
System Class References
```

Lo que vemos aquí son ejemplares en la aplicación que tienen referencias a objetos que tienen un riesgo de no se recolectados para la basura. Esto puede ocurrir algunas veces en el caso del JNI su se asigna memoria para un objeto, la memoria se deja para que la libere el recolector de basura, y el recolector de basura no teine la información que necesita para hacerlo. En esta lista de referencias, estamos principalmente interesados en un gran número de referencias a objetos o a objetos de gran tamaño.

El otro informe clave es el **Show instance counts for all classes**. Este lista los números de llamadas a un método particular. Los objetos array **String** y **Character**, **[S** y **[C**, están siempre en la parte superior de esta lista, pero algunos objetos son un poco más intrigantes. ¿Por qué hay 323 ejemplares de **java.util.SimpleTimeZone**, por ejemplo?

```
5109 instances of class java.lang.String
5095 instances of class [C
2210 instances of class java.util.Hashtable$Entry
```

```
968 instances of class java.lang.Class
407 instances of class [Ljava.lang.String;
323 instances of class java.util.SimpleTimeZone
305 instances of class
      sun.java2d.loops.GraphicsPrimitiveProxy
304 instances of class java.util.HashMap$Entry
269 instances of class [I
182 instances of class [Ljava.util.Hashtable$Entry;
170 instances of class java.util.Hashtable
138 instances of class java.util.jar.Attributes$Name
131 instances of class java.util.HashMap
131 instances of class [Ljava.util.HashMap$Entry;
130 instances of class [Ljava.lang.Object;
105 instances of class java.util.jar.Attributes
```

Para obtener más información sobre los ejemplares **SimpleTimeZone**, pulsamos sobre el enlace (la línea que empieza por 323). Esto listará las 323 referencias y calculará cuánta memoria ha sido utilizada. en este ejemplo, se han utilizado 21964 bytes.

```
Instances of java.util.SimpleTimeZone

class java.util.SimpleTimeZone

java.util.SimpleTimeZone@0x004f48c0 (68 bytes)
java.util.SimpleTimeZone@0x003d5ad8 (68 bytes)
java.util.SimpleTimeZone@0x004fae88 (68 bytes)
.....
Total of 323 instances occupying 21964 bytes.
```

Si pulsamos sobre uno de estos ejemplares **SimpleTimeZone**, veremos donde fue asignado este objeto.

```
Object allocated from:

java.util.TimeZoneData.<clinit>(()V) :
      TimeZone.java line 1222
java.util.TimeZone.getTimeZone((Ljava/lang/String;)
      Ljava/util/TimeZone;) :
      TimeZone.java line (compiled method)
java.util.TimeZone.getDefault(
      ()Ljava/util/TimeZone;) :
      TimeZone.java line (compiled method)
java.text.SimpleDateFormat.initialize(
      (Ljava/util/Locale;)V) :
      SimpleDateFormat.java line (compiled method)
```

En este ejemplo el objeto fue asignado desde **TimeZone.java**. El fichero fuente de este fichero están el fichero estándard **src.jar**, y examinando este fichero, podemos ver que de hehco hay cerca de 300 de estos objetos en memoria.

```
static SimpleTimeZone zones[] = {
 // The following data is current as of 1998.
 // Total Unix zones: 343
 // Total Java zones: 289
 // Not all Unix zones become Java zones due to
 // duplication and overlap.
 //----------------------------------------
 new SimpleTimeZone(-11*ONE_HOUR,
                    "Pacific/Niue" /*NUT*/),
```

Desafortunadamente, no tenemos control sobre la memoria usada en este ejemplo, porque es asignada cuando el programa hizo la primera solicitud al timezone por defecto. Sin embargo, esta misma técnica puede aplicarse para analizar nuestra propia aplicación donde probablemente podríamos hacer algunas mejoras.

¿Dónde Gasta el Tiempo la Aplicació?

De nuevo, podemos usar el parámetro **-Xrunhprof** para obtener información sobre el tiempo que gasta la aplicación procesando un método particular.

Podemos usar una o dos opciones de perfil de CPU para conseguir esto. La primera opción es **cpu=samples**. Esta opción devuelve el resultado de un muestreo de ejecución de threads de la Máquina Virtual Java con un conteo estadístico de la frecuencia de ocurrencia con que se usa un método particular para encontrar secciones ocupadas de la aplicación. La segunda opción es **cpu=times**, que mide el tiempo que tardan los métodos individuales y genera un ranking del porcentaje total del tiempo de CPU ocupado por la aplicación.

Usando la opción **cpu=times**, deberíamos ver algo como esto al final del fichero de salida:

```
CPU TIME (ms) BEGIN (total = 11080)
                        Fri Jan  8 16:40:59 1999
rank   self   accum   count  trace   method
 1   13.81%  13.81%       1   437   sun/
    awt/X11GraphicsEnvironment.initDisplay
 2    2.35%  16.16%       4   456   java/
    lang/ClassLoader$NativeLibrary.load
 3    0.99%  17.15%      46   401   java/
    lang/ClassLoader.findBootstrapClass
```

Si constrastamos esto con la salida de **cpu=samples**, veremos la diferencia entre la frecuencia de ejecuciónde un método durante la ejecución de la aplicación

comparada con el tiempo que tarda ese método.

```
CPU SAMPLES BEGIN (total = 14520)
                   Sat Jan 09 17:14:47 1999
rank  self   accum   count  trace   method
 1    2.93%  2.93%   425    2532    sun/
    awt/windows/WGraphics.W32LockViewResources
 2    1.63%  4.56%   237     763    sun/
    awt/windows/WToolkit.eventLoop
 3    1.35%  5.91%   196    1347    java/
    text/DecimalFormat.<init>
```

El método **W32LockView**, que llama a una rutina de bloqueo de ventana nativa, se llama 425 veces. Por eso cuando aparecen en los threads activos porque también toman tiempo para completarse. En contraste, el método **initDisplay** sólo se le llama una vez, pero es el método que tarda más tiempo en completarse en tiempo real.

Herramientas de Rendimiento de Sistema Operativo

Algunas veces los cuellos de botella del rendimiento ocurren al nivel del sistema operativo. Esto es porque la JVM depende en muchas operacioens de las librerías del sistema operativo para funcionalidades como el acceso a disco o el trabajo en red. Sin embargo, lo que ocurre después de que la JVM haya llamado a estas librerías va más alla de las herramientas de perfilado de la plataforma Java.

Aquí hay una lista de herramietnas que podemos usar para analizar problemas de rendimiento en algunos sistemas operativos más comunies.

Plataforma Solaris

System Accounting Reports, **sar**, informa de la actividad del sistema en términos de I/O de disco, actividad del programa de usuario, y actividad a nivel del sistema. Si nuestra aplicación usa una cantidad de memoria excesiva, podría requerir espacio de intercambio en disco, por lo que veriamos grandes porcentajes en la columna WIO. Los programas de usuario que se quedan en un bucle ocupado muestran un alto porcentaje en la columna user:

```
developer$ sar 1 10

SunOS developer 5.6 Generic_105181-09 sun4u
                                      02/05/99

11:20:29    %usr    %sys    %wio   %idle
11:20:30      30       6       9      55
11:20:31      27       0       3      70
```

```
11:20:32      25       1       1      73
11:20:33      25       1       0      74
11:20:34      27       0       1      72
```

El comando **truss** sigue y guarda los detalles de cada llamada al sistema por la JVM al kernel Solaris. Un forma común de usar **truss** es:

```
truss -f -o /tmp/output -p <process id>
```

El parámetro **-f** sigue cualquier proceso hijo que haya creado, el parámetro **-o** escribe la salida en el fichero nombrado, y el parámetro **-p** sigue un programa en ejecución desde sis ID de proceso. De forma alternativa podemos reemplazar **-p** <process id> con la JVM, por ejemplo:

```
truss -f -o /tmp/output java MyDaemon
```

El **/tmp/output** es usado para almacenar la salida de **truss**, lo que se debería parecer a esto:

```
15573:  execve("/usr/local/java/jdk1.2/solaris/
                bin/java", 0xEFFFF2DC,
                0xEFFFF2E8)  argc                    = 4
15573:  open("/dev/zero", O_RDONLY)                  = 3
15573:  mmap(0x00000000, 8192,
             PROT_READ|PROT_WRITE|PROT_EXEC,
             MAP_PRIVATE, 3, 0) = 0xEF7C0000
15573:  open("/home/calvin/java/native4/libsocket.so.1",
              O_RDONLY) Err#2 ENOENT
15573:  open("/usr/lib/libsocket.so.1",
              O_RDONLY)                              = 4
15573:  fstat(4, 0xEFFFEF6C)                         = 0
15573:  mmap(0x00000000, 8192, PROT_READ|PROT_EXEC,
              MAP_SHARED, 4, 0) = 0xEF7B00 00
15573:  mmap(0x00000000, 122880, PROT_READ|PROT_EXEC,
              MAP_PRIVATE, 4, 0) = 0xEF7 80000
15573:  munmap(0xEF78E000, 57344)                    = 0
15573:  mmap(0xEF79C000, 5393,
              PROT_READ|PROT_WRITE|PROT_EXEC,
              MAP_PRIVATE|MAP_FIXED, 4, 49152)
              = 0xEF79C000
15573:  close(4)                                     = 0
```

En la salida de **truss**, buscamos los ficheros que fallaran al abrirlos debido a problemas de acceso, como un error **ENOPERM**, o un error de fichero desaparecido **ENOENT**. También podemos seguir los datos leidos o escrito con los parámetros de **truss**: **-rall** para seguir todos los datos leídos, o **-wall** para seguir todos los datos escritos por el programa. Con estos parámetros, es posible analizar datos enviados a través de la red o a un disco local.

Plataforma Linux

Linux tiene un comando trace llamado **strace**. Sigue las llamadas del sistema al kernel Linux. Este ejemplo sigue el ejemplo **SpreadSheet** del directorio demo del JDK:

```
$ strace -f -o /tmp/output
                      java sun.applet.AppletViewer
                      example1.html
$ cat /tmp/output

639   execve("/root/java/jdk117_v1at/java/
               jdk117_v1a/bin/java", ["java",
               "sun.applet.AppletViewer ",
               "example1.html"], [/* 21 vars */]) = 0
639   brk(0)                                  = 0x809355c
639   open("/etc/ld.so.preload", O_RDONLY)       = -1
        ENOENT (No such file or directory)
639   open("/etc/ld.so.cache", O_RDONLY)          = 4
639   fstat(4, {st_mode=0, st_size=0, ...})       = 0
639   mmap(0, 14773, PROT_READ, MAP_PRIVATE,
              4, 0) = 0x4000b000
639   close(4)                                    = 0
639   open("/lib/libtermcap.so.2", O_RDONLY)      = 4
639   mmap(0, 4096, PROT_READ, MAP_PRIVATE,
              4, 0) = 0x4000f000
```

Para obtener información del sistema similar al comando **sar** de Solaris, lee los contenidos del fichero **/proc/stat**. El formato de este fichero se describe en las páginas del manual **proc**. Miramos la línea **cpu** para obtener la hora del sistema de usuario:

```
      cpu  4827 4 1636 168329
```

En el ejemplo anterior, la salida **cpu** indica 48.27 segundos de espacio de usuario, 0.04 de prioridad máxima, 16.36 segundos procesando llamadas al sistema, y 168 segundos libre. Esta es una ejecución total, las entradas para cada proceso están disponibles en **/proc/<process_id>/stat**.

Plataforma Windows95/98/NT

No hay herramientas de análisis de rendimiento estándard incluidas en estas plataformas, pero si hay herramientas de seguimiento disponibles mediante recursos freeware o shareware como http://www.download.com .

Análisis de memoria: Memory meter

Análisis de Red: Traceplus

Ozito

Caché en Aplicaciones Cliente/Servidor

El caché es una de las primera técnicas usadas para aumetnar el rendimiento de navegadores y servidores web. El caché del navegador hace innecesarios los bloqueos de red porque una copia reciente del fichero se mantiene en el caché local, y el caché del servidor reduce el coste de la carga de ficheros desde disco para cada petición. Esta sección explica cómo podes usar el caché de forma similar para mejorar el rendimiento en muchas aplicaciones cliente/servidor escritas en lenguaje Java™.

El API **java.util.Collections** disponible en el SDK Java® 2 hace sencilla la implementación del caché. Este API proporciona la clase **HashMap**, que funciona bien para cachear un objeto, y la clase **LinkedList**, que funciona bien en combinaciones con la clase **HashMap** para cachear muchos objetos.

Caché de un Objeto

Un objeto **HashMap** almacena objetos en una pareja clave valor. cuando ponemos un datp en un **HashMap**, le asignamos una clave y luego usamos esa clave para recuperar el dato.

Un objeto **HashMap** es muy similar a un **Hashtable** y puede ser usado para mantener una copia temporal de resultados generados préviamente. Los objetos mantenidos en el caché **HashMap** podría, por ejemplo, ser una lista de subastas completadas.

En este caso, los resultados de una consulta JDBC podrían solicitarse cientos de veces en un segundo por personas que están esperando conocer la puja más alta, pero la lista de resultados completa sólo cambia una vez por minuto cuando se ompleta una subasta. Podemos escribir nuestro programa para recuperar los objetos que no han cambiado desde el caché de resultados en vez de solicitar a la base de datos cada vez y obtener un significante aumento de rendimiento.

Este ejemplo de código ejecuta una consulta a la base de datos por cada minuto, y devuelve copias cacheadas para las solicitudes que vienen entre consultas.

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

class DBCacheRecord {
  Object data;
  long time;

  public DBCacheRecord(Object results, long when) {
    time=when;
    data=results;
  }
  public Object getResults() {
        return data;
  }
  public long getLastModified() {
        return time;
  }
}

public class DBCache {
  Map cache;

  public DBCache() {
    cache = new HashMap();
  }

  public Object getDBData(String dbcommand) {
    if(!cache.containsKey(dbcommand)) {
        synchronized(cache) {
          cache.put(dbcommand, readDBData(dbcommand));
        }
     } else {
       if((new Date().getTime() ) -
         ((DBCacheRecord)cache.get(
                dbcommand)).getLastModified()>=1000) {
         synchronized(cache) {
           cache.put(dbcommand, readDBData(dbcommand));
```

```
            }
          }
        }
        return ((DBCacheRecord)cache.get(
           dbcommand)).getResults();
    }

    public Object readDBData(String dbcommand) {

  /*Insert your JDBC code here For Example:
    ResultSet results=stmt.executeQuery(dbcommand);
  */
      String results="example results";
      return(new DBCacheRecord(results,new
                               Date().getTime()));

    }

    public static void main(String args[]) {
      DBCache d1=new DBCache();
      for(int i=1;i<=20;i++) {
          d1.getDBData(
            "select count(*) from results where
            TO_DATE(results.completed) <=SYSDATE");
      }
    }
  }
```

Cache de Muchos Objetos

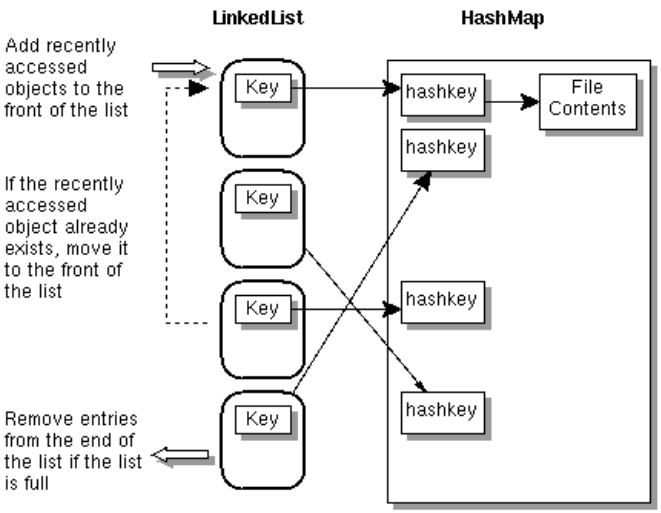
Algunas veces queremos cachear más de un objeto. Por ejemplo, podríamos querer mantener los ficheros accedidos más recientemente en el caché de un servidor web. Si usamos un objeto **HashMap** para un propósito como este, continuará creciendo y usando mucha memoria.

Si nuestra máquina tiene una gran cantidad de memoria y sólo un pequeño número de objetos que cachear entonces un creciente **HashMap** podría no ser un problema. Sin embargo, si estamos intentar cachear muchos objetos entonces podríamos queres sólo mantener los objetos más recientes en el caché proporcionando el mejor uso de la mémoria de la máquina. Podemos combinar un objeto **HashMap** con un **LinkedList** para crear un caché llamado "Most Recently Used" (MRU).

Con un caché MRU, podemos situar una restricción sobre los objetos que permanecen en el caché, y por lo tanto, control sobre el tamaño del caché. Hay tres operaciones principales que puede realizar un caché MRU:

- Si el caché no está lleno, los nuevos objetos que no están en el caché se insertan en la parte superior de la lista.
- Si el caché no está lleno y el objeto a inserta ya está en el caché, se mueve a la parte superior del caché.
- Si el caché está lleno y se inserta un nuevo objeto, el último objeto del caché es eliminado y el nuevo objeto se pone en la parte superior de la lista.

Este diagrama muestra cómo trabajan juntos **LinkedList** y **HashMap** para implementar las operaciones descritas arriba.


LinkedList
HashMap
Add recently accessed objects to the front of the list
If the recently accessed object already exists, move it to the front of the list
Remove entries from the end of the list if the list is full
Key
Key
Key
Key
hashkey
hashkey
hashkey
hashkey
File Contents

Caché MRU con LinkedList y HashMap

El **LinkedList** proporciona el mecanismo de cola, y las entradas de la **LinkedList** contienen la clave de los datos en el **HashMap**. Para añadir una nueva entrada en la parte superior de la lista, se llama al método **addFirst**.

- Si la lista ya está llena, se llama al método **removeLast** y a entrada de datos también se elimina del **HashMap**.
- Si una entrada ya existe en la lista, se elimina con una llamada al método **remove** y se inserta en la parte superior de la lista con una llamada al método **addFirst**.

El API Collectios no implementa bloqueos, por eso si eliminados o añadimos entradas a objetos **LinkedList** o **HashMap**, necesitamos bloquear los accesos a esos objetos. También podemos usar un **Vector** o **ArrayList** para obtener el mismo resultado mostrado en el códido de abajo del **LinkedList**.

Este ejemplo de código usa un caché MRU para mantener un caché de ficheros cargados desde disco. Cuando se solicita un fichero, el programa chequea para ver si el fichero está en el caché. Si el fichero no está en el caché, el programa lee el fichero desde el disco y sitúa una copia en el caché al principio de la lista.

Si el fichero está en el caché, el programa compara la fecha de modificación del fichero y la entrada del caché.

- Si la entrada del caché es más vieja, el programa lee el fichero del disco, elimina la copia del caché, y sitúa una nueva copia en el caché en la parte superior del **LinkedList**.
- Si el fichero es más viejo que el caché, el programa obtiene el fichero del caché y mueve la copia del caché a la parte superior de la lista.

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

class myFile {
  long lastmodified;
  String contents;

  public myFile(long last, String data) {
    lastmodified=last;
    contents=data;
  }
  public long getLastModified() {
    return lastmodified;
  }
  public String getContents() {
    return contents;
  }
}

public class MRUCache {

  Map cache;
  LinkedList mrulist;
  int cachesize;

  public MRUCache(int max) {
    cache = new HashMap();
    mrulist= new LinkedList();
    cachesize=max;
  }

  public String getFile(String fname) {
    if(!cache.containsKey(fname)) {
      synchronized(cache) {
        if(mrulist.size() >=cachesize) {
          cache.remove(mrulist.getLast());
          mrulist.removeLast();
        }
        cache.put(fname, readFile(fname));
        mrulist.addFirst(fname);
      }
    } else {
      if((new File(fname).lastModified())>
        ((myFile)cache.get(fname)).getLastModified()) {
          synchronized(cache) {
```

```
            cache.put(fname, readFile(fname));
          }
        }
        synchronized(cache) {
          mrulist.remove(fname);
          mrulist.addFirst(fname);
        }
      }
        return ((myFile)cache.get(fname)).getContents();
    }

    public myFile readFile(String name) {
      File f = new File(name);
      StringBuffer filecontents= new StringBuffer();

      try {
        BufferedReader br=new BufferedReader(
                              new FileReader(f));
        String line;

        while((line =br.readLine()) != null) {
          filecontents.append(line);
        }
      } catch (FileNotFoundException fnfe){
        return (null);
      } catch ( IOException ioe) {
                return (null);
      }
        return (new myFile(f.lastModified(),
                filecontents.toString()));
    }

    public void printList() {
      for(int i=0;i<mrulist.size();i++) {
        System.out.println("item "+i+"="+mrulist.get(i));
      }
    }

    public static void main(String args[]) {

      // Number of entries in MRU cache is set to 10
      MRUCache h1=new MRUCache(10);
      for(int i=1;i<=20;i++) {
        // files are stored in a subdirectory called data
        h1.getFile("data"+File.separatorChar+i);
      }
        h1.printList();
    }
  }
```

Ozito

Desplegar la Aplicación Subasta

Con la aplicación subasta testeada, depurada y ajustada, estamos listos para desplegarla. Desplegarla implica unir todos los ficheros de la aplicación, moverlos a sus localizaciones, instalar el Java Plug-In para que los administradores de la subasta puedan ejecutar el applet Administration desde sus navegadores, e instalar el fichero de policía del applet Administration. El Java Plug-In es necesario porque el applet Administration está escrito con el Java Development Kit (JDK™) 1.2, pero los navegadores de los administradores podrían ejecutar versiones anteriores del software Java Runtime Environment™ (JRE).

Este capítulo explica como usar el formato de ficheros Java Archive (JAR) para unir los ficheros de la aplicación, y cómo instalar el Java Plug-In y un fichero de policia de seguridad para las plataformas Solaris™ y Win32 para ejecutar el applet Administration.

¿Tienes Prisa?

Esta tabla contiene enlaces directos a los tópicos específicos.

Tópico	Sección
Formato de Ficheros JAR	• Unir y Desplegar Ficheros HTML • Unir y Desplegar los Beans Enterprise • Unir y Desplegar el Applet Administration
Plataforma Solaris	• Obtener las Descargas • Extraer los Ficheros Descargados • Instalar Java Plug-In • Instalar las Mejoras del Java Plug-In • Instalar Netscape Communicator • Chequear la Instalación • Convertir Ficheros HTML • Fichero de Policía de Seguridad • Ejecutar el Applet Administration

Plataforma Win32	• Descargar e Instalar • Convertir Ficheros HTML • Ficheros de Policía de Seguridad • Ejecutar el Applet Administration

---

Formato de Ficheros JAR

El formato de ficheros Java JAR es un formato de compresión y empaquetado de ficheros y una herramienta para unir ficheros ejecutables con otros ficheros relacionados con la aplicación por eso pueden desplegarse en una sóla unidad. La aplicación de subasta, tiene tres unidades o ficheros para desplegar en tres diferentes localizaciones.

1. Los ficheros HTML que crean el interface de usuario de la aplilcación desplegado en una localización accesible bajo el servidor web.
2. Los Beans Enterprise desplegado en una localización interna accesible a nuestro servidor de JavaBeans Enterprise™.
3. El Applet Administration desplegado a una localización interna accesible para los administradores de la subasta donde es ejecutado por sus navegadores

Esta sección nos muestra cómo usar la herramienta **jar** unir y desplegar los ficheros de la aplicación.bundle and deploy the application files.

- Unir y Desplegar los Ficheros HTML
- Unir y Desplegar los Beans Enterprise
- Unir y Desplegar el Applet Administration

Unir y Desplegar los Ficheros HTML

Aquí hay una lista de ficheros HTML que crean el interface de usuario de la aplicación subasta:

- all.html
- close.html
- details.html
- index.html
- juggler.med.gif
- new.html
- registration.html
- search.html
- sell.html

Aquí está el comando **jar** que los une. Todo va en un sólo fichero. Este comando se ejecuta en el mismo directorio que los ficheros. Si lo ejecutamos desde otro directorio distinto tenemos que especificar el path completo o relativo según corresponda.

```
jar cvf HTML.jar all.html close.html details.html
```

```
index.html juggler.med.gif new.html
registration.html search.html sell.html
```

jar es el comando **jar**. Si tecleamos **jar** sin opciones, optenemos la siguiente pantalla de ayuda. Podemos ver de esta pantalla que las opciones **cf** del comando **jar** crean un nuevo fichero JAR llamando **HTML.jar** y pone la siguiente lista de ficheros en él. El nuevo fichero JAR se sitúa en el directorio actual.

```
kq6py% jar
Usage: jar {ctxu}[vfm0M] [jar-file] [manifest-file]
                  [-C dir] files ...
Options:
  -c  create new archive
  -t  list table of contents for archive
  -x  extract named (or all) files from archive
  -u  update existing archive
  -v  generate verbose output on standard output
  -f  specify archive file name
  -m  include manifest information from specified
        manifest file
  -0  store only; use no ZIP compression
  -M  Do not create a manifest file for the entries
  -C  change to the specified directory and include
        the following file
If any file is a directory then it is processed
recursively. The manifest file name and the archive
file name needs to be specified in the same order
the 'm' and 'f' flags are specified.

Example 1: to archive two class files into an archive
     called classes.jar:
     jar cvf classes.jar Foo.class Bar.class
Example 2: use an existing manifest file 'mymanifest'
     and archive all the files in the foo/ director
     into 'classes.jar':
     jar cvfm classes.jar mymanifest -C foo/ .
```

Para desplegar los ficheros HTML, todo lo que tenemos que hacer es mover el fichero **HTML.jar** a un directorio públicamente accesible bajo el servidor web y descomprimirlo:

```
jar xf HTML.jar
```

Nota: Si hemos incluido un path completo o relativo cuando hemos añadido los ficheros al fichero JAR, los ficheros se situarán en la misma estructura de directorio cuando los desempaquetemos.

Unir y Desplegar los Beans Enterprise

Algunoes servidores JavaBeans Enterprise pueden crear el fichero JAR por nosotros. Sin embargo, si el nuestro no lo hace o si que sólo queremos aprender a hacerlo, esta sección describe los pasos.

Aquí están los ficheros del lado del servidor que necesitamos para desplegar los Beans de Enterprise. Esta lista está tomanda de la aplicación de subasta original descrita en el Capítulo 2: Código de la Aplicación Subasta antes de cualquier modificación hecha para hacer los Beans Enterprise controlados por contenedor. Observa la inclusión del descriptor de desarrollo, y de las clases stub y skel del contenedor-generado.

Paquete auction

Aquí están los ficheros de aplicación del paquete **auction** que crean el servlet **AuctionServlet** y el Bean Enterprise **AuctionItemBean**. Como todos ellos van a ser instalados en un directorio **auction** accesible del servidor de producción JavaBeans Enterprise, los unimos todos juntos para que puedanser desempaquetados en un paso en el directorio destino y situados en el subdirectorio **acution**.

- auction.AuctionServlet.class
- auction.AuctionItem.class
- auction.AuctionItemBean.class
- auction.AuctionItemHome.class
- auction.AuctionItemPK.class
- auction.DeploymentDescriptor.txt
- AuctionItemBeanHomeImpl_ServiceStub.class
- WLStub1h1153e3h2r4x3t5w6e82e6jd412c.class
- WLStub364c363d622h2j1j422a4oo2gm5o.class
- WLSkel1h1153e3h2r4x3t5w6e82e6jd412c.class
- WLSkel364c363d622h2j1j422a4oo2gm5o.class

Aquí está cómo unirlos. Toda va en una línea línea, y el comando se ejecuta un directorio por encima de donde se encuentran los ficheros class.

Unix:

```
jar cvf auction.jar  auction/*.class
```

Win32:

```
jar cvf auction.jar  auction\*.class
```

Una vez que el fichero JAR se ha copiado en el directorio de destino para los Beans Enterprise, lo desempaquetamos de esta forma. La extracciónrea un directorio

auction con los ficheros class denrtro.

```
jar xv auction.jar
```

Paquete registration

Aquí están los ficheros de la aplicación en el paquete **registration** que crea el Bean Enterprise **Registration**.

- registration.Registration.class
- registration.RegistrationBean.class
- registration.RegistrationHome.class
- registration.RegistrationPK.class
- auction.DeploymentDescriptor.txt
- RegistrationBeanHomeImpl_ServiceStub.class
- WLStub183w4u1f4e70p6j1r4k6z1x3f6yc21.class
- WLStub4z67s6n4k3sx131y4fi6w4x616p28.class
- WLSkel183w4u1f4e70p6j1r4k6z1x3f6yc21.class
- WLSkel4z67s6n4k3sx131y4fi6w4x616p28.class

Aquí está como unirlo. Todo va en una línea y el comando se ejecuta un directorio por encima de donde se encuentran los ficheros class.

Unix:

```
jar cvf registration.jar  registration/*.class
```

Win32:

```
jar cvf registration.jar  registration\*.class
```

Una vez que el fichero JAR seha copiado al directorio de destino para los Beans Enterprise, los desempaquetamos de esta forma. La extracción crea un directorio **registration** con los fichero class dentro de él.

```
jar xv registration.jar
```

Paquete bidder

Aquí están los ficheros de la aplicación en el paquete **bidder** que crean el Bean Enterprise **Bidder**.

- bidder.Bidder.class
- bidder.BidderHome.class
- bidder.BidderBean.class
- auction.DeploymentDescriptor.txt
- BidderBeanEOImpl_ServiceStub.class

- BidderBeanHomeImpl_ServiceStub.class
- WLStub1z35502726376oa1m4m395m4w5j1j5t.class
- WLStub5g4v1dm3m271tr4i5s4b4k6p376d5x.class
- WLSkel1z35502726376oa1m4m395m4w5j1j5t.class
- WLSkel5g4v1dm3m271tr4i5s4b4k6p376d5x.class

Aquí está cómo unirlos. Todo va en un línea y el comando se ejecuta un directorio por encima de donde se encuentran los ficheros class.

Unix:

```
jar cvf bidder.jar  bidder/*.class
```

Win32:

```
jar cvf bidder.jar  bidder\*.class
```

Una vez que el fichero JAR se ha copiado en el directorio de destino para los Beans Enterprise, lo desempaquetamos de esta forma. La extracción crea un directorio **bidder** con los ficheros class dentro de él.

```
jar xv bidder.jar
```

Paquete seller

Aquí están los ficheros de la aplicación en el paquete **seller** que crea el Bean Enterprise **Seller**.

- seller.Seller.class
- seller.SellerHome.class
- seller.SellerBean.class
- auction.DeploymentDescriptor.txt
- SellerBeanEOImpl_ServiceStub.class
- SellerBeanHomeImpl_ServiceStub.class
- WLStub3xr4e731e6d2x3b3w5b693833v304q.class
- WLStub86w3x4p2x6m4b696q4kjp4p4p3b33.class
- WLSkel3xr4e731e6d2x3b3w5b693833v304q.class
- WLSkel86w3x4p2x6m4b696q4kjp4p4p3b33.class

Aquí está cómo unirlos. Todo va en un línea y el comando se ejecuta un directorio por encima de donde se encuentran los ficheros class.

Unix:

```
jar cvf seller.jar  seller/*.class
```

Win32:

```
jar cvf seller.jar  seller\*.class
```

Una vez que el fichero JAR se ha copiado en el directorio de destino para los Beans Enterprise, lo desempaquetamos de esta forma. La extracción crea un directorio **seller** con los ficheros class dentro de él.

```
jar xv seller.jar
```

Unir y Desplegar el Applet Administration

La familia de ficheros del applet Administration consta de los ficheros AdminApplet.java y polfile.java.

Aquíe está el comando **jar** para unirlos. Todo va en una línea, y el comando se ejecuta dónde está el fichero de policia que es una directorio por encima de donde están los ficheros class.

Unix:

```
jar cvf applet.jar admin/*.class polfile.java
```

Win32:

```
jar cvf applet.jar admin\*.class polfile.java
```

Para desplegar el applet, copiamos el fichero **applet.jar** en el directorio de destino del applet y los extraemos de esta forma. La extracción crea un directorio **admin** con los ficheros del applet Administration dentro de él.

```
jar xf applet.jar
```

Nota: El applet usa los APIs JDK 1.2. Necesita un fichero de policía para acceder a la impresora y Java Plug-In para ejecutarse en un navegador pre-JDK 1.2 . Puedes encontrar información sobre cómo ejecutar el applet con Java Plug-In y un fichero de policía en las siguientes páginas Plataforma Solaris y Plataforma Win32.

Plataforma Solaris

El software Plug-In de Java™ nos permite dirigir applets o componentes JavaBeans™ en páginas de una intranet para que se ejecuten usando el Java Runtime Environment (JRE) en lugar de la máquina virtual por defecto del navegador. El Java Plug-In funciona con Netscape Communicator y Microsoft Internet Explorer.

Descarga todo el software que necesites instalar y usa el Java Plug-In que está disponible desde la página de download.

Get Downloads

Para istalar y usar el Java Plug-In en Solaris™ 2.6 o Solaris 7, necesitamos las siguientes descargar. Ponemos las descargar en cualquier directorio que querramos.

- Java Plug-In para Sistemas Operativos Solaris. Esta disponible para plataformas Intel o Sparc.
- Patches Java Plug-In para Solaris 2.6 o Solaris 7, dependiendo de la que tengamos.
- Netscape Communicator 4.5.1 (versón webstart).
- Java Plug-In HTML Converter

Estas instrucciones se probarón sobre una Sun Microsystems Ultra 2 ejecutando Solaris 2.6 con Netscape Communicator 4.5.1.

Extraer los Ficheros Descargados

Vamos al directorio dónde descargamos los ficheros y extraemos cada uno.

Extraer los ficheros Java Plug-In:

```
zcat plugin-12-webstart-sparc.tar.Z | tar -xf -
```

Extraer los Ficheros del Patch Solaris 2.6:

```
zcat JPI1.2-Patches-Solaris2.6-sparc.tar.Z | tar -xf -
```

Extraer Netscape Navigator 4.5.1:

```
zcat NSCPcom_webstart_sparc.tar.Z | tar -xf -
```

Instalar el Java Plug-In

La descarga del Java Plug-In incluye una guia de usuario que podemos ver en nuestro navegador desde el siguiente directorio:

```
plugin-12-webstart-sparc/Java_Plug-in_1.2.2/
common/Docs/en/Users_Guide_Java_Plug-in.html
```

La guía de usuario explica cómo instalar el Java Plug-In. Hay varias formas sencillas de hacerlo, y la secuencia de comandos de abajo es una forma rápida de instalar Java Plug-In en el directorio por defecto **/opt/NSCPcom** usando el comando **pkgadd**:

```
su
<root password>
cd ~/plugin-12-webstart-sparc
pkgadd -d ./Java_Plug-in_1.2.2/sparc/Product
```

Instalar las Mejoras Java Plug-In

Antes de poder ejecuar el Java Plug-In, tenemos que instalar las mejoras. Las instalamos una a una como raíz. La siguiente secuencia de comandos va al directorio de mejoras, lista los ficheros y envía el comando para instalar la primera mejora:

```
cd ~/JPI1.2-Patches-Solaris2.6-sparC
su
<password>
kq6py#ls
105210-19  105490-07  105568-13
kq6py#./105210-19/installpatch 105210-19
```

Veremos esta salida cuando la mejora se haya instalado satisfactoriamente:

```
Patch number 105210-19 has beenZ successfully
  installed.
See /var/sadm/patch/105210-19/log for details

Patch packages installed:
  SUNWarc
  SUNWcsu
```

Continuamos instalando las mejoraa una por una hasta instalarlas todas. La guía del usuario proporciona una lista de las mejoras necesarias y sugeridas y enlaces a sitios

donde poder descargar las mejoras sugeridas adicionales si queremos instalarlas.

Instalar Netscape Communicator

Los ficheros extraidos de Netscape Communicator 4.5.1 proporcionan una guía de usuario en el directorio **/home/monicap/NETSCAPE/Netscape_Communicator_4.51/common/Docs/en** que explica la instalación. LA siguiente secuencia de comandos es una forma fácil de hacerlo con el comando **pkgadd**. Por defecto, la instación pone Netscape Communicator en el directorio **/opt/NSCPcom** donde también se instalaron Java Plug-In y las mejoras.

Cuando extraemos la descarga **NSCPcom_webstart_sparc.tar.Z**. sitúa los ficheros en un directorio **NETSCAPE**. Desde este directorio ejecutamos la siguientes secuencia de comandos:

```
cd ~/NETSCAPE/Netscape_Communicator_4.51/sparc/Product
su
<password>
pkgadd -d .
```

Chequear la Instalación

Hay dos formas de chequear nuestra instalación del Java Plug-In, las mejoras y Netscape Communicator.

1. Abrir el menú de ayuda de Netscape y selección About Plug_Ins. Veremos una lista de los tipos Mime. Chequeamos esta lista contra la lista presente en la guía de usuario. Si nuestros tipos mime son correctos, la instalación está correcta y completa.
2. Arrancamos el applet del panel de control, cargando el fichero **/opt/NSCPcom/j2pi/ControlPanel.html**. Si el applet arranca la instalación es correcta y completa.

El applet de control nos permite cambiar valores por defecto usado en el arranque del Java Plug-In. Todos los applets ejecutados dentro del Java Plug-In usan esos valores.

```
cd /opt/NSCPcom/j2pi
ControlPanel &
```

Instalar el Conversor HTML

Nuestro navegador no usará automaticamente el Java Plug-In cuando carguemos un fichero HTML con un applet. Tenemos que descargar y ejcutar el Java Plug-In HTML Converter sobre la página HTML que invica al applet para ejecutarlo directamente usando el Plug-In en lugar de hacerlo en el entorno de ejecución por defecto del navegador.

Descomprimimos el fichero de descarga de Plug-In HTML Converter:

```
unzip htmlconv12.zip
```

Añadimos el programa **HTMLConverter.java** o su directorio a nuestro **CLASSPATH**.

Fichero de Policía de Seguridad

La aplicación de subasta usa un applet ejecutándose en un navegador para operaciones administrativas. En la plataforma Java™ 2, los applets están restringidos a un entorno tipo caja sellada y necesitan permisos para acceder a recursos del sistema fuera de ese entorno restrictivo. Los applets están restringidos a operaciones de lectura en su directorio local. Todos las demás operaciones de acceso requieren permisos.

Tipos de Ficheros de Policía

Necesitamos un fichero de policía que conceda permisos al applet Administration. Si el applet se ejecuta enun disco distinto al del navegador, el applet también necesitará estar firmado. Puedes ver la página Applets firmados para más información sobre firmar y desplegar applets.

Hay tres clases de ficheros de policía: sistema, usuario y programa. El fichero de policía del sistema está localizado en **jdk1.2/jre/lib/security/java.policy** o **jre1.2/lib/security/java.policy** y contiene permisos para cada uno en el sistema.

El fichero de policía de usuario está en directorio home del usuario. Este fichero proporciona una forma de dar ciertos permisos de usuario adicioanles a aquellos concedidos a todos en el sistems. Los permisos del fichero del sistema se combinan con los permisos del fichero de usuario.

Un fichero de policía de programa puede situarse en cualquier parte. Se le nombra específicamente cuando se invoca una aplicación con el comando **java** o cuando se invoca un applet con el appletviewer. Cuando una aplicación o un applet se invocan con un fichero de policía específico, los permisos de este fichero ocupan el lugar de (no son combinados con) los permisos específicados en los ficheros del sistema o de usuario. Los ficheros de policía de programa se usan para probar programas o para desplegar en una intraner applets y aplicaciones.

Instalar el Fichero de Policía

Situamos el fichero de policía en nuestro directorio home y lo llamamos **.java.policy**. Cuando el applet intente realizar una acción que requiera un fichero de policía con un permiso, se carga el fichero de policía desde este directorio y permanece en efecto hasta que salgamos del navegador y lo arranquemos de nuevo.

Si un applet intenta realizar una operación sin los permisos correctos, salé discretamente sin lanzar ningún error del applet o del navegador.

Cambiar la Posición o el Nombre

podemos cambiar el nombre y/o la localización de los ficheros de policía del sistema o de usuario por defecto. Editamos los ficheros **jdk1.2/jre/lib/security/java.security**

o **jre1.2/lib/security/java.security** y le añadimos una tercera entrada especificando el nombre y la localización del fichero de policía alternativo.

```
policy.url.1=
  file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
policy.url.3=file:/<mypolicyfile path and name>
```

Ejecutar el Applet Administration

Copiamos el archivo JAR con el applet Administration y el fichero de policía a su localización final. en este ejemplo la localización es el directorio **/home/zelda/public_html**. Luego, extraemos el fichero class del applet y el fichero de policía del fichero JAR:

```
cp admin.jar /home/zelda/public_html
jar xf applet.jar
```

La extracción sitúa el fichero de policía bajo **public_html** y crea un directorio **admin** bajo el directorio **public_html** con el fichero class del applet dentro. Renombramos el fichero de policía del directorio **public_html** como **.java.policy** y lo copiamos en nuestro directorio home.

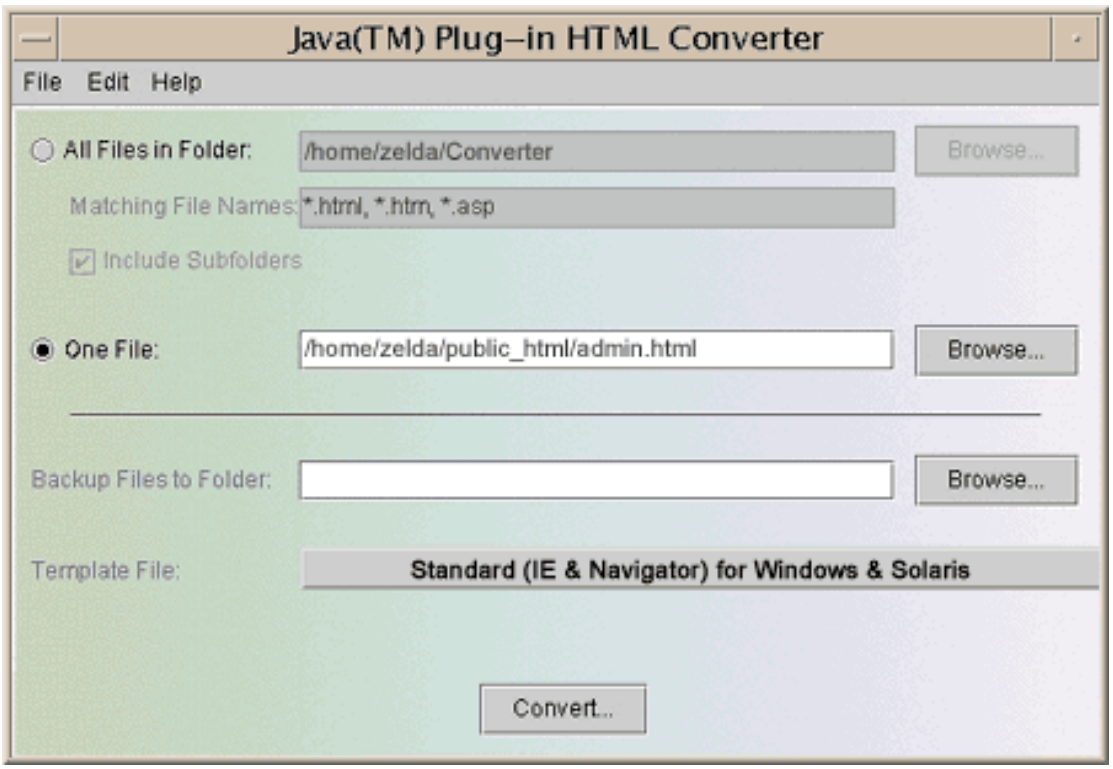
En el directorio **public_html**, creamos un fichero HTML que invoque al applet Administration. Nos debemos asegurar de incluir el directorio **admin** cuando especifiquemos la opción **CODE** del applet. Observamos que cuando usamos Java Plug-In, no podemos hacer que el navegador carge el fichero class desde el fichero JAR.

```
<HTML>
<BODY>
<APPLET CODE=admin/AdminApplet.class
  WIDTH=550
  HEIGHT=150>
</APPLET>
</BODY>
</HTML>
```

Arrancamos el HTML Converter.

```
java HTMLConverter
```

En el interface gráfico de usuario del HTML Converter graphical, seleccionamos **One File:**, especificando el path al fichero **admin.html**, y pulsamos el botón **Convert**.



Después de completar la conversión, cargamos el fichero **admin.html** en nuestro navegador.

Ozito

Plataformas Win32

En plataformas Win32, el software Java™ está unido con el Java 2 Runtime Environment (JRE). Java Plug-In permite a los navegadores usar el JRE de Java 2 para ejecutar applets basados en 1.2 y componentes JavaBeans™ en lugar de la máquina virtual por defecto de los navegadores. El Java Plug-In funciona con Netscape Communicator y Microsoft Internet Explorer.

Obtener las Descargas

Para instalar y utiliar el Java Runtime Environment con Java Plug-In, necesitamos las siguientes descargar. Ponemos las descargas en un directorio temporal.

- Java Runtime Environment com Java Plug-In para Plataformas Win32.
- Java Plug-In HTML Converter

Instalar JRE con Java Plug-In

Una versión opcionalmente instalable de la JRE de Java 2 con Java Plug-In está concluida en la descarga de Java 2 SDK. También podrmos descargar e instalar el Java 2 Runtime Environment com Java Plug-In separadamente.

De cualquier forma, instalamos el Java 2 Runtime Environment con Java Plug-In haciendo doble click sobre su icono y siguiendo las instrucciones de instalación. Cuando la instalación se complete, veremos el panel de control del Java Plug-In en nuestro menú **Start** de Windows bajo **Programs**.

Instalar el HTML Converter

Nuestro navegador no usará automaticamente el Java Plug-In cuando carguemos un fichero HTML con un applet. Tenemos que descargar y ejcutar el Java Plug-In HTML Converter sobre la página HTML que invica al applet para ejecutarlo

directamente usando el Plug-In en lugar de hacerlo en el entorno de ejecución por defecto del navegador.

Descomprimimos el fichero de desacarga del Java Plug-In HTML Converter:

```
unzip htmlconv12.zip
```

Añadimos el programa **HTMLConverter.java** o su directorio a nuestro **CLASSPATH**.

Fichero de Policía de Seguridad

La aplicación de subasta usa un applet ejecutándose en un navegador para operaciones administrativas. En la plataforma Java™ 2, los applets están restringidos a un entorno tipo caja sellada y necesitan permisos para acceder a recursos del sistema fuera de ese entorno restrictivo. Los applets están restringidos a operaciones de lectura en su directorio local. Todos las demás operaciones de acceso requieren permisos.

Tipos de Ficheros de Policía

Necesitamos un fichero de policía que conceda permisos al applet Administration. Si el applet se ejecuta enun disco distinto al del navegador, el applet también necesitará estar firmado. Puedes ver la página Applets firmados para más información sobre firmar y desplegar applets.

Hay tres clases de ficheros de policía: sistema, usuario y programa. El fichero de policía del sistema está localizado en **jdk1.2\jre\lib\security\java.policy** o **jre1.2\lib\security/java.policy** y contiene permisos para cada uno en el sistema.

El fichero de policía de usuario está en el directorio home del usuario. Este fichero proporciona una forma de dar ciertos permisos de usuario adicioanles a aquellos concedidos a todos en el sistems. Los permisos del fichero del sistema se combinan con los permisos del fichero de usuario.

Un fichero de policía de programa puede situarse en cualquier parte. Se le nombra específicamente cuando se invoca una aplicación con el comando **java** o cuando se invoca un applet con el appletviewer. Cuando una aplicación o un applet se invocan con un fichero de policía específico, los permisos de este fichero ocupan el lugar de (no son combinados con) los permisos específicados en los ficheros del sistema o de usuario. Los ficheros de policía de programa se usan para probar programas o para desplegar en una intraner applets y aplicaciones.

Instalar el Fichero de Policía

Situamos el fichero de policía en nuestro directorio home y lo llamamos **java.policy**. Cuando el applet intente realizar una acción que requiera un fichero

de policía con un permiso, se carga el fichero de policía desde este directorio y permanece en efecto hasta que salgamos del navegador y lo arranquemos de nuevo.

Si un applet intenta realizar una operación sin los permisos correctos, salé discretamente sin lanzar ningún error del applet o del navegador.

Cambiar la Posición o el Nombre

Podemos cambiar el nombre o la localización del fichero de policía de usuario o del sistema por dedecto. Editamos los ficheros **jdk1.2\jre\lib\security\java.security** o **jre1.2\lib\security\java.security** y añadimos una tercera entrada especificando el nombre y la localización del fichero de policía alternativo.

```
policy.url.1=file:${java.home}\lib\security\java.policy
policy.url.2=file:${user.home}\java.policy
policy.url.3=file:\<mypolicyfile path and name>
```

Nota: En máquinas Windows/NT, podríamos situar el fichero de policía en el directorio **C:\Winnt\Profiles\<userid>\java.policy**.

Ejecutar el Applet Administration

Copiamos el archivo JAR con el applet Administration y el fichero de policía a su localización final. En este ejemplo, esta localización es el diretorio **\home\zelda\public_html**. Luego extraemos el fichero class del applet y el fichero de policía del fichero JAR:

```
cp admin.jar \home\zelda\public_html
jar xf applet.jar
```

La extracción sitúa el fichero de policía bajo **public_html** y crea un directorio **admin** bajo el directorio **public_html** con el fichero class del applet dentro. Renombramos el fichero de policía del directorio **public_html** como **.java.policy** y lo copiamos en nuestro directorio home.

En el directorio **public_html**, creamos un fichero HTML que invoque al applet Administration. Nos debemos asegurar de incluir el directorio **admin** cuando especifiquemos la opción **CODE** del applet. Observamos que cuando usamos Java Plug-In, no podemos hacer que el navegador carge el fichero class desde el fichero JAR.

```
<HTML>
<BODY>
<APPLET CODE=admin/AdminApplet.class
```

```
    WIDTH=550
    HEIGHT=150>
  </APPLET>
  </BODY>
  </HTML>
```

Arrancamos el HTML Converter.

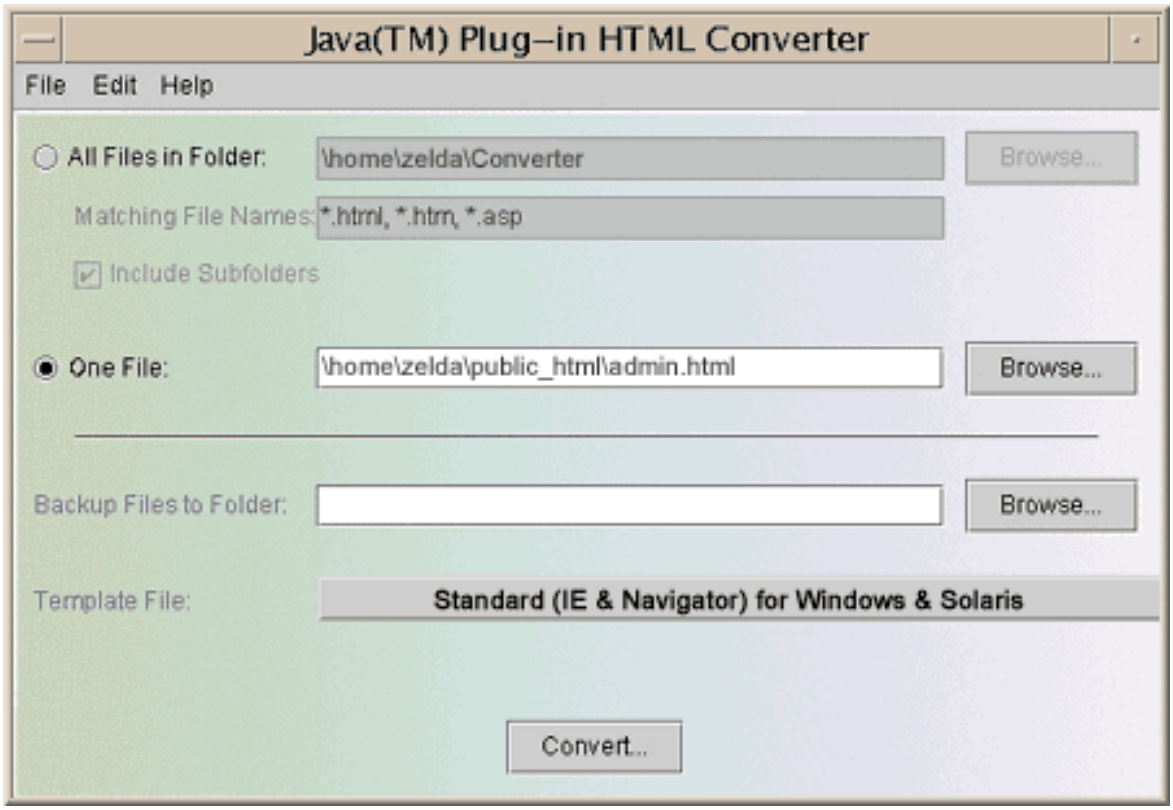
```
java HTMLConverter
```

En el interface gráfico de usuario del HTML Converter graphical, seleccionamos **One File:**, especificando el path al fichero **admin.html**, y pulsamos el botón **Convert**.

¿Cómo Funciona?

En máquinas Windows, el Java Plug-In encuentra el Java Runtime Environment (JRE) ejecutando el fichero de control OLE personalizado **beans.ocx** instalado por defecto en el directorio del navegador web **\Program Files\JavaSoft\1.2\bin**. El control OLE examina el registro de Windows para buscar la clave del Java Plug-In y usa el valor asociado con esa clave para encontrar el JRE instalado.

Si encontramos que se carga un JRE erróneo, usamos **regedit** para chequear el valor del registtro de Java Plug-In para el usuario actual. Si el JRE no está instalado, el control chequea los valores Java Plug-in para la **HKEY_LOCAL_MACHINE**. Deberíamos ver un valor para **Java Runtime Environment** bajo **Software\JavaSoft**.



Después de completar la conversión, cargamos el fichero **admin.html** en nuestro navegaor Web.

Más Tópicos de Seguridad

Este capítulo presenta dos tópicos de seguridades adicionales que podríamos encontrar interesantes.

- Applets Firmados
- Escribir un Control de Seguridad

¿Tienes Prisa?

Esta tabla tiene enlaces directos a los tópicos específicos.

Tópico	Sección
Applets Firmados	• Ejemplo de Applet Firmado • Desarrollador de Intranet • Usuario Final • Ejecutar una Aplicación con un Fichero de Policía • Applets Firmados en JDK 1.1
Escribir un Controlador de Seguridad	• El Programa FileIO • El Programa PasswordSecurityManager • Ejecutar el Programa FileIO • Información de Referencia

---

Applets Firmados

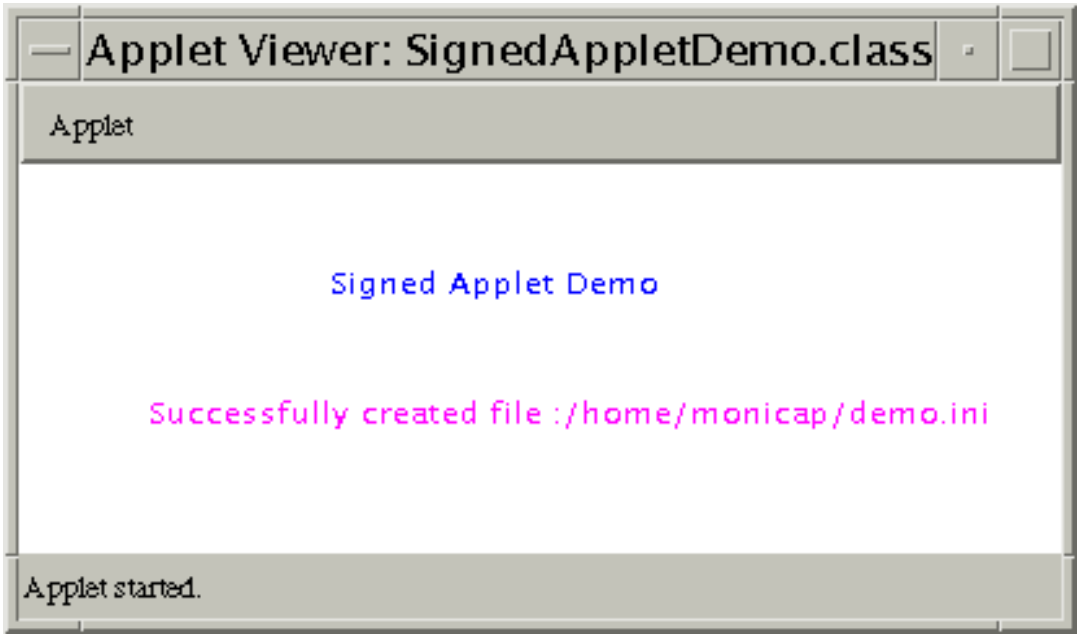
Se peude definir un fichero de policía para requerir una firma de todos los applets o aplicaciones que intenten ejecutarse con el fichero de policía. La firma es una forma de verificar que el applet o la aplicación vienen de una fuente fiable y que puede ser creíada para ejecutarse con los permisos concedidos por el fichero de policía.

Si un fichero de policía requiere una firma, un applet o una aplicación pueden obtener el acceso concedido por el fichero de policía sólo si tienen la firma correcta. Si el applet o la aplicación tienen una firma errónea o no tienen firma, no obtendrán el acceso al fichero.

Esta sección muestra un ejemplo de firma de una applet, verificación de esa firma, y ejecución del applet con un fichero de policía.

Ejemplo del Applet Firmado

El fichero de policía para conceder accesos puede configurarse para que requiera o no una firma. Si se requiere una firma, el applet tiene que estár envuelto en un fichero JAR antes de poder ser firmado. Este ejemplo muestra cómo firmar y conceder los permisos a un applet para que pueda crear un fichero **demo.ini** en el directorio Home del usuario cuando se ejecuta en el AppletViewer.



Estos ficheros son los usados en el ejemplo. Podemos copiarlos o crearlos en nuestro directorio de trabajo.

- El fichero SignedAppletDemo.java que contiene el código del applet.
- Write.jp fichero de policía que concede los accesos al directorio home del usuario.
- Una etiqueta Applet embebida en un fichero SignedApplet.html:

```
<applet code="SignedAppletDemo.class"
        archive="SSignedApplet.jar"
        width=400 height=400>
        <param name=file value="/etc/inet/hosts">
</applet>
```

Normalmente un applet se envulve y se firma mediante un desarrollador de intranet y es manejado por el usuario final que verifica la firma y ejecuta el applet. En este ejemplo, el desarrollador de intranet performa los pasos 1 al 5, y el usuario final realiza lo pasos del 6 al 8. Para mantener las cosas sencillas todos los pasos ocurren en el mismo directorio.

1. Compilar el Applet.
2. Crear el Fichero JAR.
3. Generar las Claves.
4. Firmar el Fichero JAR.
5. Exportar el Certificado de Clave Pública.
6. Importar el Certificado como Certificado Verdadero.
7. Crear el Fichero de Policía.
8. Ejecutar el Applet.

Desarrollador de Intranet

El desarrollador de intranet, envuelve el ejecutable del applet en un fichero JAR, lo firma y exporta el certificado de la clave pública.

1: Compilar el Applet

En su directorio de trabajo el desarrollador de intranet, usa el comando **javac** para compilar la clase **SignedAppletDemo.java**. La salida del comando **javac** es el **SignedAppletDemo.class**.

javac SignedAppletDemo.java

2: Crear el Fichero JAR

El desarrollador de intranet almacena el fichero **SignedAppletDemo.class** compilado en un fichero JAR. La opción **-cvf** del comando **jar** crea un nuevo archivo (c), usando modo verboso (v), y especifica el nombre del fichero archivado (f). El nombre del fichero es **SignedApplet.jar**.

jar cvf SignedApplet.jar SignedAppletDemo.class

3: Generar las Claves

Un fichero JAR se firma con la clave privada del creador del fichero JAR y la firma es verificada por el receptor del fichero JAR con el clave pública de la pareja. El certificado es una sentencia del propietario de la clave privada indicando que la clave pública de la pareja tiene una valor particular para que la persona que la está usando puede estar segura de que es auténtica. Las claves pública y privada deben existir en el almacen de calves antes de que se puede usar jarsigner para firmar o verificar la firma de un fichero JAR.

El desarrollador crea un base de datos **keystore** llamada **compstore** que tiene una entrada para cada pareja de claves recientemente generadas con la clave pública en un certificado usando el comando **keytool**.

En su directorio de trabajo, el desarrollador crea una base de datos keystore y genera las claves:

```
keytool -genkey -alias signFiles -keystore compstore
        -keypass kpi135 -dname "cn=jones"
        -storepass ab987c
```

Este comando **keytool -genkey** invoca una pareja de claves que están identificadas con el Alias signFiles. Subsecuentes llamadas al comando keytool que usarán este alias y la password (**-keypass kpi135**) para acceder a la clave privada en el par generado.

La pareja de claves generadas se almacena en un base de datos keystore llamada compstore (**-keystore compstore**) en el directorio actual y accedida con la password del compstore (**-storepass ab987c**).

La opción **-dname "cn=jones"** especifica un nombre distinguido X.500 con un valor de nombre común (cn). X.500 Distinguished Names identifica entidades para certificados X.509. En este ejemplo, el desarrollador usa su último nombre, Jones, para el nombre común. Podría haber usado cualquier otro nombre para este propósito.

Podemos ver todos las opciones y parámetros de ketool tecleando:

keytool -help

4: Firmar el Fichero JAR

JAR Signer es una herramienta de la línea de comandos para firmar y verificar la firma de ficheros JAR. En su directorio de trabajo, el desarrollado usa jarsigner para firmar una copia del fichero **SignedApplet.jar**.

```
jarsigner -keystore compstore -storepass ab987c
        -keypass kpi135
        -signedjar
        SSignedApplet.jar SignedApplet.jar signFiles
```

Las opciones **-storepass ab987c** y **-keystore compstore** especifican la base de datos keystore y password donde se almacena la clave privada pra firmar el fichero JAR. La opción **-keypass kpi135** es la password de la clave privada, **SSignedApplet.jar** es el nombre del fichero JAR firmado, y **signFiles** es el alias de la clave privada. **jarsigner** extrae el certificado desde la base de datos cuya entrada es **signFiles** y lo adjunta a la firma del fichero JAR firmado.

5: Exportar el Certificado de la Clave Pública

El certificado de la clave pública se envía con el fichero JAR al usuario final que usará el applet. Esta persona usa el certificado para autentificar la firma del fichero JAR. Un certificado se envía exportandolo desde la base de datos **compstore**.

En su directorio de trabajo, el desarrollador usa keytool para copiar el certificado desde **compstore** a un fichero llamado **CompanyCer.cer** de esta forma:

```
keytool -export -keystore compstore -storepass ab987c
        -alias signFiles -file CompanyCer.cer
```

Como el último paso, el desarrollador coloca el fichero JAR y el certificado en un directorio de distribución o en una página web.

Usuario Final

El usuario final descarga el fichero JAR desde el directorio de distribución, importa el certificado, crea un fichero de policía concediendo los accesos al applet, y ejecuta el applet.

6: Importar el Certificado como Certificado Verdadero

El usuario descarga **SSignedApplet.jar** y **CompanyCer.cer** a su directorio home. Ahora debe crear un abase de datos keystore (**raystore**) e importar el certificado en ella usando el aplias **company**. El usuario usa **keytool** en su directorio home para hacer esto:

```
keytool -import -alias company -file
        CompanyCer.cer -keystore
        raystore -storepass abcdefgh
```

7: Crear el Fichero de Policía

El fichero de policía concede al fichero **SSignedApplet.jar** firmado por el alias **company** permiso para crear **demo.ini** (y no otro fichero) en el directorio home del usuario.

El usuario crea el fichero de policía en su directorio home usando **policytool** o un editor ASCII.

```
keystore "/home/ray/raystore";

// A sample policy file that lets a program
// create demo.ini in user's home directory
// Satya N Dodda

grant SignedBy "company" {
  permission java.util.PropertyPermission
    "user.home", "read";
  permission java.io.FilePermission
    "${user.home}/demo.ini", "write";
};
```

8: Ejecutar el Applet en el AppletViewer

AppletViewer conecta con documentos HTML y los recursos especificados en la llamada a **appletviewer**, y muestra el applet en su propia ventana. Para ejecutar el ejemplo, el usuario copia el fichero JAR firmado y el fichero HTML en **/home/aURL/public_html** y llama al Appletviewer desde su directorio raíz de esta forma:

```
appletviewer -J-Djava.security.policy=Write.jp
        http://aURL.com/SignedApplet.html
```

Nota: Se teclea todo en una línea y se pone un espacio en blanco después de **Write.jp**

La opción **-J-Djava.security.policy=Write.jp** le dice al AppletViewer que ejecute el applet referenciado en el fichero **SignedApplet.html** con el fichero de policía **Write.jp**.

Nota: El fichero de policía puede almacenarse en el servidor y especificarse en la invocación al **appletviewer** como una URL.

Ejecutar una Aplicación con un Fichero de Policía

Esta invocación de aplicación restringe **MyProgram** a un entorno cerado de la misma forma en que se restringen los applet, pero permite los accesos especificados en el fichero de policía **polfile**.

```
java -Djava.security.manager
        -Djava.security.policy=polfile MyProgram
```

Applets Firmados en JDK 1.1

Los applets firmados del JDK 1.1 pueden acceser a recursos del sistema local si éste está configurado para permitirlo. Puedes ver la páginas ejemplos de Applets Firmados del JDK 1.1 para más detalles.

Escribir un Controlador de Seguridad

Un controlador de seguridad es un objeto de la Máquina Virtual Java™ (JVM) que implementa un policía de seguridad. Por defecto, la plataforma Java 2® proporciona un controlador de seguridad que desactiva todos los accesos a los recursos del sistema local menos los accesos de lectura al directorio y sus subdirectorios dónde es invocado el programa.

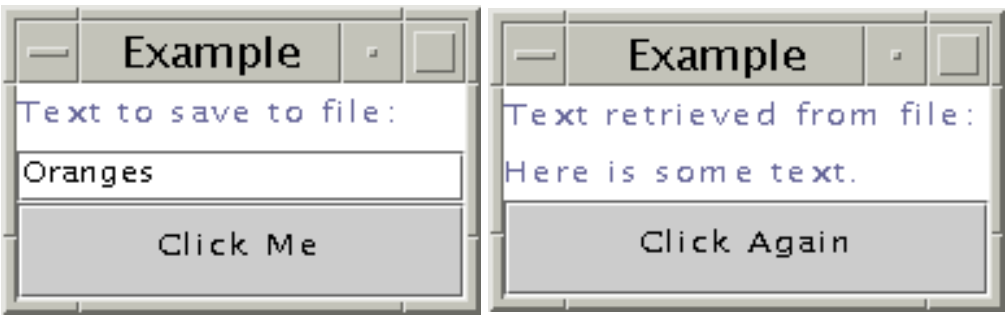
Podemos extender el controlador de seguridad por defecto para implementar verificaciones y aprovaciones personalizadas para applets y aplicaciones, pero la implementación debe incluir código de verificación de accesos apropiado para cada método **checkXXX** que sobreescribamos. Si no incluimos este código, no sucederá ningun chequeo de verificación, y nuestro programa escindirá el fichero de policía del sistema.

Esta sección usa una aplicación de ejemplo para explicar cómo escribir un controlador de seguridad personalizado antes de leer y escribir los ficheros especificados. La implementación incluye código de verificación de accesos por eso una vez que el usuario pasa el chequeo de password, todavía necesita que el fichero tenga permisos de lectura y escritua en su fichero de policía.

El ejemplo consiste en la aplicación **FileIO**, y el programa **PasswordSecurityManager** que proporciona la implementación del controlador de seguridad personalizado.

El programa FileIO

El programa FileIO muestra un sencillo interface de usuario que pide al usuario que introduzca algún texto. Cuando el usario pulsa el botón **Click Me**, el texto se graba en un fichero en el directorio home del usuario, y se abre y se lee un segundo fichero. El texto leído del segundo fichero se muestra al usuario.



Antes de Pulsar el botón Después de Pulsar el botón

El controlador de seguridad personalizado para este programa le pude al usuario final que introduzca una password antes de permitir que **FileIO** escriba o lea texto desde un fichero. El método **main** de **FileIO** crea un controlador de seguridad personalizado llamando **PasswordSecurityManager**.

```
public static void main(String[] args){
  BufferedReader buffy = new BufferedReader(
      new InputStreamReader(System.in));
  try {
    System.setSecurityManager(
      new PasswordSecurityManager("pwd", buffy));
  } catch (SecurityException se) {
    System.err.println("SecurityManager already set!");
  }
```

La Clases PasswordSecurityManager

La clase PasswordSecurityManager declara dos variables de ejemplar privadas, que son inicializadas por el constructor cuando se instala el controlador de seguridad personalziado. La variable de ejemplar **password** contiene el password real, y la variable de ejemplar **buffy** es un buffer de entrada que almacena la password de entrada del usuario final.

```
public class PasswordSecurityManager
               extends SecurityManager{

 private String password;
 private BufferedReader buffy;

 public PasswordSecurityManager(String p,
          BufferedReader b){
   super();
   this.password = p;
   this.buffy = b;
 }
```

El método **accessOK** le pide una password al usuario final, verifica la password, y devuelve **true** si el password es correcto y **false** si no lo es.

```
private boolean accessOK() {
  int c;
  String response;

  System.out.println("Password, please:");
  try {
    response = buffy.readLine();
    if (response.equals(password))
      return true;
    else
      return false;
  } catch (IOException e) {
    return false;
  }
}
```

Verificar Accesos

La clase padre **SecurityManager** proporciona métodos para verificar accesos de lectura y escritura a ficheros del sistema. Los método **checkRead** y **checkWrite** tienen una versión que acepta un **String** y otra versión que acepta un descriptor de ficero.

Este ejemplo sólo sobreescrie las versiones **String** para mantener el ejemplo sencillo, y como el programa **FileIO** usa accesos a directorios y ficheros como **Strings**.

```
public void checkRead(String filename) {
  if((filename.equals(File.separatorChar + "home" +
        File.separatorChar + "monicap" +
        File.separatorChar + "text2.txt"))){
  if(!accessOK()){
    super.checkRead(filename);
    throw new SecurityException("No Way!");
  } else {
    FilePermission perm = new FilePermission(
      File.separatorChar + "home" +
      File.separatorChar + "monicap" +
      File.separatorChar + "text2.txt", "read");
      checkPermission(perm);
      }
   }
}
```

```
public void checkWrite(String filename) {
  if((filename.equals(File.separatorChar + "home" +
                File.separatorChar + "monicap" +
                File.separatorChar + "text.txt"))){
    if(!accessOK()){
      super.checkWrite(filename);
      throw new SecurityException("No Way!");
    } else {
      FilePermission perm = new FilePermission(
                File.separatorChar + "home" +
                File.separatorChar + "monicap" +
                File.separatorChar + "text.txt" ,
                "write");
            checkPermission(perm);
      }
    }
  }
}
```

El mértodo **checkWrite** es llamado antes de escribir la entrada del usuario en el fichero de salida. Esto es porque la clase **FileOutputStream** llama primero a **SecurityManager.checkWrite**.

La implementación personalizadapara **SecurityManager.checkWrite** chequea el pathname **/home/monicap/text.txt**, si es **true** le pide al usuario una password. Si la password es correcta, el método **checkWrite**realiza el chequeo del acceso creando un ejemplar del permiso requerido y pasandolo al método **SecurityManager.checkPermission**. Este chequeo sucederá si el controlador de seguirdad encuentra un fichero de seguridad de sistemam de usuario o de programa con el permiso especificado.

Una vez completada la operación de escritura, al usuario final se le pide la password dos veces más. La primera vez para leer el directorio **/home/monicap**, y la segunda vez para leer el fichero **text2.txt**. Se realiza un chequeo de acceso antes de que tenga lugar la operación de lectura.

Fichero de Policía

Aquñi estña el fichero de policía que necesita el programa **FileIO** para las operaciones de lectura y escritura. También conceder permiso al controlador de seguridad personalizado para acceder a la cola de eventos en representación de la aplicación y mostrar la ventana de la aplicación si ningún aviso.

```
grant {
  permission java.io.FilePermission
        "${user.home}/text.txt", "write";
```

```
    permission java.util.PropertyPermission
          "user.home", "read";
    permission java.io.FilePermission
          "${user.home}/text2.txt", "read";
    permission java.awt.AWTPermission
          "accessEventQueue";
    permission java.awt.AWTPermission
          "showWindowWithoutWarningBanner";
};
```

Ejecutar el programa FileIO

Aquí está cómo ejecutar el programa **FileIO** con el fichero de policía:

```
java -Djava.security.policy=polfile FileIO
```

Información de Referencia

El Apéndice A: Seguridad y Permisos describe los permisos disponibles y explica las consecuencias de conceder permisos. Una forma de usar esta es información es para ayudarnos a limitar los permisos concedidos a un applet o aplciación podrían necesitar ser ejecutados satisfactoriamente. Otra forma de usar esta información es educarnos en la forma en un permiso particular puede ser explotado por código mailicioso.

El Apéndice B: Clases, Métodos y Permisos proporciona lista de métodos de la plataforma Java 2 que están implementados para chequeos de seguridad, los permisos que cada uno requiere, y el método **java.security.SecurityManager** llamado para realizar el chequeo de accesos.

Podemos usar esta referencia para escribir implementaciones de nuestro propio controlador de seguridad o cundo implementamos métodos abstactos que realizan tareas relacionadas con la seguridad.

El Apéndide C: Métodos del SecurityManager lista los chequeos de permisos para los método de **SecurityManager**.i

---

Apéndice A: Seguridad y Permisos

Todos los applets y cualquier aplicacion invocada con un controlador de seguridad debe conceder permisos explícitos para acceder los recursos locales del sistema apaerte del acceso de lectura en el directorio y subdirectorios desde donde se invocó el programa. La plataforma Java™ proporciona permisos para permitir varios niveles de accesos a diferentes tiposde información local.

Como los permisos permiten a un applet o una aplicación sobreescribir el policía de seguridad por defecto, deberíamos ser muy cuidadosos cuando asignemos permisos para no crear una antrada al código malicioso para que ataque nuestro sistema.

Este apéndice describe los permisos disponibles y explica cómo cada permiso puede crear una entrada para un ataque de código malicioso. Una forma de usar esta información es ayudarnos a limitar los permisos dados a un applet o una aplicación a sólo los necesarios para su ejecución. Otra forma de usar esta información es para aprender nosotros mismos las formas en que un permisi particular puede ser explotado por código malicioso.

Como consejo, nunca creas en un applet o una aplicación desconocidos. Siempre chequea el código cuidadosamente con la información de este apéndice para asegurarte de que no ofreces permisos a codigo malicios que cause serios problemas en el sistema local.

Introducción

Los permisos se le conceden a un programa con un ficheo de policía. Un fichero de policía conteine permisos para accesos específicos. Un permiso consta de un nombre de permiso, una fuente, y en algunos casos, una lista de acciones separadas por comas.

Por ejemplo, la siguiente entrada de fichero de policía especifica un permiso **java.io.FilePermission** que concede acceso de **read** (la acción) a la fuente **${user.home}/text2.txt**.

```
grant {
  permission java.io.FilePermission
        "${user.home}/text2.txt", "read";
};
```

Hay un fichero de policía de la instalación de la plataforma Java (Sistema) y opcionalmente un fichero de policía por cada usuario. El fichero de policía del sistema está en **{java.home}/lib/security/java.policy**, y el fichero de policía de usuario está en cada directorio home de los usuarios. Los ficheros de policía de sistema y de usuario se combinan. Por eso por ejemplo, podría haber un fichero de policía con muy pocos permisos concedidos a todos los usuarios del sistema, y un fichero de policía individual concediendo permisos adicionales a ciertos usuarios.

Para ejecutar una aplicación con el controlador de seguridad y un fichero de policía llamado **polfile** en el directorio home del usuario, tecleamos:

```
java -Djava.security.main
-DJava.security.policy=polfile  FileIO
```

Para ejecutar un applet en el aplletviewer con un fichero de policía llamando **polfile** en el directorio home del usuario, tecleamos:

```
appletviewer
-J-Djava.security.policy=polfile fileIO.html
```

Cuando ejecutamos un applet en un navegador, este busca los ficheros de policía de usuario y del sistema para encontrar los permisos que necesita el applet para acceder a los recursos del sistema local en representación del usuario que descargó el applet.

Conocer qué Permisos

Cuando ejecutamos un applet o invocamos una aplicación con un controlador de seguridad que necesita permisos, obtendremos un seguimiento de pila si no hemos proporcionado un fichero de policía con todos los permisos necesarios. El seguimiento de pila contiene la información que necesitamos para añadir los permisos al fichero de policía que causó el seguimiento de pila. Si el programa

necesita permisos adicionales, seguiremos obteniendo el seguimiento d epila hasta que se añadan todos los permisos requeridos al fichero de policía. El único inconveniente de esta aproximación es que tenemos que probar cada posbiel path de código de nuestra aplicación.

Otra forma de determinar qué permisos necesita nuestro programa es visitar Apéndice B: Métodos y Permisos. Este apéndice nos cuenta qué métodos de la plataforma Java 2 tienen impedida la ejecución sin los permisos listados. La información del Apéndice B también es útil para los desarrolladores que quieran escribir su propio controlador de seguridad para personalizar las verificaciones y aprovaciones necesarias en un programa.

Aquí tenemos un pequeño ejemplo que nos muestra como traducir el primer par de líneas del seguimiento de pila en una entrada del fichero de policí. La primera línea nos dice que el acceso está denegado. Esto significa que el seguimiento de pila fue generado porque el programa intentó acceder a recursos del sistema sin el permiso apropiado. La segunda línea significa que necesitamos un **java.net.SocketPermission** que le de programa permiso para **connect** y para **resolve**el nombre de host para la dirección (IP) 129.144.176.176, puerto 1521.

```
java.security.AccessControlException: access denied
     (java.net.SocketPermission
     129.144.176.176:1521 connect,resolve)
```

Para volver esto en una entrada del fichero de policía, listamos el nombre del permiso, una fuente, u una lista de acciones donde **java.net.SocketPermission** es el nombre del permiso, **129.144.176.176:1521** es la fuente, y **connect,resolve** es la lista de acciones:

```
grant {
  permission java.net.SocketPermission
  "129.144.176.176:1521", "connect,resolve";
};
```

AllPermission

java.security.AllPermission especifica todos los permisos en el sistema para todos los posibles fuentes y acciones. Este permiso sólo debería usarse para pruebas ya que concede permiso para ejecutar con todas las restricciones de seguridad desactivadas, como si no hubiera controlador de seguridad.

```
grant {
  permission java.security.AllPermission;
};
```

AWTPermission

java.awt.AWTPermission concede permisos a las siguientes fuentes AWT. Las posibles fuentes se listan por nombre sin lista de acciones.

```
grant {
  permission java.awt.AWTPermission
              "accessClipboard";
  permission java.awt.AWTPermission
              "accessEventQueue";
  permission java.awt.AWTPermission
              "showWindowWithoutWarningBanner";
};
```

accessClipboard: Esta fuente concede permiso para poner información y para recuperarla desde el portapapeles del AWT. Conceder este permiso puede permitir a código malicioso que comparta informacion potencialmente sensible o confidencial.

accessEventQueue: Esta fuente concede permiso para acceder a la cola de eventos del AWT. Conceder este permiso podría permitir a código mailicioso que observe y elimine eventos del sistema, y ponga eventos falsos que podrían causar que la aplicación o el appler realizarán acciones maliciosas.

listenToAllAWTEvents: Esta fuente concede permiso para escuvhar todos los eventos AWT a través del sistema. Conceder este permiso podría permitir a código malicios lea y explote entradas confidenciales del usuario como las passwords.

Cada oyente de evento AWT es llamado desde dentro del contexto **EventDispatchThread** de la cola de eventos, por eso si también está activado el permiso **accessEventQueue**, código malicioso podría modificar el contenido de la cola de eventos del AWT a través del sistema, lo que podría causar que el applet o la aplicación realizarán acciones maliciosas.

readDisplayPixels: Esta fuente concede permiso para leer pixels desde la pantalla. Condeder este permiso podría permitir a interfaces como **java.awt.Composite** permitan examinar los pixels de la pantalla y fisgonee las actividades del usuario.

showWindowWithoutWarningBanner: Esta fuene concede permiso para mostrar una ventana sin mostrar el aviso de que la ventana fue creada por un applet. Sin este aviso, un applet podría mostrar ventanas si que el usuario supiera que pertenecen al applet. Esto podría ser un problema en entornos dónde el usuario toma decisiones sensibles de seguridad básandose en a quién pertenece la ventana si a un applet o a una aplicación. Por ejemplo, desactivar este aviso podría significar que el usuario introdujeta información sensile como el nombre de usuario y la password.

FilePermission

java.io.FilePermission concede permiso para acceder a un fichero o directorior. La fuentes consisten en el pathname y una lista de acciones separadas por comas.

Este fichero de policía concede permisos de lectura, escritura, borrado y ejecución para todos los ficheros.

```
grant {
  permission java.io.FilePermission
        "<<ALL FILES>>", "read, write, delete, execute";
};
```

Este fichero de policía concede permiso de lectura y escritura sobre **text.txt** en el directorio home del usuario.

```
grant {
  permission java.io.FilePermission
        "${user.home}/text.txt", "read, write";
};
```

Podemos usar los siguiente comodines para especificar al pathname de la fuente:

- Un pathname que termine en **/***, donde **/*** es el carácter separador de ficheros significa un directorio y todos los ficheros contenidos en ese directorio.
- Un pathname que termine con **/-** indica un directorio, y recursivamente, todos los ficheros y subdirectorios incluidos en ese directorio
- Un pathname que consista en un sólo asterisco (*****) indica todos los ficheros del directorio actual.
- Un pathname que consista en un sólo guión (**-**) indica todos los ficheros del directorio actual, y recursivamente, todos los ficheros y subdirectorios contenidos en el directorio actual.

Las acciones son especificadas en una lista de palabras clave separadas por comas que tienen el siguiente significado:

- read: Permiso para leer un fichero o directorio.
- write: Permiso para escribir o crear un fichero o directorio.
- execute: Permiso para ejecutar o ficheo o buscar un directorio.
- delete: Permiso para borrar un fichero o directorio.

Cuando concedamos permisos de ficheros, siempre debemos pensar en las implicaciones de conceder permisos de lectura y especialmente de escritura a varios ficheros y directorios. El permiso **<<ALL FILES>>** con acción de escritura es especialmente peligroso porque concede permiso para escribir en todo el sistema de ficheros. Esto significa que el sistema binario puede ser reemplazado, lo que incluye el entorno de ejecución de la máquina virtual Java.

NetPermission

java.net.NetPermission concede permisos a varias fuentes de red. Las posible fuentes se listan por el nombre sin lista de acciones.

```
grant {
  permission java.net.NetPermission
        "setDefaultAuthenticator";
  permission java.net.NetPermission
        "requestPasswordAuthentication";
};
```

setDefaultAuthenticator: Esta fuente concede permiso para seleccionar la forma en que información de autentificación es recuperad cuando un proxy o un servidor HTTP piden autentificación. Concedere ste permiso podría significar que código mailicioso puede seleccinar un autentificador que monitorice y robe la entrada de autentificación del usuario como si recuperar la información desde el usuario.

requestPasswordAuthentication: Esta fuente concede permiso para pedir al autentificador registrado con el sistema una password. Conceder este permiso podría significar que código preciado podría robar la password.

specifyStreamHandler: Esta fuente concede permiso para especificar un manejador de stram cuando se construye una URL. Conceder este permiso podría significar que código malicioso podría crear una URLK con recursos a los que normalmente no tendría acceso, o especificar un controlador de stream para obtener los bytes reales desde algun real al que tenga acceso. Esto significa que el código malicios podría embaucar al sistema para crear una clase ProtectionDomain/CodeSource incluso aunque la clase realmente no venga de esa localización.

PropertyPermission

java.util.PropertyPermission concede acceso a las propiedades del sistema. La clase **java.util.Properties** represetna selecciones persistentes como la localización del directorio de instalación, el nombre de usuario o el directorio home del usuario.

```
grant {
  permission java.util.PropertyPermission
        "java.home", "read";
  permission java.util.PropertyPermission
        "os.name", "write";
  permission java.util.PropertyPermission
        "user.name", "read, write";
};
```

La lista de fuentes contiene el nombre de la propiedad, por ejemplo, **java.home** o **os.name**. La convención de nombres para propiedades sigue la convención de nombres hereditarios, e incluye comodines. Un asterisco al final del nombre de propiedad, después de un punto (**.**), o en solitario, significa un comodin. Por ejemplo, **java.*** o ***** son válidos, pero ***java** o **a*b** no lo son.

Las acciones se especifican en una listra de palabras claves separadas por comas, que tienen este significado:

- read: Permiso para leer (obtener) una propiedad.
- write: Permiso para escribir (seleccionar) una propiedad.

Conceder permisos a propiedades puede dejar nuestro sistema abierto a la intrusión. Por ejemplo, conceder permiso para acceder a la propiedad **java.home** hace vulnerable a ataques el directorio de la instalación, y conceder permiso de acceso a las propiedades **user.name** y **user.home** podría revelar el nombre de cuenta del usuario y el directorio home.

ReflectPermission

java.lang.reflect.ReflectPermission concede permiso para varias operaciones reflectivas. Las posibles fuentes se listan por el nombre sin lista de acciones.

```
grant {
  permission java.lang.reflect.ReflectPermission
        "suppressAccessChecks";
};
```

suppressAccessChecks: Esta fuente concede permiso para acceder a los campos e invocar métodos de una clase. Esto incluye campos y métodos públicos, protegidos y privados. Conceder este permiso pidría revelar información confidencial y poner métodos importantes al alcance del código malicioso.

RuntimePermission

java.lang.RuntimePermission concede permiso a varias fuentes del entorno de ejecución, como el cargador de clases, la máquina virtual Java y los threads. Las posibles fuentes se listan por el nombre sin lista de acciones.

```
grant {
  permission java.lang.RuntimePermission
        "createClassLoader";
  permission java.lang.RuntimePermission
        "getClassLoader";
  permission java.lang.RuntimePermission
        "exitVM";
  permission java.lang.RuntimePermission
```

```
            "setFactory";
        permission java.lang.RuntimePermission
            "setIO";
        permission java.lang.RuntimePermission
            "modifyThread";
        permission java.lang.RuntimePermission
            "modifyThreadGroup";
        permission java.lang.RuntimePermission
            "getProtectionDomain";
        permission java.lang.RuntimePermission
            "setProtectionDomain";
        permission java.lang.RuntimePermission
            "readFileDescriptor";
        permission java.lang.RuntimePermission
            "writeFileDescriptor";
        permission java.lang.RuntimePermission
            "loadLibrary.<library name>";
        permission java.lang.RuntimePermission
            "accessClassInPackage.<package name>";
        permission java.lang.RuntimePermission
            "defineClassInPackage.<package name>";
        permission java.lang.RuntimePermission
            "accessDeclaredMembers.<class name>";
        permission java.lang.RuntimePermission
            "queuePrintJob";
    };
```

Las convenciones de nombrado para la fuente sigue la convención de nombres hereditarios, e incluye comodines. Un asterisco al final del nombre de propiedad, después de un punto (**.**), o en solitario, significa un comodin. Por ejemplo, **loadLibrary.*** o ***** son válidos, pero ***loadLibrary** o **a*b** no lo son.

createClassLoader: Esta fuente concede permiso para crear un cargador de clases. Conceder este permiso podría permitir a una aplicación maliciosa que ejemplarize su propio cargador de clases y carge clases peligrosas en el sistema. Una vez cargado, el cargador de clases podría situar esas clases bajo cualquier dominio proegido dándoles total control sobre ese dominio.

getClassLoader: Esta fuene concede permiso para recuperar el cargador de clases para la clase llamante. Conceder este permiso podría permitir que código malicioso obtuviere el cargador de clases para una clase particular y cargar clases adicionales.

setContextClassLoader: Esta fuente concede permiso para seleccionar el contexto del cargador de clases usado por un thread. El código del sistema y las extensione susan este contexto para buscar recursos que podrían no existir en el cargador de clases del sistema. Conceder este permiso permite cambiar el contexto

del cargador de clases usado para un thread particular, incluyendo los threads del sistema. Esto podría causar problemas si el contexto del cargador de clases tiene código malicioso.

setSecurityManager: Esta fuente concede permiso para seleccionar o reemplazar el controlador de seguridad. El controlador de seguridad es una clase que permite a la aplicaciones implementar un policía de seguridad. Conceder este permiso podría permitir al código mailicioso instalar un controlador menos restrictivo, y por lo tanto, evitar los chequeos a los que se forzado el controlador de seguridad original.

createSecurityManager: Esta fuente concede permiso para crear un nuevo controlador de seguridad. Conceder este permiso podría darle al código malicioso acceso a método protegidos qie podrían revelar información sobre otras clases o la pila de ejecución.

exitVM: Esta fuente concede permiso para parar la máquina virtual Java. Conceder este permiso podría permitir que código malicioso a montar un ataque de denegación de servicio forzando automáticamente a que se pare la JVM.

setFactory: Esta fuente concede permiso para seleccioanr una fábrica de socket usada por las clases **ServerSocket** o **Socket**, o la fábrica de manejadores de streams usada por la clase **URL**. Conceder este permiso permite al código seleccionar la implementación actual para la factoría de socket, server socket, stream handler, o Remote Method Invocation (RMI). Un atacante podría seleccionar una implementación que manejara los streams de datos.

setIO: Esta fuente concede permiso para cambiar los valores de los streams **System.out**, **System.in**, **System.err**. Conceder este permiso podría pemritir a un atancante cambiar el **System.in** para robar la entrada del usuario, o seleccionar **System.err** a un stream de salida **null**, lo que podría ocultar cualquier error enviado a **System.err**.

modifyThread: Esta fuente concede permiso para modicar los threads mediante llamadas a los métodos **stop**, **suspend**, **resume**, **setPriority**, y **setName** de la clase **Thread**. Conceder este permiso podría permitir a un atancante arrancar o suspender cualquier tread e el sistema.

stopThread: Esta fuente concede permiso para parar threads. Conceder este permisos permtie al código que pare cualquier thread en el sistema proporcionando el código que ya tiene permiso para acceder a ese thread, EL código malicioso podría corromper el sistema eliminado threads existentes.

modifyThreadGroup: Esta fuente concede permiso para modificar threads mediante llamadas a los métodos **destroy**, **resume**, **setDaemon**, **setmaxPriority**, **stop**, y **suspend** de la clase **ThreadGroup**. Conceder este permiso podría permitir al atancante que cree un grupo de threadas y seleccionar su prioridad de ejecución.

getProtectionDomain Esta fuente concede permiso para recuperar el ejemplar

ProtectionDomain para una clase. Conceder este permiso pemite al código obtener información de policía para ese código fuente. Mientras que la obtención de la información de policía no compromete la seguridad del sistema, si que le ofrece información adicional al atacante como los nombres de ficheros locales, por ejemplo.

readFileDescriptor: Esta fuente concede permiso para leer descriptores de ficheros. Conceder este permiso permite al código leer el fichero particular asociado con el descriptor de fichero, que es peligroso si el fichero contiene datos confidenciales.

writeFileDescriptor: Esta fuente concede permiso para escribir descriptores de ficheros. Conceder este permiso permite al código escribir el fichero asociado con el descriptor de fichero, lo que es peligroso si el descriptor apunta a un fichero local.

loadLibrary.{library name}: Este fichero concede permiso para enlazar dinámicamente la librería especificada. Conceder este permiso podría ser peligroso porque la arquitectura de seguridad no está diseñada y no se extiende para las clases nativas cargadas mediante el método **java.lang.System.loadLibrary**.

accessClassInPackage.{package name} Esta fuente concede permiso para acceder al paquete especificado mediante el método **loadClass** del cargador de la clase cuando el cargador de la clase llama al método **SecurityManager.checkPackageAcesss**. Conceder este permiso le da al código acceso a las clases de paquetes a los que normalmente no tiene acceso. El código mailicioso podría usar estas clases para ayudarse en su intento de comprometer la seguridad del sistema.

defineClassInPackage.{package name}: Esta fuente concede permiso para definir las clases del paquete especificado mediante un método **defineClass** del cargador de clases cuando el cargador llama al método **SecurityManager.checkPackageDefinition**. Conceder este permiso permite al código definir una clase en un paquete particular, lo que podría ser peligroso porque el código malicioso con este permiso podría definir clases peligrosas en paquetes verdaderos como **java.security** o **java.lang**, por ejemplo.

accessDeclaredMembers: Esta fuente concede permiso para acceder a miembros declarados de una clase. Conceder este permiso permite al código solicitar una clase por sus campos y métodos públicos, protegidos, por defecto (paquete) y privados. Aunque el código podría tener acceso a los nombres de los campos y métodos privados y protegidos, no podrá acceder a sus datos y no podrá invocar ningún método privado. A pesar de esto, el código malicioso podría usar esta información para mejorar su ataque. Además, el código malicios podría invocar métodos públicos o acceder a campos públicos de la clase, lo que podría ser peligroso.

queuePrintJob: Esta fuente concede permiso para inicializar una petición de

trabajo de impresión. Conceder este permiso podría permitir al código qe imprima información sensible en una impresora o que gaste papel maliciosamente.

SecurityPermission

java.security.SecurityPermission conceder permiso ha varios parámetros de configuración de seguridad. Las fuentes posibles se listan por el nombre sin lista de acciones. Los permisos de seguridad actualmente se aplican a método llamados sobre los siguientes objetos:

- **java.security.Policy**, que representa la policía de seguridad del sistema para aplicaciones.
- **java.security.Security**, que centraliza todas las propiedades de seguridad y métodos comunes. Maneja proveedores.
- **java.security.Provider**, que repesenta una implementación de cosas como algoritmos de seguridad (DSA, RSA, MD5, or SHA-1) y generación de claves.
- **java.security.Signer**, que maneja claves privadas. Aunque, **Signer** está obsoleto, los permisos relacionados están disponibles por razones de compatibilidad.
- **java.security.Identity**, que maneja objetos del mundo real como son personas, compañias, y organizaciones, cuyas identidades pueden ser autentificadas usando sus claves públicas.

```
grant {
  permission java.security.SecurityPermission
        "getPolicy";
  permission java.security.SecurityPermission
        "setPolicy";
  permission java.security.SecurityPermission
        "getProperty.os.name";
  permission java.security.SecurityPermission
        "setProperty.os.name";
  permission java.security.SecurityPermission
        "insertProvider.SUN";
  permission java.security.SecurityPermission
        "removeProvider.SUN";
  permission java.security.SecurityPermission
        "setSystemScope";
  permission java.security.SecurityPermission
        "setIdentityPublicKey";
  permission java.security.SecurityPermission
        "setIdentityInfo";
  permission java.security.SecurityPermission
        "addIdentityCertificate";
  permission java.security.SecurityPermission
```

```
            "removeIdentityCertificate";
      permission java.security.SecurityPermission
            "clearProviderProperties.SUN";
      permission java.security.SecurityPermission
            "putProviderProperty.<provider name>";
      permission java.security.SecurityPermission
            "removeProviderProperty.SUN";
      permission java.security.SecurityPermission
            "getSignerPrivateKey";
      permission java.security.SecurityPermission
            "setSignerKeyPair";
   };
```

getPolicy: Esta fuente concede permiso para recuperar el policía de seguridad del sistema. Conceder estep permiso revela qué permisis seberían concederse a una aplicación o un applet dados. Mientras que la revelación del policía no compromete la seguridad del sistema, proporciona al código malicios información adicional que podría usar para un mejor ataque.

setPolicy: Esta fuente concede permiso para seleccionar el policía de seguridad del sistema. Conceder este permiso podría permitir al código malicioso que se conceda a sí mismo todos los permiso para montar un ataque al sistema.

getProperty.{key}: Esta fuente concede permiso para recuperar la propiedad de seguridad especificada mediante **{key}**. Dependiendo de la clave particular para el que se concede el acceso, el código podría tener accedo a una lista de proveedores de seguridad y la localización de las policías de seguridad del sistema y de usuario. Mientras que la revelación de esta información no compromete la seguridad del sistema, si proporciona información adicional que podría usar para un mejor ataque.

setProperty.{key}: Esta fuente concede permiso para seleccionar la propiedad de seguridad especificada por **{key}**. Esto podría incluir la selección de un proveedor de seguridad o definir la localización del policía de seguridad del sistema. El código malicioso podría usar un proveedor maligno que robará información confidencial como las claves privadas. Además, el código malicioso podría seleccionar con los permisos seleccionar la localización del policía de seguridad del sistema que podría apuntar a un policía de seguridad que conceda al atancante todos los permisos necesarios que requiera para montar el ataque al sistema.

insertProvider.{provider name}: Esta fuente concede permiso para añadir un nuevo proveedor de seguridad especificado por **{provider name}**. Conceder este permiso permite la introducción un proveedor posiblemente malicioso que podría desubrir cosas como las claves provadas que se les pasa. Esto es posible porque el objeto **Security**, que maneja todos los proveedores instaladore, no cheque realmente la integridad o la autenticidad de un proveedor antes de adjuntarlo.

removeProvider.{provider name}: Esta fuente concede permiso para eliminar un proveedor de seguridad especificado por **{provider name}**. Conceder este permiso podría cambiar el comportamietno o desactivar la ejecuciónde partes del programa. Si un proveedor solicitado por el programa ha sido eliminado, la ejecución podría fallar.

setSystemScope: Esta fuente concede permiso para seleccionar el ámbito de identidad del sistema. Conceder este permiso podría permitir al atacante configurar el ámbito de seguridad del sistema con certificados que no deberían ser creidos. Esto podría conceder al código firmado cuyos privilegios certificados podrían ser denegados por el ámbito de identidad original.

setIdentityPublicKey: Esta fuente concede permiso para seleccionar la clave pública de un objeto **Identity**. Si la identidad se marca como *trusted*, permite al atacante introducir su propia clave pública que no es verdadera mediante el ámbito de identidad del sistema. Esto podría conceder al código firmado privilegios de clave pública que de otra forma serían denegados.

SetIdentityInfo: Esta fuente concede permiso para seleccionar un string de información general para un objeto **Identity**. Conceder este permiso permite al atancate seleccionar una descripción general para una identidad. Haciéndolo podríamos embaucar a las aplicaciones a usar una identidad diferente que evite a las aplicacioens encontrar una identidas particular.

addIdentityCertificate: Esta fuente concede permiso para añadir un certificado para un objeto **Identity**. Conceder este permiso permite a los atacantes seleccionar un certificado para una clave pública de identidad haciendo que la clave pública sea verdadera a una audiencia mayor de la original.

removeIdentityCertificate: Esta fuente concede permiso para eliminar un certificado de un objeto **Identity**. Conceder este permiso permite al atacante eliminar un certificado para la clave pública de una identidad. Esto podría ser peligroso porque una clave pública podría ser considerada como menos verdadera de lo que podría ser.

printIdentity: Esta fuente concede permiso para imprimir el nombre de un prinicpal el ámbito en que se usa el principal, y cuando el principal es considerado como **verdadero** en este ámbito. El ámbito impreso podría ser un nombre de fichero, en cuyo caso podría robar información del sistema local. Por ejemplo, aquí hay un ejemplo de impresión de un nombre de identidad *carol*, que está marcado como verdadero en la base de datos de identidad del usario:

```
carol[/home/luehe/identitydb.obj][not trusted].
```

clearProviderProperties.{provider name} Esta fuente concede permiso para borrar un objeto **Provider** para que no contenga más propiedades usadas para buscar servicios implementados por el proveedor. Conceder este permiso desactiva los servicios de búsqueda implementados por el proveedor. Esto podría cambiar el

comportamiento o desactivar la ejecuciónde otras partes del programa que normalmente utiliará el Provider, como se describe bajo el permiso **removeProvider.{provider name}** de arriba.

putProviderProperty.{provider name}: Esta fuente concede permiso para seleccionar propiedades del proveedor seleccionado. Cada propiedad del proveedor especifica el nombre y la localización de un servicio particular implementado por el proveedor. Conceder este permiso permite al código reemplazar la especificación de servicio con otro con una diferente implementación y podría ser peligroso si la nueva implementación tiene código malicioso.

removeProviderProperty.{provider name}: Esta fuente concede permiso para eliminar propiedades del proveedor especificado. Conceder este permiso desactiva la búsqueda de servicios implementada por el proveedor haciéndola inaccesible. Conceder este permiso a código malicioso podría permitirle cambiar el comportamiento o desactivar la ejecución de otras partes del programa que normalmente podrían utilizar el objeto **Provider**, como se describe el permiso bajo **removeProvider.{provider name}**.

getSignerPrivateKey: Esta fuente concede permiso para recuperar la clave privada de un objeto **Signer**. Las calves privadas deberían ser siempre secretas. Conceder este permiso podría permtir a código malicioso utilizar la clave privada para firmar ficheros y reclamar que la firma venga del objeto **Signer**.

setSignerKeyPair: Esta fuente concede permiso para seleccionar la pareja de claves pública y privada para un objeto **Signer**. Conceder este pemriso podría permitir al atacante reemplazar la pareja de claves con una posible y pequeña pareja de claves. Esto también podría permitir a un atacante escuchar una comunicación encriptada entre la fuente y sus pares. Los pares de la fuente podrían envolver la sesión de encriptación bajo la clave pública *new*, que podría el atacante (que posee la clave privada correspondiente) para desempaquetar la clave de sesión y desencriptar la comunicación.

SerializablePermission

java.io.SerializablePermission concede acceso a operaciones de serialización. La fuentes posibles se listan por el nombre y no hay lista de acciones.

```
grant {
  permission java.io.SerializablePermission
        "enableSubclassImplementation";
  permission java.io.SerializablePermission
        "enableSubstitution";
};
```

enableSubclassImplementation: Esta fuente concede permiso para implementar una subclase de **ObjectOutputStream** o **ObjectInputStream** para

sobreescribir la serialización o deserialización por defecto de objetos. Conceder este permiso podría permitir al código usar esto para serializar o deserializar clases de una forma maliciosa. Por ejemplo, durante la serialización, el código malicioso podría almacenar campos privados confidenciales de una forma fácilmente accesible para los atacantes; o durante la deserialización el código malicioso podría deserializar una clase con todos sus campos privados puestos a cero.

enableSubstitution: Esta fuente concede permiso para sustituir un objeto por otro durante la serialización deserialización. Conceder este permiso podría permitir a código malicioso reemplazar el objeto real con otro que tenga datos incorrectos o malignos.

SocketPermission

El permiso **java.net.SocketPermission** concede acceso a una red mediante sockets. La fuente es un nombre de host y la dirección del puerto, y la acciónes una lista que especifica las formas de conexión con ese host. Las coneciones posibles son **accept**, **connect**, **listen**, y **resolve**.

Esta entrada de fichero de policía permite que una conexión acepte conexiones al puerto 7777 en el host **puffin.eng.sun.com**.

```
grant {
  permission java.net.SocketPermission
        "puffin.eng.sun.com:7777",
        "connect, accept";
};
```

Esta entrada de fichero de policia permite a la conexión, aceptar conexiones para escuchar cualquier puerto entre el 1024 y el 65535 en el host local.

```
grant {
  permission java.net.SocketPermission
        "localhost:1024-",
        "accept, connect, listen";
};
```

El host se expresa con la siguiente sintaxis como un nombre DNS, una dirección IP numérica, o como **localhost** (para la máquina local). El comodin asterisco (*****) se puede incluir una vez en una especificación de nombre DNS. Si se incluye dene estár en la posición más a la izquierda, como en ***.sun.com**.

```
  host = (hostname | IPaddress)[:portrange]
  portrange = portnumber | -portnumber |
        portnumber-[portnumber]
```

El puerto o rango de puertos es opcional. Una especificación de puerto de la forma **N-**, donde **N** es un número de puerto, significa todos los puertos numerados **N** y

superiores, mientras que una especificación de la forma **-N** indica todos los puertos numerados **N** e inferiores.

La acción **listen** es sólo importante cuando se usa con el **localhost**, y **resolve** (resuelve la dirección del servicio host/ip) cuando cualquiera de las otras opcioones está presente.

Conceder permiso al código para aceptar o crear conexiones sobre host remotos podría ser peligroso porque código malevolente podría más fácilmente transferir y compartir datos confidenciales.

Nota: En plataformas Unix, sólo el raíz tiene permiso para acceder a los puertos inferiores a 1024.

Apéndice B: Clases, Métodos y Permisos

Un gran número de métodos de la plataforma Java™ 2 estan implementados para verificar permisos de acceso. Esto significa que antes de ejecutarse, verifican si ay un fichero de policia de sistema, usuario o programa con los permisos requeridos para que continue la ejecución. Si no se encuentran dichos permisos, la ejecución se detiene con una condición de error.

El código de verificación de acceso pasa los permisos requeridos al controlador de seguridad, y el controlador de seguridad comprueba estos permisos contra los permisos del fichero de policía para determinar los accesos. Esto significa que los métodos de la plataforma Java 2 están asociados con permisos específicos, y los permisos específcios están asociados con métodos específicos del **java.security.SecurityManager**.

Este apéndide lista los métodos de la plataforma Java 2, los permisos asociados con cada método, y el método **java.security.SecurityManager** llamado para verificar la existencia de este permiso. Necesitamos esta información cuando implementamos ciertos métodos abstractos o creamos nuestro propio controlador de seguridad para que podamos incluir código de verificación de acceso para mantener nuestras implementaciones en línea con la política de seguridad de la plataforma Java 2. Si no incluimos código de verificación de acceso, nuestras implementaciones no pasarán los chequeos de seguridad internos de la plataforma Java 2.

- java.awt.Graphics2D
- java.awt.ToolKit
- java.awt.Window
- java.beans.Beans
- java.beans.Introspector
- java.beans.PropertyEditorManager
- java.io.File
- java.io.FileOutputStream
- java.io.ObjectInputStream
- java.io.ObjectOutputStream
- java.io.RandomAccessFile
- java.lang.Class
- java.lang.ClassLoader
- java.lang.Runtime
- java.lang.SecurityManager

- java.lang.System
- java.lang.Thread
- java.lang.ThreadGroup
- java.lang.reflect.AccessibleObject
- java.net.Authenticator
- java.net.DatagramSocket
- java.net.HttpURLConnection
- java.net.InetAddress
- java.net.MulticastSocket
- java.net.ServerSocket
- java.net.Socket
- java.net.URL
- java.net.URLConnection
- java.net.URLClassLoader
- java.rmi.activation.ActivationGroup
- java.rmi.server.RMISocketFactory
- java.security.Identity
- java.security.IdentityScope
- java.security.Permission
- java.security.Policy
- java.security.Provider
- java.security.SecureClassLoader
- java.security.Security
- java.security.Signer
- java.util.Locale
- java.util.Zip

---

java.awt.Graphics2D

```
public abstract void setComposite(Composite comp)
java.Security.SecurityManager.checkPermission
java.awt.AWTPermission "readDisplayPixels"
```

El código de verificaciónde acceso para setComposite debería llamar a java.Security.SecurityManager.checkPermission y pasarle java.awt.AWTPermission

"readDisplayPixels" cuando el contexto Graphics2D dibuje un componente en la pantalla y el compuesto es un objeto personalizado en vez de un objeto AlphaComposite.

java.awt.Toolkit

```
    public void addAWTEventListener(
            AWTEventListener listener,
            long eventMask)
    public void removeAWTEventListener(
            AWTEventListener listener)
    checkPermission
    java.awt.AWTPermission "listenToAllAWTEvents"
~~~~~~~~~~
```

```
    public abstract PrintJob getPrintJob(
            Frame frame, String jobtitle,
            Properties props)
    checkPrintJobAccess
    java.lang.RuntimePermission "queuePrintJob"
~~~~~~~~~~
```

```
    public abstract Clipboard
            getSystemClipboard()
    checkSystemClipboardAccess
    java.awt.AWTPermission "accessClipboard"
~~~~~~~~~~
```

```
    public final EventQueue
            getSystemEventQueue()
    checkAwtEventQueueAccess
    java.awt.AWTPermission "accessEventQueue"
```

java.awt.Window

```
    Window()
    checkTopLevelWindow
    java.awt.AWTPermission
            "showWindowWithoutWarningBanner"
```

java.beans.Beans

```
    public static void setDesignTime(
            boolean isDesignTime)
    public static void setGuiAvailable(
            boolean isGuiAvailable)
    checkPropertiesAccess
    java.util.PropertyPermissions "*", "read,write"
```

java.beans.Introspector

```
    public static synchronized void
            setBeanInfoSearchPath(String path[])
    checkPropertiesAccess
    java.util.PropertyPermissions "*", "read,write"
```

java.beans.PropertyEditorManager

```
    public static void registerEditor(
            Class targetType,
            Class editorClass)
    public static synchronized void
            setEditorSearchPath(String path[])
    checkPropertiesAccess
    java.util.PropertyPermissions "*", "read,write"
```

java.io.File

```
    public boolean delete()
    public void deleteOnExit()
    checkDelete(String)
    java.io.FilePermission "{name}", "delete"
~~~~~~~~~~
```

```
    public boolean exists()
    public boolean canRead()
    public boolean isFile()
    public boolean isDirectory()
    public boolean isHidden()
    public long lastModified()
    public long length()
```

```
    public String[] list()
    public String[] list(FilenameFilter filter)
    public File[] listFiles()
    public File[] listFiles(FilenameFilter filter)
    public File[] listFiles(FileFilter filter)
    checkRead(String)
    java.io.FilePermission "{name}", "read"
~~~~~~~~~~

    public boolean canWrite()
    public boolean createNewFile()
    public static File createTempFile(
            String prefix, String suffix)
    public static File createTempFile(
            String prefix,  String suffix,
            File directory)
    public boolean mkdir()
    public boolean mkdirs()
    public boolean renameTo(File dest)
    public boolean setLastModified(long time)
    public boolean setReadOnly()
    checkWrite(String)
    java.io.FilePermission "{name}", "write"
```

java.io.FileInputStream

```
    FileInputStream(FileDescriptor fdObj)
    checkRead(FileDescriptor)
    java.lang.RuntimePermission "readFileDescriptor"
~~~~~~~~~~

    FileInputStream(String name)
    FileInputStream(File file)
    checkRead(String)
    java.io.FilePermission "{name}", "read"
```

java.io.FileOutputStream

```
    FileOutputStream(FileDescriptor fdObj)
    checkWrite(FileDescriptor)
    java.lang.RuntimePermission "writeFileDescriptor"
~~~~~~~~~~
```

```
    FileOutputStream(File file)
    FileOutputStream(String name)
    FileOutputStream(String name, boolean append)
    checkWrite(String)
    java.io.FilePermission "{name}", "write"
```

java.io.ObjectInputStream

```
    protected final boolean
            enableResolveObject(boolean enable);
    checkPermission
    java.io.SerializablePermission
            "enableSubstitution"
~~~~~~~~~~
```

```
    protected ObjectInputStream()
    protected ObjectOutputStream()
    checkPermission
    java.io.SerializablePermission
            "enableSubclassImplementation"
```

java.io.ObjectOutputStream

```
    protected final boolean
            enableReplaceObject(boolean enable)
    checkPermission
    java.io.SerializablePermission
            "enableSubstitution"
```

java.io.RandomAccessFile

```
    RandomAccessFile(String name, String mode)
    RandomAccessFile(File file, String mode)
    checkRead(String)
    java.io.FilePermission "{name}", "read"
```

En ambos métodos el modo es *r*.

```
~~~~~~~~~~
```

```
    RandomAccessFile(String name, String mode)
    checkRead(String) and checkWrite(String)
```

```
    java.io.FilePermission "{name}", "read,write"
```

En este método el modo es *rw*.

~~~~~~~~~~

java.lang.Class

```
    public static Class forName(
            String name, boolean initialize,
            ClassLoader loader)
    checkPermission
    java.lang.RuntimePermission("getClassLoader")
```

El código de verificación de acceso para este método llama a **checkPermission** y lo pasa a **java.lang.RuntimePermission("getClassLoader")** cuando **loader** es **null** y el cargador de la clase llamante no es **null**.

~~~~~~~~~~

```
    public Class[] getClasses()
    checkMemberAccess(this, Member.DECLARED)
    java.lang.RuntimePermission
            "accessDeclaredMembers"
    java.lang.RuntimePermission
            "accessClassInPackage.{pkgName}"
```

El código de verificación de acceso para esta clase y cada una de sus superclases llama a **checkMemberAccess(this, Member.DECLARED)**. Si la clase está en un paquete, **checkPackageAccess({pkgName})** también se llama. Por defecto, **checkMemberAccess** no requiere permiso si el cargador de clase de esta clase es el mismo que el de la otra. De otra forma requiere **java.lang.RuntimePermission "accessDeclaredMembers"**. Si la clase está en un paquete, también se requiere **java.lang.RuntimePermission "accessClassInPackage.{pkgName}"**.

~~~~~~~~~~

```
    public ClassLoader getClassLoader()
    checkPermission
    java.lang.RuntimePermission "getClassLoader"
```

Si el llamador de la clase llamante es **null**, o si el si es el mismo que el del ancestro del cargador de la clase para la clase cuyo cargador de clase está siendo solicitado, no se necesita permiso. De otra forma, se necesita **java.lang.RuntimePermission "getClassLoader"**.

~~~~~~~~~~

```
    public Class[] getDeclaredClasses()
```

```
public Field[] getDeclaredFields()
public Method[] getDeclaredMethods()
public Constructor[]
        getDeclaredConstructors()
public Field getDeclaredField(
        String name)
public Method getDeclaredMethod(...)
public Constructor
        getDeclaredConstructor(...)
checkMemberAccess(this, Member.DECLARED)
checkPackageAccess({pkgName})
java.lang.RuntimePermission
        "accessDeclaredMembers
java.lang.RuntimePermission
        "accessClassInPackage.{pkgName}
```

Si la clase está en un paquete, el código de verificación de acceso debería llamar a **checkPackageAccess({pkgName})** y pasarlo a **java.lang.RuntimePermission "accessClassInPackage.{pkgName}"**.

Si la clase no está en un paquete, el código de verificación de acceso para estos métodos debería llamar a **checkMemberAccess(this, Member.DECLARED)** y pasarlo a **java.lang.RuntimePermission "accessClassInPackage.{pkgName}"**.

~~~~~~~~~~

```
public Field[] getFields()
public Method[] getMethods()
public Constructor[] getConstructors()
public Field getField(String name)
public Method getMethod(...)
public Constructor getConstructor(...)
checkMemberAccess(this, Member.PUBLIC)
checkPackageAccess({pkgName})
java.lang.RuntimePermission
        "accessClassInPackage.{pkgName}"
```

Si la clase no está en un paquete, el código de verificación de acceso para estos métodos llama a **checkMemberAccess(this, Member.PUBLIC)**, pero no se pasa ningún permiso.

Si la clase está en un paquete, el código de verificación de acceso para estos métodos debería llamar a **checkPackageAccess({pkgName})** y pasarle **checkPackageAccess({pkgName})**.

~~~~~~~~~~

```
    public ProtectionDomain
            getProtectionDomain()
    checkPermission
    java.lang.RuntimePermission "getProtectionDomain"
```

java.lang.ClassLoader

```
    ClassLoader()
    ClassLoader(ClassLoader parent)
    checkCreateClassLoader
    java.lang.RuntimePermission "createClassLoader"
~~~~~~~~~~
```

```
    public static ClassLoader
            getSystemClassLoader()
    public ClassLoader getParent()
    checkPermission
    java.lang.RuntimePermission "getClassLoader"
```

Si el cargador de clases del llamante es **null** o es el mismo que el del ancestro del cargador de clases para la clase cuyo cargador está siendo solicitado, no se necesita permiso. De otra forma, se requiere **java.lang.RuntimePermission "getClassLoader"** .

java.lang.Runtime

```
    public Process exec(String command)
    public Process exec(String command,
            String envp[])
    public Process exec(String cmdarray[])
    public Process exec(String cmdarray[],
            String envp[])
    checkExec
    java.io.FilePermission "{command}", "execute"
~~~~~~~~~~
```

```
    public void exit(int status)
    public static void
            runFinalizersOnExit(boolean value)
    checkExit(status) where status is 0 for
            runFinalizersOnExit
    java.lang.RuntimePermission "exitVM"
~~~~~~~~~~
```

```
    public void load(String lib)
    public void loadLibrary(String lib)
    checkLink({libName})
    java.lang.RuntimePermission
            "loadLibrary.{libName}"
```

En estos métodos **{libName}** es el argumento **lib**, **filename** o **libname**.

java.lang.SecurityManager

```
    <all methods>
    checkPermission
    See Security Manager Methods.
```

java.lang.System

```
    public static void exit(int status)
    public static void
            runFinalizersOnExit(boolean value)
    checkExit(status) where status is 0 for
            runFinalizersOnExit
    java.lang.RuntimePermission "exitVM"
~~~~~~~~~~

    public static void load(String filename)
    public static void loadLibrary(
            String libname)
    checkLink({libName})
    java.lang.RuntimePermission
            "loadLibrary.{libName}"
```

En estos métodos **{libName}** es el argumento **lib**, **filename** o **libname**.

```
~~~~~~~~~~

    public static Properties getProperties()
    public static void setProperties(Properties props)
    checkPropertiesAccess
    java.util.PropertyPermission "*", "read,write"
~~~~~~~~~~

    public static String getProperty(String key)
    public static String getProperty(String key,
```

```
                                        String def)
    checkPropertyAccess
    java.util.PropertyPermission "{key}", "read"
~~~~~~~~~~

    public static void setIn(InputStream in)
    public static void setOut(PrintStream out)
    public static void setErr(PrintStream err)
    checkPermission
    java.lang.RuntimePermission "setIO"
~~~~~~~~~~

    public static String setProperty(String key,
                                        String value)
    checkPermission
    java.util.PropertyPermission "{key}", "write"
~~~~~~~~~~

    public static synchronized void
            setSecurityManager(SecurityManager s)
    checkPermission
    java.lang.RuntimePermission "setSecurityManager"
```

java.lang.Thread

```
    public ClassLoader getContextClassLoader()
    checkPermission
    java.lang.RuntimePermission "getClassLoader"
```

Si el cargador de clases del llamante es **null** o es el mismo que el del ancestro del cargador de clases para la clase cuyo cargador está siendo solicitado, no se necesita permiso. De otra forma, se requiere **java.lang.RuntimePermission "getClassLoader"**.

```
~~~~~~~~~~

    public void setContextClassLoader
            (ClassLoader cl)
    checkPermission
    java.lang.RuntimePermission
            "setContextClassLoader"
~~~~~~~~~~

    public final void checkAccess()
```

```
    public void interrupt()
    public final void suspend()
    public final void resume()
    public final void setPriority
            (int newPriority)
    public final void setName(String name)
    public final void setDaemon(boolean on)
    checkAccess(this)
    java.lang.RuntimePermission "modifyThread"
~~~~~~~~~~

    public static int
            enumerate(Thread tarray[])
    checkAccess({threadGroup})
    java.lang.RuntimePermission "modifyThreadGroup"
~~~~~~~~~~

    public final void stop()
    checkAccess(this).
    checkPermission
    java.lang.RuntimePermission "modifyThread"
    java.lang.RuntimePermission "stopThread"
```

El código de verificación de accesos debería llamar a **checkAccess** y pasarlo a **java.lang.RuntimePermission "modifyThread"**, a menos que thread actul intente parar otro thread distinto a sí mismo. En este caso, el código de verificación de acceso debería llamat a **checkPermission** y pasarlo a **java.lang.RuntimePermission "stopThread"**.

```
~~~~~~~~~~

    public final synchronized void
            stop(Throwable obj)
    checkAccess(this).
    checkPermission
    java.lang.RuntimePermission "modifyThread"
    java.lang.RuntimePermission "stopThread"
```

El código de verificación de accesos debería llamar a **checkAccess** y pasarlo a **java.lang.RuntimePermission "modifyThread"**, a menos que thread actul intente parar otro thread distinto a sí mismo. En este caso, el código de verificación de acceso debería llamat a **checkPermission** y pasarlo a **java.lang.RuntimePermission "stopThread"**.

```
~~~~~~~~~~

    Thread()
```

```
    Thread(Runnable target)
    Thread(String name)
    Thread(Runnable target, String name)
    checkAccess({parentThreadGroup})
    java.lang.RuntimePermission "modifyThreadGroup"
~~~~~~~~~~

    Thread(ThreadGroup group, ...)
    checkAccess(this) for ThreadGroup methods, or
    checkAccess(group) for Thread methods
    java.lang.RuntimePermission "modifyThreadGroup"
```

java.lang.ThreadGroup

```
    public final void checkAccess()
    public int enumerate(Thread list[])
    public int enumerate(Thread list[],
            boolean recurse)
    public int enumerate(ThreadGroup list[])
    public int enumerate(ThreadGroup list[],
            boolean recurse)
    public final ThreadGroup getParent()
    public final void
            setDaemon(boolean daemon)
    public final void setMaxPriority(int pri)
    public final void suspend()
    public final void resume()
    public final void destroy()
    checkAccess(this) for ThreadGroup methods, or
    checkAccess(group) for Thread methods
    java.lang.RuntimePermission "modifyThreadGroup"
~~~~~~~~~~

    ThreadGroup(String name)
    ThreadGroup(ThreadGroup parent,
    String name)
    checkAccess({parentThreadGroup})
    java.lang.RuntimePermission "modifyThreadGroup"
~~~~~~~~~~

    public final void interrupt()
    checkAccess(this)
    java.lang.RuntimePermission "modifyThreadGroup"
```

```
    java.lang.RuntimePermission "modifyThread"
```

El código de verificación de accesos para este método también requiere **java.lang.RuntimePermission "modifyThread"** porque el método **java.lang.Thread interrupt()** se llama para cada thread en el grupo de threads y todos sus subgrupos.

~~~~~~~~~~

```
    public final void stop()
    checkAccess(this)
    java.lang.RuntimePermission "modifyThreadGroup"
    java.lang.RuntimePermission "modifyThread"
    java.lang.RuntimePermission "stopThread"
```

El código de verificación de accesos para este método también requiere **java.lang.RuntimePermission "modifyThread"** porque el método **java.lang.Thread interrupt()** se llama para cada thread en el grupo de threads y todos sus subgrupos.

java.lang.reflect.AccessibleObject

```
    public static void setAccessible(...)
    public void setAccessible(...)
    checkPermission
    java.lang.reflect.ReflectPermission
            "suppressAccessChecks"
```

java.net.Authenticator

```
    public static PasswordAuthentication
      requestPasswordAuthentication(InetAddress addr,
                                    int port,
                                    String protocol,
                                    String prompt,
                                    String scheme)
    checkPermission
    java.net.NetPermission
            "requestPasswordAuthentication"
```

~~~~~~~~~~

```
    public static void
            setDefault(Authenticator a)
    checkPermission
    java.net.NetPermission "setDefaultAuthenticator"
```

java.net.DatagramSocket

```
    public void send(DatagramPacket p)
    checkMulticast(p.getAddress())
    checkConnect(p.getAddress().getHostAddress(),
                 p.getPort())
    java.net.SocketPermission((
                 p.getAddress()).getHostAddress(),
                 "accept,connect")
    java.net.SocketPermission "{host}","resolve"
```

El código de verificación de acceso para **send** llama a **checkMulticast** en los siguientes casos:

```
    if (p.getAddress().isMulticastAddress()) {
      java.net.SocketPermission(
        (p.getAddress()).getHostAddress(),
         "accept,connect")
    }
```

El código de verificación de acceso para **send** llama a **checkConnect** en los siguientes casos:

```
    else {
      port = p.getPort();
      host = p.getAddress().getHostAddress();
      if (port == -1) java.net.SocketPermission
                            "{host}","resolve";
      else java.net.SocketPermission
                            "{host}:{port}","connect"
    }
~~~~~~~~~~
```

```
    public InetAddress getLocalAddress()
    checkConnect({host}, -1)
    java.net.SocketPermission "{host}", "resolve"
~~~~~~~~~~
```

```
    DatagramSocket(...)
    checkListen({port})
```

El código de verificación de acceso para este método llama a **checkListen** y le pasa permisos de sockets de esta forma:

```
    if (port == 0)
      java.net.SocketPermission "localhost:1024-",
                                "listen";
    else
      java.net.SocketPermission "localhost:{port}",
                                "listen"
~~~~~~~~~~

    public synchronized void receive(DatagramPacket p)
    checkAccept({host}, {port})
    java.net.SocketPermission "{host}:{port}",
                              "accept"
```

java.net.HttpURLConnection

```
    public static void setFollowRedirects(boolean set)
    checkSetFactory
    java.lang.RuntimePermission "setFactory"
```

java.net.InetAddress

```
    public String getHostName()
    public static InetAddress[]
            getAllByName(String host)
    public static InetAddress getLocalHost()
    checkConnect({host}, -1)
    java.net.SocketPermission "{host}", "resolve"
```

java.net.MulticastSocket

```
    public void joinGroup(InetAddress mcastaddr)
    public void leaveGroup(InetAddress mcastaddr)
    checkMulticast(InetAddress)
    java.net.SocketPermission(
                    mcastaddr.getHostAddress(),
            "accept,connect")
~~~~~~~~~~

    public synchronized void
            send(DatagramPacket p, byte ttl)
    checkMulticast(p.getAddress(), ttl)
    checkConnect(p.getAddress().getHostAddress(),
```

```
                p.getPort())
    java.net.SocketPermission((
                p.getAddress()).getHostAddress(),
                "accept,connect")
    java.net.SocketPermission "{host}","resolve"
```

El código de verificación de acceso para **send** llama a **checkMulticast** en los siguientes casos:

```
    if (p.getAddress().isMulticastAddress()) {
      java.net.SocketPermission(
        (p.getAddress()).getHostAddress(),
         "accept,connect")
    }
```

El código de verificación de acceso para este método llama a **checkConnect** en los siguientes casos:

```
    else {
      port = p.getPort();
      host = p.getAddress().getHostAddress();
      if (port == -1) java.net.SocketPermission
            "{host}","resolve"
      else java.net.SocketPermission
            "{host}:{port}","connect"
    }
~~~~~~~~~~

    MulticastSocket(...)
    checkListen({port})
```

El código de verificación de acceso para este método llama a **checkListen** en los siguientes casos:

```
    if (port == 0)
      java.net.SocketPermission
            "localhost:1024-", "listen";
    else
      java.net.SocketPermission
            "localhost:{port}","listen"
```

java.net.ServerSocket

```
    ServerSocket(...)
    checkListen({port})
```

El código de verificación de acceso para este método llama a **checkListen** en los siguientes casos:

```
    if (port == 0)
      java.net.SocketPermission
            "localhost:1024-","listen";
    else
      java.net.SocketPermission
            "localhost:{port}","listen"
~~~~~~~~~~
```

```
    public Socket accept()
    protected final void implAccept(Socket s)
    checkAccept({host}, {port})
    java.net.SocketPermission
            "{host}:{port}", "accept"
~~~~~~~~~~
```

```
    public static synchronized void
            setSocketFactory(...)
    checkSetFactory
    java.lang.RuntimePermission "setFactory"
```

java.net.Socket

```
    public static synchronized void
            setSocketImplFactory(...)
    checkSetFactory
    java.lang.RuntimePermission "setFactory"
~~~~~~~~~~
```

```
    Socket(...)
    checkConnect({host}, {port})
    java.net.SocketPermission
            "{host}:{port}", "connect"
```

java.net.URL

```
    public static synchronized void
            setURLStreamHandlerFactory(...)
    checkSetFactory
    java.lang.RuntimePermission "setFactory"
```

~~~~~~~~~~

```
URL(...)
checkPermission
java.net.NetPermission "specifyStreamHandler"
```

java.net.URLConnection

```
public static synchronized void
        setContentHandlerFactory(...)
public static void setFileNameMap(
        FileNameMap map)
checkSetFactory
java.lang.RuntimePermission "setFactory"
```

java.net.URLClassLoader

```
URLClassLoader(...)
checkCreateClassLoader
java.lang.RuntimePermission "createClassLoader"
```

java.rmi.activation.ActivationGroup

```
public static synchronized ActivationGroup
        createGroup(...)
public static synchronized void setSystem(
        ActivationSystem system)
checkSetFactory
java.lang.RuntimePermission "setFactory"
```

java.rmi.server.RMISocketFactory

```
public synchronized static void setSocketFactory(...)
checkSetFactory
java.lang.RuntimePermission "setFactory"
```

java.security.Identity

```
public void addCertificate(...)
checkSecurityAccess("addIdentityCertificate")
```
~~~~~~~~~~

```
    java.security.SecurityPermission
            "addIdentityCertificate"
~~~~~~~~~~

    public void removeCertificate(...)
    checkSecurityAccess("removeIdentityCertificate")
    java.security.SecurityPermission
            "removeIdentityCertificate"
~~~~~~~~~~

    public void setInfo(String info)
    checkSecurityAccess("setIdentityInfo")
    java.security.SecurityPermission
            "setIdentityInfo"
~~~~~~~~~~

    public void setPublicKey(PublicKey key)
    checkSecurityAccess("setIdentityPublicKey")
    java.security.SecurityPermission
            "setIdentityPublicKey"
~~~~~~~~~~

    public String toString(...)
    checkSecurityAccess("printIdentity")
    java.security.SecurityPermission
            "printIdentity"
```

java.security.IdentityScope

```
    protected static void setSystemScope()
    checkSecurityAccess("setSystemScope")
    java.security.SecurityPermission
            "setSystemScope"
```

java.security.Permission

```
    public void checkGuard(Object object)
    checkPermission(this)
```

Este objeto Permission es el permiso chequeado.

java.security.Policy

```
    public static Policy getPolicy()
    checkPermission
    java.security.SecurityPermission "getPolicy"
~~~~~~~~~~
```

```
    public static void setPolicy(Policy policy);
    checkPermission
    java.security.SecurityPermission "setPolicy"
~~~~~~~~~~
```

java.security.Provider

```
    public synchronized void clear()
    checkSecurityAccess("clearProviderProperties."
                        +{name})
    java.security.SecurityPermission
            "clearProviderProperties.{name}"
```

En este método *name* es el nombre del proveedor.

```
~~~~~~~~~~
```

```
    public synchronized Object put(Object key,
                                   Object value)
    checkSecurityAccess("putProviderProperty."
                                   +{name})
    java.security.SecurityPermission
            "putProviderProperty.{name}"
```

En este método *name* es el nombre del proveedor.

```
~~~~~~~~~~
```

```
    public synchronized Object remove(Object key)
    checkSecurityAccess("removeProviderProperty."
                        +{name})
    java.security.SecurityPermission
            "removeProviderProperty.{name}"
```

En este método *name* es el nombre del proveedor.

java.security.SecureClassLoader

```
    SecureClassLoader(...)
    checkCreateClassLoader
    java.lang.RuntimePermission "createClassLoader"
```

java.security.Security

```
    public static void getProperty(String key)
    checkPermission
    java.security.SecurityPermission "getProperty.{key}"
~~~~~~~~~~

    public static int addProvider(Provider provider)
    public static int insertProviderAt(
                            Provider provider,
                            int position);
    checkSecurityAccess("insertProvider."
                            +provider.getName())
    java.security.SecurityPermission
                    "insertProvider.{name}"
~~~~~~~~~~

    public static void removeProvider(String name)
    checkSecurityAccess("removeProvider."+name)
    java.security.SecurityPermission "removeProvider.{name}"
~~~~~~~~~~

    public static void setProperty( String key,
                                    String datum)
    checkSecurityAccess("setProperty."+key)
    java.security.SecurityPermission
                                    "setProperty.{key}"
```

java.security.Signer

```
    public PrivateKey getPrivateKey()
    checkSecurityAccess("getSignerPrivateKey")
    java.security.SecurityPermission
            "getSignerPrivateKey"
~~~~~~~~~~
```

```
public final void setKeyPair(KeyPair pair)
checkSecurityAccess("setSignerKeypair")
java.security.SecurityPermission
        "setSignerKeypair"
```

java.util.Locale

```
public static synchronized void setDefault(
        Locale newLocale)
checkPermission
java.util.PropertyPermission
        "user.language","write"
```

java.util.zip.ZipFile

```
ZipFile(String name)
checkRead
java.io.FilePermission "{name}","read"
```

Apéndice C: Métodos del Controlador de Seguridad

Esta tabla lista los permisos chequeados mediante las implementaciones de los métodos de **java.lang.SecurityManager**. Cada método de chequeo llama al método **SecurityManager.checkPermission** con el permiso indicado, excepto para los permisos **checkConnect** y **checkRead** que toman un argumento de contexto. Los métodos **checkConnect** y **checkRead** esperan que el contexto sea un **AccessControlContext** y llaman al método **checkPermission** del permiso con el permiso especificado.

```
public void checkAccept(String host, int port);
java.net.SocketPermission "{host}:{port}", "accept";

public void checkAccess(Thread g);
java.lang.RuntimePermission "modifyThread");

public void checkAccess(ThreadGroup g);
java.lang.RuntimePermission "modifyThreadGroup");

public void checkAwtEventQueueAccess();
java.awt.AWTPermission "accessEventQueue";

public void checkConnect(String host, int port);
if (port == -1)
  java.net.SocketPermission "{host}","resolve";
else
  java.net.SocketPermission "{host}:{port}","connect";

public void checkConnect(String host, int port,
                         Object context);
if (port == -1)
  java.net.SocketPermission "{host}","resolve";
else
  java.net.SocketPermission "{host}:{port}","connect";

public void checkCreateClassLoader();
java.lang.RuntimePermission "createClassLoader";

public void checkDelete(String file);
java.io.FilePermission "{file}", "delete";

public void checkExec(String cmd);
if (cmd is an absolute path)
  java.io.FilePermission "{cmd}", "execute";
```

```
else
  java.io.FilePermission "-", "execute";

public void checkExit(int status);
java.lang.RuntimePermission "exitVM");

public void checkLink(String lib);
java.lang.RuntimePermission "loadLibrary.{lib}";

public void checkListen(int port);
if (port == 0)
  java.net.SocketPermission "localhost:1024-","listen";
else
  java.net.SocketPermission "localhost:{port}","listen";

public void checkMemberAccess(Class clazz, int which);
if (which != Member.PUBLIC) {
 if (currentClassLoader() != clazz.getClassLoader()) {
  checkPermission(
   new java.lang.RuntimePermission(
                   "accessDeclaredMembers"));
 }
}

public void checkMulticast(InetAddress maddr);
java.net.SocketPermission(
              maddr.getHostAddress(),"accept,connect");

public void checkMulticast(InetAddress maddr, byte ttl);
java.net.SocketPermission(
              maddr.getHostAddress(),"accept,connect");

public void checkPackageAccess(String pkg);
java.lang.RuntimePermission
                 "accessClassInPackage.{pkg}";

public void checkPackageDefinition(String pkg);
java.lang.RuntimePermission
                 "defineClassInPackage.{pkg}";

public void checkPrintJobAccess();
java.lang.RuntimePermission "queuePrintJob";

public void checkPropertiesAccess();
java.util.PropertyPermission "*", "read,write";
```

```
public void checkPropertyAccess(String key);
java.util.PropertyPermission "{key}", "read,write";

public void checkRead(FileDescriptor fd);
java.lang.RuntimePermission "readFileDescriptor";

public void checkRead(String file);
java.io.FilePermission "{file}", "read";

public void checkRead(String file, Object context);
java.io.FilePermission "{file}", "read";

public void checkSecurityAccess(String action);
java.security.SecurityPermission "{action}";

public void checkSetFactory();
java.lang.RuntimePermission "setFactory";

public void checkSystemClipboardAccess();
java.awt.AWTPermission "accessClipboard";

public boolean checkTopLevelWindow(Object window);
java.awt.AWTPermission "showWindowWithoutWarningBanner";

public void checkWrite(FileDescriptor fd);
java.lang.RuntimePermission "writeFileDescriptor";

public void checkWrite(String file);
java.io.FilePermission "{file}", "write";

public SecurityManager();
java.lang.RuntimePermission "createSecurityManager";
```

Epílogo...

Esta sección no forma parte del tutor original de Sun.

Podeís encontrar la versión original en Inglés de este **"Curso sobre Programación Avanzada en Java 2"** en las páginas de Trainings OnLine de la propia Sun MicroSystem.

Los nombres de los autores de la versión original son:

- **Calvin Austin**
- **Monica Pawlan**
- **Tony Squier** como autor invitado.
