Java a Tope:

JavaMail

(JavaMail en ejemplos)

Sergio Gálvez Rojas
Ignacio García Sucino

UNIVERSIDAD DE MÁLAGA

JAVA A TOPE: JAVAMAIL EN EJEMPLOS. EDICIÓN ELECTRÓNICA

AUTORES: SERGIO GÁLVEZ ROJAS
IGNACIO GARCÍA SUCINO

ILUSTRACIÓN DE PORTADA: JOSÉ MIGUEL GÁLVEZ ROJAS
JAVIER MACÍAS GÁLVEZ

Sun, el logotipo de Sun, Sun Microsystems y Java son marcas o marcas registradas de Sun Microsystems Inc. en los EE.UU. y otros países.

Depósito Legal: MA-1287-2006
ISBN: 84-690-0697-5

Java a tope:

JavaMail

(JavaMail en ejemplos)

Sergio Gálvez Rojas
Doctor Ingeniero en Informática

Ignacio García Sucino
Ingeniero Técnico en Informática de Sistemas

Dpto. de Lenguajes y Ciencias de la Computación
E.T.S. de Ingeniería Informática
Universidad de Málaga

Índice

Índice

Prólogo

El correo electrónico es, actualmente, uno de los principales medios de comunicación electrónicos asíncronos. De hecho, el primer uso que se dio a la, hoy omnipresente, red internet fue precisamente el de comunicar entre sí a los miembros de un equipo de investigación dispersos geográficamente mediante *emails*. Y aún hoy día, a pesar del enorme auge de la *World Wide Web*, el tráfico que soporta internet debido a las comunicaciones vía correo electrónico supone un buen porcentaje del total.

Aparte de la indudable importancia de este tipo de comunicaciones realizadas a través de aplicaciones clientes como Eudora, Outlook, Thunderbird, etc. en las que intervienen directamente personas escribiendo y leyendo los correos, también resulta muy interesante el que las aplicaciones sean capaces de enviar y recibir automáticamente mensajes ante ciertas circunstancias detectadas de forma autónoma. Por ejemplo, resulta conveniente informar a un administrador de bases de datos cuándo se ha alcanzado un tamaño crítico en los espacios de tablas disponibles para almacenar información, o cuándo se ha intentado algún tipo de acceso sospechoso no autorizado. Análogamente, una aplicación puede ser capaz de recibir mensajes con un formato predeterminado y procesarlos autónomamente: pedidos de almacén, solicitudes de vigilancias para exámenes, etc. informando asimismo de la correcta o incorrecta recepción de las peticiones.

Para todo ello, el lenguaje Java incorpora la API JavaMail que permite gestionar cualquier tipo de correo electrónico actual, ya sea a través del protocolo POP, IMAP o cualquier otro que pueda surgir en el futuro. Con JavaMail es posible manipular mensajes de texto plano o HTML, e incluso manejar múltiples adjuntos, imágenes incrustadas, prioridades o solicitar acuse de recibo por parte del destinatario.

Aún más importante, JavaMail es relativamente fácil de manejar siempre y cuando se conozcan sus fundamentos. Por este motivo, los capítulos que viene a continuación establecen las bases de funcionamiento y, ejemplo a ejemplo, se va profundizando en cada uno de los temas más interesantes proponiendo, en cada caso, el código al completo para que el lector pueda hacer uso de él directamente.

Capítulo 1

Fundamentos del correo electrónico

1.1 El correo electrónico: definición e historia

El correo electrónico es un servicio de red que permite intercambiar mensajes entre distintos usuarios de manera asíncrona; estos mensajes pueden contener o no ficheros adjuntos. Según la el diccionario de la RAE el correo electrónico se define como: «Sistema de comunicación personal por ordenador a través de redes informáticas».

El correo electrónico fue creado por Ray Tomlinson en 1971. Ray Tomlison se graduó en ingeniería eléctrica en el Instituto de Tecnología de Massachusetts (MIT) y entró a trabajar en la empresa BBN[1] en 1967, poco antes de que su empresa recibiera el encargo de trabajar para ARPANET (red de ordenadores creada por encargo del Departamento de Defensa de los Estados Unidos como medio de comunicación para los diferentes organismos del país), la red precursora de Internet. En esa empresa utilizó un programa llamado SNDMSG para enviar mensajes entre los distintos usuarios de un mismo ordenador. Eran tiempos en que los usuarios trabajan en informática mediante "terminales tontas", es decir, una pantalla y un teclado, sin memoria ni procesador propios, conectadas a un ordenador central. Estas terminales recibieron ese nombre porque no realizaban ningún cómputo, sólo eran usadas para enviar datos de forma asíncrona a la computadora principal para que ésta los procesara.

En septiembre de 1971, cuando la empresa BBN ya estaba conectada a ARPANET y haciendo un amplio uso de ella, Ray adaptó el programa SNDMSG de forma que sirviera también para enviar mensajes entre diferentes ordenadores conectados en red. Fue entonces cuando se le ocurrió utilizar el símbolo @ para unir el nombre del usuario y el de el ordenador que utilizaba como servidor. Según algunas teorías se trataba de utilizar un símbolo que estuviera en todos los teclados pero que no apareciera en los nombres de las personas ni de los ordenadores. En realidad, la @ estaba en los teclados pero no se utilizaba prácticamente para nada por lo que no era probable que entrara en conflicto con ninguna otra cosa. Otras teorías, dicen que

[1]BBN es una empresa de alta tecnología que en la actualidad trabaja en proyectos de vanguardia como seguridad en Internet, reconocimiento avanzado del habla o criptografía cuántica. Sus siglas se corresponden con las de sus fundadores Leo Beranek, Richard Bolt y Robert Newman.

el término empleado como divisor entre el usuario y la máquina fue la arroba porque ésta en inglés se pronuncia "at" (en), lo que hace que una dirección **x@y** se lea como "usuario x en máquina y". A partir de ahí la expansión fue imparable. Surgieron los primeros programas de correo electrónico como RD (el primero en crearse), NRD, WRD y MSG considerado el primer programa moderno de gestión de correo electrónico. Ya en en 1973 un estudio señalaba que el correo electrónico representaba el 75% del tráfico en ARPANET.

En 1989 desapareció ARPANET y, por otro lado, el investigador Tim Berners-Lee del centro europeo CERN en Suiza, desarrolló una propuesta de sistema de hipertexto, lo que daría lugar a la World Wide Web (www).

En la actualidad el correo electrónico es la aplicación más popular de Internet, y se calcula que la cantidad de información que se mueve a través del correo supera varias veces a la información contenida en páginas Web.

En cuanto a España, el primer correo electrónico a través de Internet se envió a finales de 1985 desde la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Madrid, donde se montó el primer nodo español conectado a la red EUNET.

EUNET era la parte europea de la red USENET, que daba servicio de correo electrónico a la mayoría de universidades de los Estados Unidos. USENET -el equivalente en sus inicios a un foro de debate e intercambio en la Internet que conocemos hoy día- había convergido con Internet a principios de los ochenta, creando un dominio único de correo electrónico basado en el formato RFC 822. Este estándar especifica la sintaxis de los mensajes de correo electrónico de tipo texto enviados a través de Internet. RFC son las siglas de *Request For Comment* o Petición de Comentarios; se trata de una serie de documentos o informes técnicos que edita el IAB (*Internet Architecture Board* - Consejo de Arquitectura de Internet), con el propósito de regular los mecanismos de trabajo y procedimientos de comunicación a través de Internet

El primer nodo de correo instalado en España se denominó «Goya» y se conectaba con el nodo central de EUNET en Holanda, en el Mathematical Centre de Amsterdam. En 1992 se creó el primer proveedor comercial de servicios de Internet en España: Goya Servicios Telemáticos.

En la actualidad se extienden distintos tipos de envíos perniciosos y amenazas a través de este sistema de comunicación. Cabría destacar como los más empleados los virus, los hoax y el spam. El ataque a través de virus consiste en enviar ficheros adjuntos infectados por algún virus. El hoax es un mensaje con contenido falso o engañoso y que normalmente es distribuido en cadena. El spam coincide con el hoax en que es distribuido de forma masiva y satura las redes, pero sin embargo suele tratarse de información publicitaria, no solicitada por el receptor del mensaje.

En España el spam está terminantemente prohibido por la Ley de Servicios de la Sociedad de la Información y de Comercio Electrónico (LSSICE), publicada en el BOE del 12 de julio de 2002.

1.2 Funcionamiento del correo electrónico

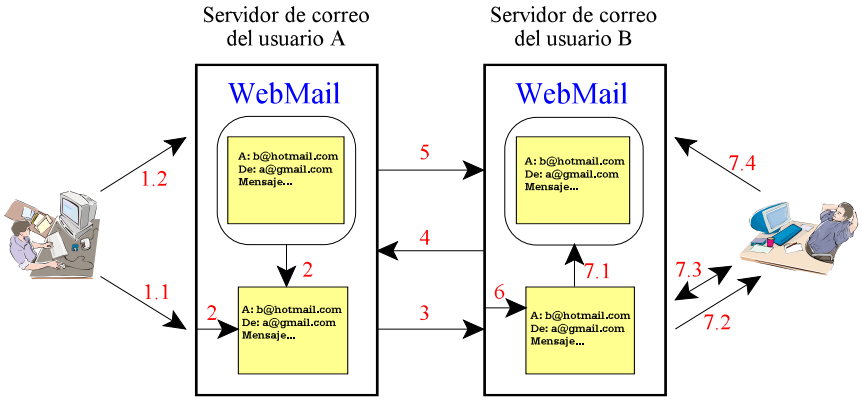

Figura 1.1 Esquema de funcionamiento del correo electrónico

La figura 1.1 muestra el funcionamiento básico del envío y recepción de un mensaje de correo electrónico entre dos usuarios llamados A y B.

Los pasos que se siguen para enviar un mensaje desde **A** hasta **B** son:

1.- El usuario A se prepara para enviar un correo electrónico al usuario B. Este procedimiento se puede realizar de dos formas:

1.1.- El usuario A usa un programa de tipo MUA (*Mail User Agent* o Agente de Usuario de Correo), que son programas que se usan para crear y enviar (entre otras funciones) correos electrónicos, y entre los que se pueden destacar Microsoft Office Outlook , Mozilla Thunderbird, Eudora, Pegasus, etc. El usuario compone el mensaje y selecciona la opción de enviar. El MUA del usuario **A** le indica al servidor de correo de este usuario quién está enviando el mensaje y quiénes son los destinatarios, y a continuación envía el mensaje usando el protocolo SMTP (*Simple Mail Transfer Protocol* - Protocolo Simple de Transferencia de Correo Electrónico). Cuando el servidor recoge el mensaje y acepta transferirlo hasta su destino, el MUA informa al usuario **A** que el envió ha tenido éxito.

1.2.- El usuario **A** usa directamente su servidor de correo electrónico vía web.

Esta forma de acceder al correo electrónico se conoce como *WebMail* o Correo Web. Se trata de un servicio que permite el acceso al correo mediante páginas web, usando para ello un navegador. Para ello, el usuario ha de autenticarse en el servidor introduciendo su nombre de usuario y contraseña; a continuación compone el mensaje y selecciona la opción de enviar. Entre las ventajas de este método se puede decir que los mensajes no se descargan al ordenador, es posible acceder desde cualquier máquina sin que tenga que tener ningún *software* específico (todos los sistemas operativos suelen incluir un navegador web) y la creación de una cuenta de correo es más sencilla que desde un programa MUA. La principal desventaja es que el espacio de almacenamiento de mensajes se limita a lo que ofrece el servidor de correo, en lugar de la capacidad de almacenamiento del propio disco duro que ofrece un programa MUA. Además requiere una conexión permanente a Internet mientras lo usemos y los mensajes ocupan bastante más espacio pues van embebidos en HTML, lo que implica que tardan más en enviarse.

2.- El mensaje es almacenado en el servidor de correo del usuario **A**.

3.- El servidor de correo del usuario **A** solicita al del usuario **B** el nombre del servidor al que tiene que enviar el mensaje. El servidor del usuario **B** se conoce gracias a la parte de la dirección de correo que sigue a la arroba, ej. @gmail.com.

4.- El servidor de correo del usuario **B** le responde al del usuario **A**, enviándole un registro MX (*MX record*), que es un registro de correo. En este registro constan las direcciones de correo que pertenecen a un dominio concreto y la máquina que se corresponde con cada dirección. Así cuando se envía un correo, se comprueba en la lista de registros MX de ese dominio si contiene esa dirección de correo, no aceptándola en caso contrario. En el registro MX pueden aparecer distintas máquinas, estableciéndose prioridades entre ellas o balanceando la carga de correo por las distintas máquinas. Por ejemplo un servidor le devolverá al otro una lista con una serie de máquinas en orden de prioridad. Si al enviar el correo a la primera máquina esta fallase o estuviese saturada, se enviaría a la segunda, y así sucesivamente.

5.- El servidor de correo del usuario **A** envía entonces el mensaje al servidor correspondiente a esa dirección de correo, usando el protocolo SMTP.

6.- El mensaje es almacenado en el servidor de correo del usuario **B** y se coloca en el buzón de correo del usuario.

7.- El último paso sería la lectura del mensaje por parte del usuario **B**, lo que se puede llevar a cabo de 3 formas:

7.1.- El usuario **B** accede directamente a su servidor de correo vía *webmail* y lee sus mensajes del servidor sin descargarlos.

7.2.- El usuario **B** accede a su correo a través de un programa de tipo MUA. Si lo hace de esta forma tiene dos opciones:

7.2.1.- Descargar los mensajes a su máquina usando el protocolo POP3 (*Post Office Protocol* - Protocolo de Oficina Postal versión 3).

7.2.2.- Descargar los mensajes a su máquina o leerlos directamente en el servidor usando el protocolo IMAP (*Internet Message Access Protocol* - Protocolo de Internet para el Acceso a Mensajes Electrónicos).

1.3 Estructura de un mensaje de correo electrónico

El estándar SMTP usa el estándar RFC 822 como formato para la creación de mensajes enviados a través de este protocolo. En el año 2001 este RFC fue actualizado por el RFC 2822, que es un superconjunto del anterior y permite mayor flexibilidad. En el contexto del RFC 822 los mensajes se dividen en sobre y contenido. El sobre (en la mayoría de los sitios se denomina cabecera) contiene la información necesaria para que se complete la transmisión y la entrega. El contenido es la información que va a ser entregada en el destino.

Según el RFC 822 el estándar afecta sólo al contenido del mensaje y no al sobre, si bien algunos sistemas de mensajes pueden necesitar usar información del contenido para formar el sobre, por lo que el estándar se encarga también de facilitar la adquisición de estos datos por parte de dichos programas.

A continuación se detallan los principales campos del sobre o cabecera de un mensaje; el nombre de cada uno de estos campos está definido en el estándar en su forma inglesa:

- *From* (De): se trata de la dirección de correo del usuario que envía el mensaje. Este campo puede ser alterado por el emisor para que aparezca otra dirección. En este caso puede aparecer en la cabecera un mensaje de X-Autentication-Warning indicando que la dirección que aparece en el *From* puede no corresponderse con la que realmente ha enviado el correo.
- *To* (Para): ss la dirección y nombre (opcional) del usuario al que va dirigido el correo. En el campo *To* pueden aparecer varias direcciones.
- *Cc* (*Carbon Copy* - Copia en Papel Carbón): dirección y nombre (opcional) del usuario al que queremos que se envíe una copia del mensaje. En este campo pueden aparecer varias direcciones o ninguna. El término copia en papel carbón procede de las antiguas máquinas de escribir en las que, para sacar varias copias de un documento, se colocaba bajo el original un papel de carbón y otro papel en blanco (la copia), de manera que al ir escribiendo,

los golpes de los martillos de la máquina imprimían en el original y, por presión, dejaban una marca de carboncillo con la forma de la letra en la copia que había por debajo.

- *Bcc* (*Blind Carbon Copy* - Copia Oculta en Papel Carbón): es equivalente al campo *Cc* con la diferencia de que la dirección que escribamos en este campo no será vista por el resto de usuarios que reciban el mensaje. Puede aparecer también como *Cco* (Copia de carbón oculta). En este campo pueden aparecer varias direcciones o ninguna.
- *Subject* (Asunto): breve descripción o título que queremos que figure en el mensaje, y que informa al destinatario de la naturaleza de éste. Este campo se puede dejar en blanco, pero no es recomendable.

Los campos que se explican a continuación no suelen ser mostrados por defecto por los MUA o al consultar el correo desde el servidor. Para verlos suele ser necesario seleccionar opciones como encabezado completo, ver todas las cabeceras, etc.:

- *Return-Path*: este campo es añadido por el último servidor por el que pasa el mensaje antes de ser entregado a su destino, y se usa para hallar la ruta de retorno en caso de devolución del mensaje.
- *Received*: en este campo están registrados todos los servidores por los que pasa el mensaje, por si se necesita realizar un seguimiento del mismo.
- *Message-ID*: este campo contiene un identificador que hace que el mensaje asociado a él sea único en todo Internet.
- *X-Priority*: indica la prioridad con la que el lector debe considerar al mensaje.
- *X-Mailer*: Nombre del *software* o aplicación utilizada para transferir el correo.
- *MIME-Version*: indica la versión de MIME que utiliza el mensaje. MIME son las siglas de *Multipurpose Internet Mail Extension* (Extensiones Multipropósito de Correo por Internet). Básicamente, permiten enviar información adicional junto con el texto de un correo, de manera que sepueda identificar correctamente cada uno de estos bloque de información y la naturaleza de su contenido.
- *Content-Type*: indica la naturaleza de los bytes que constituyen el contenido del mensaje para que el sistema receptor del mismo pueda tratar los datos de forma apropiada: imagen, audio, video, texto, etc.
- *Content-Transfer-Encoding*: indica el tipo de codificación que se ha usado con el mensaje, para que el sistema receptor sepa como decodificarlo.

Existen muchos más campos (no obligatorios en la mayoría de los casos) que forman la cabecera de un mensaje. El lector que desee conocer todas las posibilidades puede consultar el RFC 2822 y los RFC que tratan sobre MIME: RFC 2045 a 2049.

1.4 Direcciones de correo electrónico

Las direcciones de correo electrónico se corresponden de forma unívoca con un usuario, es decir, identifican de forma única a cada usuario dentro de una red. Así cuando se envía un correo electrónico (a los que denotaremos a partir de ahora como **correo-e**) se asegura que este llegará sólo al usuario deseado.

Las direcciones tienen la forma **usuario@dominio** donde el **usuario** indica la cuenta o buzón dentro del servidor del usuario al que va destinado el mensaje. A continuación aparece el **dominio**, es decir, el nombre de la máquina donde se encuentra dicha cuenta. Este nombre es único en todo Internet, por lo que el propietario del servidor de correo tendrá que haber registrado su dominio previamente.

Para que SMTP conozca la dirección de red de un dominio dado, se usan los servicios de un servidor DNS (*Domain Name System* - Sistema de Nombres de Dominio). El servidor DNS se encarga de traducir los nombres de dominio a direcciones IP (*Internet Protocol* - Protocolo de Internet) y viceversa. El servidor DNS consiste en una base de datos distribuida, jerárquica, replicada y tolerante a fallos. Cuando escribimos una dirección, ésta está compuesta por palabras separadas por puntos. La dirección leída de derecha a izquierda se asemeja a recorrer un árbol desde la raíz hasta una hoja, donde cada etiqueta está asociada con un servidor primario y varios secundarios que irán encaminando la dirección hasta la máquina correspondiente. Los RFC que definen el DNS son actualmente los RFC 1034 y 1035.

La idea de esto es usar direcciones más cómodas que si tuviésemos que especificar direcciones IP directamente. Estas direcciones son las que realmente utilizan en Internet cada máquina y cada dispositivo de enrutamiento o *router*. Son direcciones formadas por cuatro números entre 0 y 255 separados por punto, de la forma [0-255].[0-255].[0-255].[0-255] y existen cinco tipos de direcciones IP en función del tipo de red a que están destinadas, y se distinguen por los bits más significativos del primer número de la dirección IP. Así, tenemos las clases A, B, C, D y E; las tres primeras se utilizan en función del tamaño de la red, las del tipo D permiten hacer multidifusión (*multicasting*), es decir, enviar un paquete de datos a múltiples destinos; y las del tipo E están reservadas para un uso futuro.

Por otro lado, los nombres de los dominios se dividen en diferentes categorías:

- .com: dominio asociado a empresas o negocios.

- .edu: para universidad e instituciones educativas.
- .net: especifica una determinada red.
- .gov: organismos gubernamentales.
- Indicativos del país en que se encuentra el servidor:
  - ✓ .es (España)
  - ✓ .us (Estados Unidos)
  - ✓ .de (Alemania), etc.

1.5 Protocolos y extensiones relacionadas

A continuación se comentan los protocolos más importantes que intervienen en el funcionamiento del correo electrónico, y que son los que JavaMail incluye por defecto. Además se comentan las extensiones MIME, que hoy día aparecen prácticamente en todas las aplicaciones de correo electrónico.

1.5.1 POP (*Post Office Protocol*)

El protocolo POP (Protocolo de Oficina Postal), actualmente en su versión 3 referido como POP3, se usa para que un usuario pueda descargar su correo desde el servidor a su máquina local. El estándar de este protocolo se encuentra en el documento RFC 1939 (http://www.rfc-es.org/rfc/rfc1939-es.txt).

Se podría decir de forma somera, que POP3 se basa en 3 fases: la fase de autorización, la fase de transacción, y la fase de actualización, junto con una serie de comandos.

- El primer paso cuando un cliente se conecta a un servidor POP es establecer una conexión en el puerto 110 de éste. Una vez establecida la conexión, el cliente POP recibe una invitación y se produce una serie de intercambios de comandos POP entre el cliente y el servidor. Durante este intercambio el cliente debe identificarse, con nombre de usuario y contraseña, ante el servidor POP en lo que se conoce como fase de autorización o auntenticación.
- La siguiente fase es la fase de transacción en la que se pueden usar los siguientes comandos:
  - ✓ LIST: para mostrar los mensajes.
  - ✓ RETR: para recuperar los mensajes.
  - ✓ DELE: para borrar mensajes.
  - ✓ STAT: que devuelve información sobre un buzón.
  - ✓ NOOP: el servidor no hace nada, sólo devuelve una respuesta positiva.
  - ✓ RSET: elimina las marcas de los mensajes marcados como borrados en

el servidor.

✓ QUIT: en esta fase hace que se pase a la fase de actualización.

- La última fase es la fase de actualización, en la que se eliminan los mensajes marcados y se eliminan los recursos asociados a la conexión.

Por último señalar que los servidores basados en POP3 sólo ofrecen al usuario acceso a un único buzón, mostrado normalmente a modo de carpeta por el servidor. En otras palabras, el servidor no mantiene ninguna estructura de carpetas, sino que ésta debe ser creada y mantenida por un programa MUA.

1.5.2 SMTP (*Simple Mail Transfer Protocol*)

El protocolo SMTP es un protocolo que se usa para el envío de correo electrónico (Protocolo Simple de Transferencia de Correo Electrónico). SMTP transfiere los mensajes desde la máquina cliente al servidor, quedándose este último con el mensaje, si se trata del destino final, o transfiriéndolo a otro servidor, también vía SMTP, para que el mensaje llegue al servidor del destinatario. El protocolo completo está descrito en los RFC 821 y 822, donde se define el funcionamiento de SMTP y el formato de mensaje respectivamente. Estos RFC han sido actualizados por los RFC 2821 y 2822 (http://www.faqs.org/rfcs/rfc2821.html y http://www.faqs.org/rfcs/rfc2822.html).

Algunas de las características de SMTP son las siguientes:

- Utiliza el puerto 25 para establecer la conexión.
- Se pueden comunicar entre sí servidores de plataformas distintas.
- Dos sistemas que intercambien correo mediante SMTP no necesitan una conexión activa, ya que posee sistema de almacenamiento y reenvío de mensajes, es decir, si el receptor no está disponible el mensaje será reenviado posteriormente.

Algunos de los principales problemas de SMTP en su forma original son:

- Falta de soporte para idiomas distintos del inglés pues solo acepta caracteres ASCII de 7 bits, lo que impedía usar caracteres propios de otros idiomas como la ñ o los acentos.
- Los datos no viajan de forma segura.
- Facilita la expansión del spam, pues no requiere autenticación en el servidor.

Estos problemas se han ido solventando con distintas revisiones de SMTP. El uso de códigos de 8 bits se puede conseguir empleando ESMTP (*Extended SMTP* - SMTP Extendido) o mediante extensiones MIME. También existe una revisión para

trabajar con SMTP seguro sobre TLS[2] , lo que permite un transporte más seguro de los datos; esta revisión se define en el RFC 3207. Por último, comentar la extensión definida en el RFC 2554 que introduce el comando AUTH, lo que permite autenticación frente al servidor.

Si bien los problemas que planteaba el SMTP básico se han mejorado, el problema del *spam* sigue sin solucionarse, por lo que ya se proponen mecanismos alternativos para definir una infraestructura de correo, como Internet Mail 2000 (http://www.im2000.org./).

1.5.3 IMAP (*Internet Message Access Protocol*)

El protocolo IMAP (Protocolo de Internet para el Acceso a Mensajes Electrónicos), actualmente en su versión 4 primera revisión, es un protocolo que al igual que POP se utiliza para que un usuario pueda acceder a sus mensajes de correo electrónico contenidos en un servidor. IMAP está definido en el RFC 2060 , aunque desde 2003 ha sido actualizado por el RFC 3501 (http://www.faqs.org/rfcs/rfc3501.html).

Algunas de las características principales de IMAP son las siguientes:

- Una de las ventajas más destacadas es la capacidad de un usuario de poder manejar varios buzones, llamados carpetas remotas de mensajes en el RFC, pudiendo crear, borrar y renombrar estos buzones. En POP3, como se comentó anteriormente, el usuario sólo tiene acceso a un único buzón.
- Se permite que varios usuarios accedan simultáneamente a un mismo buzón, a diferencia de POP3 donde cada buzón pertenece a un único usuario.
- Se pueden activar y desactivar banderines (*flags*) para controlar el estado de un mensaje, es decir, saber si ha sido borrado, leído, no leído, etc. POP3 no puede manejar banderines. Como nota curiosa decir que la palabra *flag* proviene del banderín de los antiguos taxis, donde una palanca en forma de banderita indicaba si el taxímetro estaba en funcionamiento o no. También se utiliza un banderín en los buzones de algunos países para que el cartero indique si ha depositado correo o no.
- Se pueden leer los mensajes desde el servidor sin necesidad de descargarlos, como ocurre con POP3.
- Se pueden realizar búsquedas en el servidor, con el objetivo de localizar los mensajes que cumplan ciertas características.

---

[2] *Transport Layer Security*: versión estandarizada por el IETF (*Internet Engineering Task Force*) del protocolo SSL (*Secure Sockets Layer*), que consiste en un protocolo criptográfico que permite cifrar la conexión y garantizar la autenticación.

Por todas estas características se observa que IMAP4 es más potente que POP3, si bien aún se manejan los dos protocolos.

1.5.4 MIME (*Multipurpose Internet Mail Extension*)

Las extensiones MIME son una ampliación del RFC 822, definidas en los RFC 2045 a 2049 (http://www.rfc-es.org/rfc/rfc2045-es.txt, http://www.rfc-es.org/rfc/rfc2046-es.txt, http://www.rfc-es.org/rfc/rfc2047-es.txt), pensadas para superar algunos de los problemas impuestos por las restricciones de SMTP y del formato de mensaje del RFC 822. Algunos de estos problemas son los siguientes:

- El contenido del mensaje sólo puede ser texto por lo que no se pueden transmitir ficheros ejecutables, binarios o de cualquier otro tipo.
- El conjunto de caracteres aceptado está limitado al US-ASCII, de 7 bits, por lo que no se admiten caracteres de otros idiomas, que necesitan de 8 bits para su representación.
- El tamaño total de los mensajes no puede superar un determinado tamaño.
- Problemas en la conversión entre distintos formatos.

MIME describe una serie de mecanismos que, cooperando entre sí, solucionan la mayoría de estos problemas sin añadir incompatibilidades a los correos basados en RFC 822. Estos mecanismos se basan en:

- Definición de cinco nuevos campos en la cabecera del mensaje:
  - Versión MIME: este campo debe contener al menos la versión 1.0
  - Tipo de contenido: indica el tipo y subtipo de los datos contenidos en el cuerpo del mensaje.
  - Codificación para la transferencia del contenido: tipo de codificación que se ha usado con el cuerpo del mensaje.
  - Identificador de Contenido y Descripción de contenido: amplían la descripción de la información contenida en el mensaje. Por ejemplo para añadir información sobre una imagen o un fichero de voz.
- Se definen varios tipos de contenido y dentro de estos tipos varios subtipos. A continuación se detallan algunos de los más importantes; el tipo principal se separa del subtipo mediante una barra inclinada, por ejemplo **application/msword**:
  - *application*:
    - *msword*: ficheros propios de Microsoft Office Word.
    - *pdf*: formato de documento comprimido de Adobe Acrobat.
    - *zip*: ficheros comprimidos con el formato zip.
  - *audio*:
    - *ac3*: estándar de compresión de audio.

- *mpeg4-generic*: estándar de compresión de datos de audio y video.

- *image*:
  - *jpeg*: formato de imagen JPG.
  - *gif*: formato de imagen GIF.
- *message*:
  - *rfc822*: se trata de un mensaje encapsulado que cumple con el formato RFC 822.
  - *delivery-status*: este tipo se puede usar para informar al emisor si el mensaje ha sido entregado con éxito.
  - *partial*: usado para permitir la fragmentación de correos grandes.
- *text*:
  - *plain*: texto plano sin formato.
  - *html*: texto en formato html.
  - *rtf*: formato enriquecido para el intercambio de información entre programas de edición de texto.
  - *enriched*: se trata de un formato de texto más rico que el plano pero más pobre que el *html* o *rtf*.
- *multipart*: este formato es muy interesante pues permite integrar en el cuerpo del mensaje múltiples partes independientes. Las más destacadas son:
  - *mixed*: se usacuando las partes del cuerpo son independientes y necesitan ser agrupadas en un orden determinado para ser entregadas al receptor.
  - *alternative*: cada una de las partes del cuerpo son versiones alternativas de la misma información, de forma que el receptor puede tomar la que mejor le convenga.
  - *digest*: convierte las partes de cuerpo de tipo de contenido *text/plain* en tipo de contenido *message/rfc822*.
  - *parallel*: es equivalente al *multipart/mixed* pero sin definir ningún orden determinado.
  - *related*: las partes del cuerpo del mensaje están relacionadas entre sí, como por ejemplo el texto de una página web y sus imágenes, si éstas están contenidas en el propio mensaje.

- Se definen nuevos mecanismos estándares de codificación de datos para evitar problemas de conversión entre distintos sistemas de correo. Por ejemplo muchos tipos de datos están en formato de 8 bits por lo que no se pueden transmitir tal cual por un sistema SMTP basado en RFC 821 de 7 bits. MIME define un mecanismo estándar para codificar estos datos en el formato de líneas cortas de 7 bits (líneas no más largas de 1000 caracteres incluyendo los separadores de líneas CR-LF al final).

Capítulo 2

Primeros pasos

2.1 Introducción a Java y JavaMail

En este epígrafe se comentarán algunas de las principales características del lenguaje de programación Java y sus paquetes principales. Se hablará también del objeto principal de nuestro estudio, la API JavaMail, explicando qué es, para qué se usa y cuáles son sus cualidades más importantes.

2.1.1 El lenguaje de programación Java

Java es una plataforma de *software* desarrollada por Sun Microsystems, de tal manera que los programas creados en ella puedan ejecutarse sin cambios en diferentes tipos de arquitecturas y dispositivos computacionales.

La plataforma Java consta de las siguientes partes:

- El lenguaje de programación en sí, totalmente orientado a objetos.
- La máquina virtual de Java junto con el entorno de ejecución, también llamado JRE (Java Runtime Environment), que debe existir en cada plataforma en la que se quieran ejecutar programas Java.
- Las API de Java, un conjunto de bibliotecas estándares para el lenguaje que lo dotan de múltiples capacidades: acceso a base de datos, entrada/salida de ficheros, conexiones a redes, construcción de ventanas y sus componentes (botones, cuadros de texto, listas, etc.), gestión de eventos, etc.

Originalmente llamado OAK por los ingenieros de Sun Microsystems, Java fue diseñado para ser ejecutado en ordenadores empotrados. Sin embargo, en 1995, dada la importancia que estaba adquiriendo la web, Sun Microsystems la distribuyó para sistemas operativos tales como Microsoft Windows.

El lenguaje en sí se inspira en la sintaxis de C++, pero su funcionamiento es más similar al de Smalltalk que a éste. Incorpora sincronización y gestión multitarea mediante hilos (*threads*) y utiliza el concepto de *interface* como mecanismo alternativo a la herencia múltiple de C++ y otros lenguajes orientados a objetos.

Java se ha convertido en uno de los lenguajes de mayor acogida para construir programas de servidor. Utilizando una tecnología llamada JSP-*JavaServer Pages* (similar a otras tecnologías del lado del servidor como ASP de Microsoft o PHP), se hace muy fácil escribir páginas dinámicas para sitios de Internet. Por otro

lado, la tecnología de *JavaBeans*, cuando se utiliza junto con JSP, permite adaptar al mundo web el patrón MVC (modelo-vista-controlador) que ya se había aplicado con éxito a interfaces gráficas. La máxima expresión de este patrón aplicado al mundo web ha venido de la mano de *JavaServer Faces*, tecnología que aporta un entorno de trabajo que similar al de la API Swing pero orientado a páginas JSP.

Java llegó a ser aún más popular cuando Sun Microsystems introdujo la especificación J2EE (*Java 2 Enterprise Edition*). Este modelo permite, entre otros, una separación entre la presentación de los datos al usuario (JSP o *applets*), el modelo de datos (EJB-*Enterprise Java Beans*), y el control (*servlets*). EJB es una tecnología de objetos distribuidos que ha logrado el sueño de muchas empresas como Microsoft e IBM de crear una plataforma de objetos distribuidos con un monitor de transacciones. Con este nuevo estándar, empresas como BEA, IBM, Sun Microsystems, Oracle y otros, han creado servidores de aplicaciones que han tenido gran acogida en el mercado.

Además de programas del servidor, Java permite escribir programas de cliente como cualquier otro lenguaje de programación, pudiéndose crear aplicaciones con una interfaz gráfica o textual. También se pueden ejecutar programas Java como parte de una página web en forma de *applets* de manera que son interpretados porlos navegadores web. No obstante, esto no ha llegado a popularizarse tanto como se esperaba en un principio.

Los programas fuente en Java son compilados a un lenguaje intermedio formado por *bytecodes* (código máquina de un microprocesador virtual almacenado en un fichero de extensión **.class**), que luego son interpretados por una máquina virtual (JVM - *Java Virtual Machine*). Esta última sirve como una plataforma de abstracción entre el ordenador y el lenguaje permitiendo que se pueda «escribir el programa una vez, y ejecutarlo en cualquier lado». En la figura 2.1 puede apreciarse este proceso.

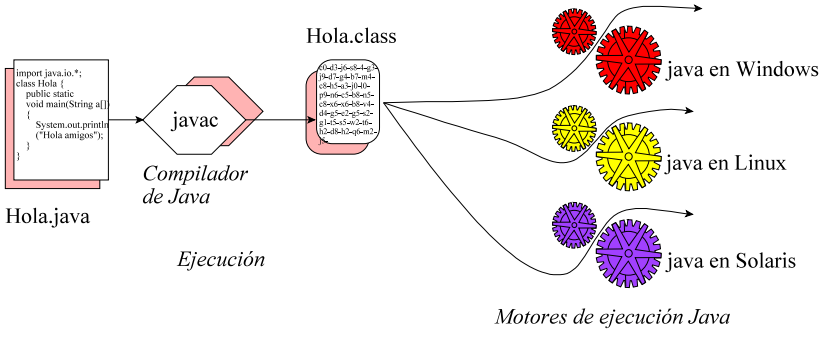

Figura 2.1 Filosofía «escribe una vez, ejecuta en cualquier lado».

Algunos de los paquetes más importantes incluidos en Java son los siguientes (un paquete no es más que un directorio que contiene un conjunto de clases Java):

- **java.lang**: contiene las clases más importantes del lenguaje como la clase **Object** (de la que heredan todas las demás), **String** y **StringBuffer** (manejo de cadenas), **System** (permite usar recursos del sistema), etc.
- **java.io**: proporciona los recursos para poder realizar operaciones de entrada/salida. Contiene clases como **File** (representación abstracta de ficheros y directorios), **FileReader** y **FileWriter** (lectura y escritura de ficheros), etc.
- **java.util**: conjunto de clases de utilidades como por ejemplo estructuras de datos (**Vector**, **Hashtable**, etc.), **Date** (manejo de fechas), **GregorianCalendar** (gestión avanzada de fechas), la interfaz **Observable** (permite a los objetos notificarse entre sí cuando han sufrido alguna modificación), etc.
- **java.net**: proporciona soporte para realizar distintas operaciones de red. Incluye clases para dar soporte a TCP/IP, manejar **Sockets**, recursos **URL**, etc.
- **java.applet**: provee las clases necesarias para poder crear y usar *applets*.
- **java.awt**: proporciona los mecanismos necesarios para crear interfaces de usuario y para dibujar gráficos e imágenes. Se suele usar junto con el paquete **javax.swing**.

2.1.2 La API JavaMail

JavaMail es una API opcional a la versión estándar de Java (J2SDK) que proporciona funcionalidades de correo electrónico, a través del paquete **javax.mail**. Permite realizar desde tareas básicas como enviar, leer y componer mensajes, hasta tareas más sofisticadas como manejar gran variedad de formatos de mensajes y datos, y definir protocolos de acceso y transporte. Aunque a primera vista pueda parecer que su utilidad se orienta a construir clientes de correo-e de tipo Outlook, ThunderBird, etc., su aplicación se puede generalizar a cualquier programa Java que necesite enviar y/o recibir mensajes, como por ejemplo, aplicaciones de *intranets*, páginas JSP, etc.

JavaMail soporta, por defecto, los protocolos de envío y recepción SMTP, IMAP, POP3 y las versiones seguras de estos protocolos SMTPS, IMAPS, POP3S (estos 3 últimos a partir de la versión JDK 1.3.2), si bien puede implementar otros protocolos. El envío y recepción son independientes del protocolo, aunque podremos necesitar usar un protocolo u otro según nuestras necesidades.

Este paquete va unido al uso del paquete *JavaBeans Activation Framework* (JAF), que es también un paquete opcional a la versión estándar de Java (J2SDK), aunque tanto éste como JavaMail vienen incorporados en la versión empresarial (J2EE) . Este paquete es el que se encarga de la manipulación de los canales (*streams*) de bytes de los tipos MIME, que pueden incluirse en un mensaje.

2.2 Posibilidades de JavaMail

JavaMail está diseñada para facilitar las funcionalidades más comunes, si bien también posee características más avanzadas, que permiten sacar el máximo partido a los estándares de correo electrónico. Las funciones más importantes se detallan a continuación:

- Componer y enviar un mensaje: el primer paso que ha de realizar cualquier aplicación que pretenda enviar un correo electrónico es componer el mensaje. Es posible crear un mensaje de texto plano, es decir, un mensaje sin adjuntos que contenga exclusivamente texto formado por caracteres ASCII; pero es igualmente sencillo componer mensajes más completos que contengan adjuntos. También se pueden componer mensajes que contengan código HTML e incluso imágenes embebidas en el texto.
- Descargar Mensajes: se pueden descargar los mensajes desde la carpeta que se indique ubicada en el servidor que en ese momento se esté usando. Estos mensajes pueden ser de texto plano, contener adjuntos, HTML, etc.
- Usar *flags* (banderines): para indicar, por ejemplo, que un mensaje ha sido leído, borrado, etc., en función de cuáles de estos *flags* estén soportados por el servidor.
- Establecer una conexión segura: actualmente algunos servidores de correo requieren de una conexión segura, ya sea para descargar el correo almacenado en ellos, o para enviar mensajes a través de ellos. JavaMail ofrece la posibilidad de establecer este tipo de conexiones, indicando que se trata de una conexión segura, además de poder indicar otros parámetros, en algunos casos necesarios, como el puerto donde establecer la conexión. Esta capacidad está disponible desde la versión de JDK 1.3.2
- Autenticarse en un servidor: ciertos servidores requieren autenticación ya no sólo para leer sino también para enviar. JavaMail ofrece también la posibilidad de autenticación a la hora de enviar un correo.
- Definir protocolos: JavaMail soporta por defecto los protocolos de envío y recepción SMTP, IMAP, POP3, (y sus correspondiente seguros a partir de la versión de JDK 1.3.2), si bien puede implementar otros protocolos. Para

usar otros protocolos, o desarrollar por nuestra cuenta soporte para algún protocolo no definido por defecto en JavaMail, podemos es posible consultar la página de Sun, http://java.sun.com/products/javamail/Third_Party.html, donde se detallan una serie de protocolos que podemos usar, y que no se ofrecen por defecto con JavaMail ya que no son estándares.

- Manejar adjuntos: JavaMail ofrece la posibilidad de añadir adjuntos a los mensajes de salida, así como obtener y manipular los adjuntos contenidos en un mensaje de entrada.
- Búsquedas: JavaMail ofrece la posibilidad de buscar mensajes dentro de la cuenta de correo en el propio servidor -sin necesidad de descargar todo el correo- que concuerden con un criterio de búsqueda previamente definido. Para ello dispone de varias clases que permiten buscar mensajes con un *subject* determinado, un *from* concreto, etc., devolviendo un *array* con los mensajes que cumplan estos criterios. En el capítulo 7 veremos esto con más detenimiento.
- Acuse de recibo y prioridad: Se puede incluir un acuse de recibo en los mensajes de salida, para que el remitente sepa si un mensaje ha sido leído o no por el destinatario, si bien no todos los servidores soportan esta funcionalidad. Además se puede establecer la prioridad de un mensaje de forma muy sencilla.

2.3 Relación con JMS (*Java Message Service*)

JMS es una API de Java que permite a las aplicaciones crear, enviar, recibir y leer mensajes. La filosofía de JMS está orientada al paso de mensajes entre aplicaciones y no al intercambio de mensajes de correo electrónico, como ocurre con JavaMail.

JMS se sitúa entre dos aplicaciones que mantienen una comunicación. La aplicación **A** puede enviar un mensaje al sistema de mensajes que constituye JMS. Este a su vez se lo enviará a **B**, cuando la aplicación **B** se conecte al sistema. Esto permite una comunicación asíncrona entre las aplicaciones.

JMS permite además que una aplicación envíe algo al sistema de mensajes y lo reciban todas las aplicaciones conectadas a ese servicio. Este tipo de comunicación se denomina *Publisher/Subscriber* (Editor/Subscriptor).

2.4 Instalación de JavaMail

La instalación de JavaMail es muy sencilla y se realiza en pocos pasos. Por

supuesto, antes de proceder a la instalación hay que comprobar que tenemos instalada la versión de JDK 1.1.x o una posterior.

En el caso de tener instalada una versión de Java J2EE igual a la 1.3 o posterior no tendremos que realizar ninguna instalación de JavaMail, pues ésta ya forma parte de las API incorporadas por J2EE.

Los pasos para instalar JavaMail son los siguientes:

- Descargar la API de JavaMail. Para ello se accede a la dirección http://java.sun.com/products/javamail y se descarga la versión que creamos más oportuna. Se recomienda la versión 1.4, ya que incorpora algunas mejoras sobre el tratamiento de los componentes MIME. De esta manera se obtiene un fichero .**zip** llamado **javamail-version**, donde **version** se corresponde con la versión de JavaMail que descargada.

- Descomprimir el fichero y copiar el fichero **mail.jar** en el directorio de extensiones de Java. Suponiendo que Java está instalado en una máquina con sistema operativo Microsoft Windows XP en el directorio **C:\j2sdk1.4.2_09**, el fichero **mail.jar** se debe copiar en el subdirectorio **C:\j2sdk1.4.2_09\jre\lib\ext**.

- Una alternativa a copiar el fichero **mail.jar** en el directorio de extensiones es modificar el **CLASSPATH**, añadiéndole la ruta del directorio en el que se haya copiado el fichero. Por ejemplo, en una máquina con Microsoft Windows XP (ver figura 2.2):
    - Con el botón derecho sobre **Mi Pc** se selecciona **Propiedades**.
    - Escoger la pestaña **Opciones avanzadas** y luego pulsar el botón **Variables de entorno**.
    - Aparecerá una ventana dividida en dos cuadros: variables de usuario y de sistema. Hay que mirar en estos dos cuadros si aparece la variable **Classpath** (puede aparecer en los dos), se la debe seleccionar y pulsar el botón **Modificar**.
    - Aparecerá un cuadro con el nombre de la variable y su valor. En valor de la variable, añadimos la ruta donde se haya copiado el fichero **mail.jar** y se pulsa **Aceptar**. Por ejemplo si **mail.jar** se ha copiado en un directorio llamado **javamail** que cuelga del directorio *home* de java, se debe añadir al final del valor de la variable: ***<valor anterior>*;C:*java_home*\javamail\mail.jarM;.**. Nótese que el punto al final es necesario.

- En una máquina con sistema operativo Unix/Linux, habría que escribir (suponiendo que el contenido del fichero **.zip** que contiene JavaMail se descomprime en **usr/java/javamail**):

```
$CLASSPATH=$CLASSPATH:usr/java/javamail
```

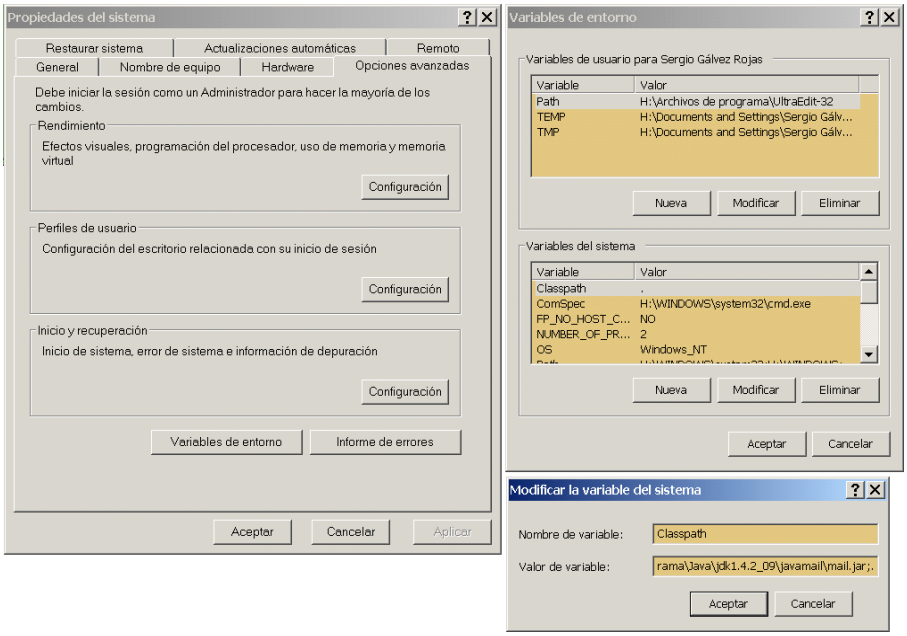

Figura 2.2 Modificación de la variable de entorno CLASSPATH.

```
$export $CLASSPATH
```

- El último paso consiste en instalar la API *JavaBeans Activation FrameWork* (JAF) que, como ya se comentó anteriormente, se usa para manipular ciertos tipos de contenidos que se usan con las extensiones MIME y que JavaMail necesita. Al igual que ocurre con JavaMail, los usuarios de J2EE no necesitan efectuar tampoco la instalación de JAF pues con J2EE ya viene incluida. SUN recomienda el uso de la versión JAF 1.0.2 o posterior.

  La instalación de JAF es igual a la de JavaMail. De la dirección http://java.sun.com/products/javabeans/glasgow/jaf.html hay que descargar el fichero **jaf-version.zip**. El resto de los pasos son iguales pero usando el fichero **activation.jar** en lugar del **mail.jar**.

2.5 Contenido de javamail-1_4.zip

Al descomprimir el fichero que contiene JavaMail (**javamail-1_4.zip** por ejemplo), se observa que contiene una serie de ficheros de texto, un fichero llamado **mail.jar** del cual ya se ha hablado anteriormente, y tres directorios. A continuación se comentan qué contienen y para qué sirven estos directorios y ficheros y, sobre todo, el significado del resto de ficheros **.jar** que incorpora.

2.5.1 Directorios

El directorio **\docs** contiene ficheros de texto correspondientes a cada versión de JavaMail en los que se explica los cambios introducidos en cada una de ellas. Contiene también dos ficheros en formato Adobe Acrobat, llamados **javamail-version.pdf** y **providers.pdf** que son el tutorial oficial de JavaMail desarrollado por Sun y un tutorial sobre cómo desarrollar un proveedor de servicios, es decir, dar soporte a algún protocolo no recogido en la implementación estándar de JavaMail. Por último, contiene un directorio llamado **\docs\javadocs**, en el que está la ayuda sobre la API en formato HTML generado por **javadoc**. Este es el formato en el que se suele consultar la información sobre las APIs que componen Java, aunque resulta tan cómodo de utilizar que, últimamente, su uso se está extendiendo a otros lenguajes de programación.

El directorio **\demo** contiene una serie de ejemplos en los que se usa JavaMail y que nos pueden servir para hacernos una idea de sus posibilidades. En el fichero **readme.txt** es posible consultar una lista en la que, de forma muy somera, se comenta que hace cada uno de estos ejemplos.

2.5.2 Ficheros .jar

Por último, es fundamental comentar los ficheros **.jar** contenidos en el archivo de instalación. Por un lado se suministra un fichero llamado **mail.jar** y del directorio **\lib**, que contiene otros cuatro ficheros **.jar** llamados **imap.jar**, **mailapi.jar**, **pop3.jar** y **smtp.jar**.

- El fichero **mail.jar** contiene la implementación completa de la API JavaMail y todos los proveedores de servicio de Sun, es decir, el soporte para los protocolos IMAP, POP3 y SMTP y sus correspondientes versiones seguras. Por tanto, como ya se comentó en el apartado 2.4, la mayoría de los usuarios sólo necesitan este fichero **.jar** aunque, en ciertas ocasiones, puede ser necesario el uso de alguno de los otros ficheros **.jar**.
- El fichero **mailapi.jar** contiene también la implementación completa de la API de JavaMail, pero sin incluir los proveedores de servicio (nótese que el tamaño de este fichero es aproximadamente la mitad que el de **mail.jar**). Estos proveedores vienen independientemente empaquetados en los respectivos ficheros **imap.jar**, **pop3.jar** y **smtp.jar**.

En los casos en que se quiera que nuestra aplicación tenga el menor tamaño posible, se puede usar **mailapi.jar** y añadir sólo el soporte para el protocolo o los protocolos que necesitemos de forma separada, incluyendo el correspondiente **.jar**. La instalación es la misma que la descrita en el apartado 2.4, sólo que sustituyendo el

fichero **mail.jar** por **mailapi.jar** y repitiendo el proceso con el resto de ficheros **.jar** necesarios en función del protocolo a usar. Un ejemplo de esto puede ser que queramos realizar un pequeño *applet* cuya única función es enviar mensajes. En tal caso es recomendable que la aplicación ocupe poco espacio. Además no necesita manejar los protocolos POP3 ni IMAP, por tanto lo ideal sería que esa aplicación usara sólo **mailapi.jar** y **smtp.jar**.

La utilización de los ficheros **.jar** por separado en lugar de **mail.jar** entraña las siguientes consideraciones:

- Para usar más de un fichero **.jar** de los relacionados con los protocolos, hay que tener instalada una versión de Java igual o superior a J2SE 1.2. Con el **mail.jar** no sucede esto, pudiéndose usar incluso con la versión de J2SE 1.1.
- No es posible mezclar los **.jar** que dan soporte a los protocolos si éstos pertenecen a distintas versiones de JavaMail.

Capítulo 3

Enviar y recibir mensajes básicos

3.1 Visión general

En este capítulo procederemos a exponer dos ejemplos sencillos de cómo enviar un correo y cómo recibirlo. En ambos casos los mensajes serán sólo de texto plano, con contenido, asunto, fecha, etc. pero sin adjuntos ni otras florituras.

En ambos casos se parte de cuentas de correo predeterminadas, es decir, la dirección del servidor SMTP que usaremos para enviar el mensaje y la del servidor POP3 que usaremos para recibirlo vendrán codificadas directamente en el código. En el caso de POP3 la cuenta debe existir previamente en el servidor.

3.2 Envío de un mensaje de correo electrónico

Una vez instalado JavaMail, en este apartado vamos a estudiar cómo enviarun mensaje sencillo de correo electrónico: texto plano (sin HTML ni caracteres en negrita, subrayado, etc.) y sin adjuntos.

El proceso básico para enviar un correo consiste en:

- Crear una sesión a la que le pasan las propiedades de la conexión, entre las que se ha debido incluir el servidor SMTP que se quiere usar.
- A continuación se crea un mensaje y se rellenan sus campos: asunto, texto del mensaje, emisor y destinatario.
- Por último se envía el mensaje.

A continuación se desgranarán y explicarán estos pasos, con sus correspondientes líneas de código y sus posibles alternativas.

3.2.1 Establecimiento de propiedades de la conexión

Vamos a crear la clase EnviarCorreo, y lo primero que se debe hacer es importar las clases del paquete javax.mail y del paquete javax.mail.internet. El primero de ellos contiene clases que modelan un sistema de correo genérico, es decir, características que son comunes a cualquier sistema de correo, como pueden ser un mensaje o una dirección de correo. javax.mail.internet contiene clases específicas para modelar un sistema de correo a través de Internet, como por ejemplo clases para

representar direcciones de Internet o usar características de las extensiones MIME. También será necesario usar la clase java.util.Properties, que representa un conjunto de propiedades, y que permitirá establecer el servidor SMTP como una propiedad de la conexión. Por tanto la aplicación comenzará importando estos elementos.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
```

Al ser éste un ejemplo sencillo, el método main incluirá todo el código como un bloqueindivisible. El main tendrá dos argumentos: el *from* (la dirección del remitente) y el *to* (la dirección del destinatario).

```
public class EnviarCorreo {
    public static void main (String [] args) {
        if (args.length != 2) {
            System.out.println("Ha de enviar dos parámetros \n" +
                    "java EnviarCorreo fromAddress toAddress");
            System.exit(1);
        }
        String from = args [0];
        String to   = args [1];
```

Ahora comienza lo interesante. El primer paso consiste en indicar el servidor SMTP que se va a usar para enviar el correo. Esto se hace añadiendo una nueva propiedad a las propiedades del sistema, que indicará que se quiere usar un servidor SMTP e identificará su correspondiente valor, es decir su dirección. En este ejemplo se usa un servidor predeterminado; para usar cualquier otro servidor SMTP sólo hay que consultar la dirección de algún servidor de correo como YAHOO!, GMAIL, Hotmail, o la de nuestro lugar de estudio o trabajo y ver la explicación que dan acerca de cómo configurar un programa de correo electrónico para usar sus servicios.

```
String smtpHost = "smtp.auna.com";
Properties props = System.getProperties();
props.put("mail.smtp.host",smtpHost);
```

Para comprobar si la dirección de un servidor es correcta se puede usar, en Windows XP, el comando **ping** o el comando **tracert** seguidos de la dirección del servidor. En Linux también se puede usar el comando **ping** o el comando **traceroute** de la misma forma.

La clase Properties contiene un conjunto de pares (clave, dato) que serán ambos de tipo String, y mediante un nuevo par es como se añade el servidor SMTP que se pretende usar a las propiedades. Con el método put es posible añadir otras propiedades referidas al correo electrónico, como el servidor para cada protocolo, el puerto de conexión, el nombre de usuario, etc. Para consultar estas propiedades se puede visitar la página http://java.sun.com/products/javamail/javadocs/index.html. En la tabla 3.1 podemos ver algunas de estas propiedades.

Nombre	Descripción
mail.debug	Hace que el programa se ejecute en modo depurador, es decir, muestra por pantalla todas las operaciones que realiza y los posibles errores. Por defecto no está activada. P.ej. props.put ("mail.debug", "true")
mail.from	Añade la dirección del remitente en el mensaje. P.ej. props.put ("mail.from", "ignacio@gmail.com")
mail.*protocolo*.host	Dirección del servidor para el protocolo especificado, es decir, el que se va a usar para enviar o recibir. P.ej. props.put ("mail.smtp.host", "smtp.ono.com")
mail.*protocolo*.port	Número de puerto del servidor que hay que usar. P.ej. props.put ("mail.pop3.port", "995")
mail.*protocolo*.auth	Indica que vamos a autenticarnos en el servidor. P.ej. props.put("mail.smtp.auth", "true")
mail.*protocolo*.user	Nombre de usuario con que conectarse al servidor. P.ej. props.put ("mail.pop3.user", "pruebasmail2005")

Tabla 3.1 Propiedadesmás comunes en una conexión de correo-e

Hay que indicar que para enviar el correo a través de algunos servidores no es suficiente con indicar sólo su nombre sino que, además, hay que autenticarse y/o utilizar una conexión segura. Esto se explica con detalle en el capítulo 6.

A continuación es necesario crear una sesión de correo usando para ello un objeto de la clase Session; para ello hay que hacer uso del objeto de tipo Properties creado en el paso anterior. Un objeto de la clase Session representa una sesión de correo y reúne sus principales características, de manera que cualquier operación posterior de envío/recepción de mensajes se hace en el contexto de una sesión. Dichos objetos se crean haciendo uso de dos métodos estáticos que devuelven un objeto de este tipo.

El primero de ellos devuelve la sesión por defecto, siempre la misma cuantas veces sea invocado este método. Este método recibe como parámetro el objeto de tipo Properties creado anteriormente y un objeto de la clase Authenticator. En el caso que nos ocupa, como segundo parámetro enviaremos null, y en el capítulo 7 se verá cómo usar convenientemente el objeto de tipo Authenticator:

```
Session sesion = Session.getDefaultInstance(props,null);
```

El segundo método es similar al primero con la diferencia de que las aplicaciones no comparten esta sesión, pues cada vez que se invoca retorna una sesión diferente.

```
Session sesion = Session.getInstance(props,null);
```

3.2.2 Creación del mensaje

Una vez especificado el servidor SMTP y creada la sesión de correo-e, ya se puede proceder a la creación del mensaje que se quiere enviar. Para modelar un mensaje de correo electrónico existe la clase Message del paquete javax.mail, pero al ser esta clase abstracta es necesario crear una instancia de la única clase estándar que la implementa, la clase MimeMessage. Esta clase representa un mensaje de correo electrónico de tipo MIME. Posee seis constructores, pero en este caso vamos a usar uno que nos devolverá un mensaje vacío que iremos rellenando paso a paso con

Constructor	Comportamiento
public MimeMessage(Session session)	Es el constructor que se suele usar y devuelve un mensaje vacío.
public MimeMessage (MimeMessage source)	Crea un mensaje a partir de otro mensaje que se le pasa como parámetro. Es una copia del otro mensaje.
public MimeMessage(Session session, InputStream in)	Construye un mensaje a partir de los datos devueltos por **in** que serán el contenido y las cabeceras del mensaje.
protected MimeMessage (Folder folder, InputStream in, int msgnum)	Crea el mensaje leyendo los datos del canal de entrada **in** contenido en la carpeta **folder**, y que ocupa la posición **msgnum** dentro de ésta.
protected MimeMessage(Folder folder, int msgnum)	Crea un mensaje a partir del mensaje que ocupa la posición **msgnum** dentro de la carpeta **folder**.
protected MimeMessage(Folder folder, InternetHeaders headers, byte[] content, int msgnum)	Crea un mensaje a partir del mensaje que ocupa la posición **msgnum** dentro de la carpeta **folder**, leyendo las cabeceras desde **headers** y el contenido desde **content**.

Tabla 3.2 Constructores de la clase MimeMessage. Nótese que los tres últimos constructores son de tipo protected y sólo pueden ser utilizados directamente por clase hijas de MimeMessage.

posterioridad. En la tabla 3.2 pueden verse estos constructores. Así pues, en este ejemplo la creación del mensaje queda como:

```
Message mensaje = new MimeMessage(sesion);
```

A continuación hay que proceder a rellenar los atributos (asunto, fecha de envío, remitente, etc.) y el contenido del mensaje. Existe una gran variedad de métodos para rellenar y obtener tanto estos atributos como el contenido en sí. Para consultarlos todos podemos ver la correspondiente documentación de la API de JavaMail; no obstante en la tabla 3.3 se detallan algunos de los más importantes. Debe tenerse en cuenta que la clase MimeMessage no sólo hereda de Message sino que también implementa las interfaces Part y MimePart, por lo que puede usarse cualquier método de estas clases/interfaces.

Así, lo más fácil es rellenar el asunto del mensaje usando el método setSubject qué recibe como parámetro la cadena que constituirá el asunto:

```
mensaje.setSubject("Hola Mundo");
```

Nótese que, dado que estos métodos pueden elevar excepciones (principalmente del tipo MessagingException), hay que colocar el código encerrado en el correspondiente **try-catch**.

El campo *from* hay que rellenarlo para que el receptor del mensaje sepa de

Método	Comportamiento
void setFrom(Address dirección)	Establece como dirección del remitente la que se pasa como parámetro.
void setSubject(String asunto)	Establece como asunto el String quese pasa como parámetro.
void setContent(Object obj, String type)	Establece el contenido del mensaje del tipo que le indicamos. Puede ser *"text/plain"* si el mensaje se rellena con un String sencillo, "*text/html*" si se hace con texto en formato HTML, o "*multipart/alternative*", "*multipart/related*", "*multipart/mixed*", si el mensaje se rellena con un objeto Multipart.
void setContent (Multipart mp)	Rellena el mensaje con el Multipart que se le pasa y cuyo tipo MIME habrá sido establecido al crearlo.
void setText(String texto)	Es equivalente a usar el método setContent insertando un String e indicando que se trata de texto plano ("*text/plain*").

Tabla 3.3 Métodos para establecer atributos en un mensaje. Todos pueden elevar la excepción MessagingException. Los métodos de color azul son de la clase Message y los de color verde de la interfaz Part..

quién proviene: método setFrom, que requiere como parámetro un objeto de tipo Address que modela una dirección cualquiera en el sentido más general. Como esta clase es abstracta hay que usar en su lugar alguna clase que la implemente, en nuestro caso InternetAddress. Esta clase representa una dirección de Internet que cumpla con la sintaxis especificada en el estándar RFC 822, y que suele ser de la forma *usuario@máquina.dominio* o bien simplemente *usuario* si, por ejemplo tanto el remitente como el destinatariuo poseen sus cuentas de correo en la misma máquina. La clase InternetAddress posee diversos constructores, de los que usaremos uno que sólo requiere una dirección pasada en forma de String. Nótese que aquí se podría haber incluido cualquier dirección y no necesariamente nuestra dirección real de correo, motivo por el cual resulta perfectamente posible enviar correo de forma anónima e incluso aparentar ser otra persona. Así pues se tiene:

```
mensaje.setFrom(new InternetAddress(from));
```

A continuación se debe rellenar el atributo correspondiente al receptor del mensaje. Para ello se utiliza el método addRecipient de la clase Message al que se le pasan dos parámetros. El primero es un objeto del tipo Message.RecipientType (tipo de destinatario), que es una clase interna a la clase Message, que tiene tres constantes: TO, CC (*Carbon Copy*) y BCC (*Blind Carbon Copy*), que son los tres tipos de recipientes o destinatarios. Con esto se especifica si la dirección del destinatario se añade en el TO, el CC o el BCC del mensaje. El segundo parámetro es la dirección que se añadirá al recipiente y es del tipo InternetAddress, al igual que en el método setFrom. En este ejemplo sólo habrá un destinatario:

```
mensaje.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
```

Existe otro método que permite añadir varias direcciones de una sola tacada en vez de hacerlo de una en una. Para ello se utiliza un *array* de direcciones en lugar de una sola, y el método a emplear es addRecipients, en plural. Por ejemplo:

```
Address [] direcciones = new Address []{
        new InternetAddress ("juan15@gmail.com"),
        new InternetAddress ("migaro@hotmal.com"),
};
mensaje.addRecipients(Message.RecipientType.TO, direcciones);
```

Este apartado finaliza rellenando ahora el contenido del mensaje, para lo que se usa el método setText de la clase Message tal y como se vio en la tabla 3.3. Este método además de establecer como contenido del mensaje el que se le especifique a través de un String, establece que el tipo de contenido es texto plano (*text/plain*).

```
mensaje.setText("Este es el cuerpo del mensaje");
```

3.2.3 Envío del mensaje

Con todo esto, el mensaje ya está listo para ser enviado, lo que constituye el último paso de nuestro programa. Para enviarlo se usa la clase Transport que se

encarga de esto de una forma muy sencilla mediante sus métodos estáticos. En este ejemplo se usará el método send:

```
Transport.send(mensaje);
```

Esta clase tiene sobrecargado el método send, de forma que también permite enviar un mensaje a varios destinatarios, mandándole a este método un *array* de direcciones. De esta forma se ignoran las direcciones que pudiesen estar definidas en el mensaje y se envía sólo y exclusivamente a las especificadas en el *array*:

```
Transport.send(String mensaje,Address [] direcciones);
```

Por último, y como ya se ha indicado anteriormente, los métodos con los que se crea el mensaje, con los que se rellena y el método send que lo envía pueden lanzar excepciones por lo que es necesario capturarlas o relanzarlas. En el caso que nos ocupa todas estas instrucciones se incluyen dentro de un bloque **try-catch** que captur la excepción javax.mail.MessagingException, que es la clase base para manejar las excepciones provocadas por los mensajes:

```
try {
        Message mensaje = new MimeMessage(sesion);
        mensaje.setSubject("Hola Mundo");
        mensaje.setFrom(new InternetAddress(from));
        mensaje.addRecipient(     Message.RecipientType.TO,
                                  new InternetAddress(to));
        mensaje.setText("Este es el cuerpo del mensaje");
        Transport.send(mensaje);
} catch (MessagingException e) {
        System.out.println(e.getMessage());
}
```

3.2.4 Ejemplo completo

A continuación se muestra el listado completo de cómo enviar un mensaje de correo electrónico, fusionando todos los trozos que se han ido estudiando.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class EnviarCorreo {
   public static void main (String [] args) {
      // Se comprueba que el número de argumentos es el correcto
      if (args.length != 2) {
         System.out.println(   "Ha de enviar dos parámetros \n" +
                               "java EnviarCorreo fromAddress toAddress");
         System.exit(1);
      }
      // Se obtienen el from y el to recibidos como parámetros
```

```
        String from = args [0];
        String to   = args [1];
        // Se obtienen las propiedades del sistema y se establece el servidor SMTP
        String smtpHost  = "smtp.auna.com";
        Properties props = System.getProperties();
        props.put("mail.smtp.host",smtpHost);
        // Se obtiene una sesión con las propiedades anteriormente definidas
        Session sesion = Session.getDefaultInstance(props,null);
        // Capturar las excepciones
        try {
          // Se crea un mensaje vacío
          Message mensaje = new MimeMessage(sesion);
          // Se rellenan los atributos y el contenido
          // Asunto
          mensaje.setSubject("Hola Mundo");
          // Emisor del mensaje
           mensaje.setFrom(new InternetAddress(from));
           // Receptor del mensaje
           mensaje.addRecipient(    Message.RecipientType.TO,
                                    new InternetAddress(to));
           // Cuerpo del mensaje
           mensaje.setText("Este es el cuerpo del mensaje");
           // Se envía el mensaje
           Transport.send(mensaje);
        } catch (MessagingException e) {
           System.err.println(e.getMessage());
        }
    }
}
```

3.3 Recepción de mensajes de correo electrónico

En este epígrafe se mostrará cómo recuperar mensajes de correo electrónico. Los pasos básicos para recuperar los mensajes del servidor de correo son:

- Crear una sesión.
- Crear un almacén (Store) a partir de esta sesión y conectarse a él.
- Abrir una carpeta del almacén y recuperar los mensajes.
- Procesar estos mensajes convenientemente.

En este ejemplo se supone que los mensajes no tienen adjuntos.

Los dos protocolos más usados para recuperar mensajes son POP3 e IMAP, siendo POP3 el más usado, y el que usaremos en este ejemplo aunque, no obstante, se explicará también como hacerlo con IMAP en el apartado 3.3.2. El hecho de usar POP3 acarrea una serie de limitaciones que pueden conducir a errores: muchas de las capacidades de JavaMail no están soportadas por el protocolo POP3 y si se intenta

usarlas se lanzará la excepción MethodNotSupportedException. En especial hay que mencionar las limitaciones con respecto al manejo de carpetas y de banderines, ya que POP3 sólo admite una única carpeta llamada **INBOX** y sólo permite trabajar con el banderín que permite borrar los mensajes del servidor. Por último hay que indicar que POP3 no proporciona un mecanismo para obtener fecha y hora en que un mensaje es recibido, por lo que el método getReceivedDate devolverá null.

3.3.1 Establecimiento de propiedades de la conexión

Esta aplicación usa los mismos paquetes y clases que en el caso del envío, así que el primer paso es importar estos paquetes. Además es necesario importar el paquete de entrada java.io para el posterior manejo de excepciones, cosa que se explica en el apartado 3.3.4:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
```

El método main estará dentro de la clase que envía el correo -llamada ObtenerCorreo- y ahí se añadirá todo el código en un solo bloque. Se pasarán sólo como argumentos el nombre de usuario y la contraseña con los que el usuario tenga abierta una cuenta en el servidor.

```
public class ObtenerCorreo {
    public static void main (String [] args) {
        if (args.length != 2) {
            System.out.println(     "Ha de enviar dos parámetros\n" +
                                    "java ObtenerCorreo usuario clave");
            System.exit(1);
        }
        String usuario = args [0];
        String clave   = args [1];
```

Al igual que en ejemplo anterior, se usa un servidor predeterminado, cuya dirección se guarda en la variable popHost. Para usar cualquier otro servidor POP3 sólo hay que consultar el sitio web de algún servidor de correo como YAHOO!, GMAIL, Hotmail, etc. y ver la explicación que dan acerca de cómo configurar un programa de correo electrónico para usar sus servicios.

A continuación hay que obtener las propiedades del sistema y con ellas crear una sesión de correo. La única diferencia con respecto al ejemplo anterior, es que en este caso no es necesario añadir ninguna propiedad a las ya obtenidas:

```
String popHost = "pop.correo.yahoo.es";
Properties props = System.getProperties();
Session sesion = Session.getDefaultInstance(props,null);
```

3.3.2 Conexión a un almacén de mensajes (*Store*)

A continuación crearemos un objeto de la clase Store apropiado para este protocolo, usando la sesión creada anteriormente, y con el que es posible conectarse al servidor indicado nombre de usuario y contraseña. Conceptualmente un Store representa nuestro buzón de correo dentro del servidor, es decir, nuestra cuenta de correo. La clase Store permite acceder a una estructura de carpetas (clase Folder) y a los mensajes contenidos en cada una de ellas. Define además el protocolo de acceso que se usará en la conexión:

```
Store store = sesion.getStore("pop3");
store.connect(popHost,usuario,clave);
```

Si usamos IMAP en vez de POP3 sólo hemos de cambiar **"pop3"** por **"imap"** y, por supuesto, el nombre del servidor (contenido en la variable popHost) por el del correspondiente servidor de IMAP.

En vez de usar la clase Store y el método connect al que le pasamos dirección del servidor, nombre de usuario y contraseña, esposible usar un objeto de la clase POP3Store, que extiende a la clase Store y permite usar el método connect sin argumentos. Para crear este objeto usamos el constructor de la clase POP3Store que toma como argumentos la sesión creada y un objeto de tipo URLName. A su vez, para crear el objeto URLName se puede usar un constructor al que se le pasa un String de la forma:

"pop3://usuario:clave@servidorPOP:puerto_de_conexión/INBOX"

El puerto de conexión por defecto es el 110, aunque si no funciona habrá que averiguar qué puerto utiliza realmente nuestro servidor POP. Así pues, las dos líneas de código anteriores se sustituirían por:

```
URLName url = new URLName (
                    "pop3://" + usuario + ":" + clave + "@" + popHost +
            ":" + puerto_de_conexion + "/INBOX");
POP3Store store = new POP3Store (sesion, url);
store.connect();
```

debiéndose importar, además, la clase com.sun.mail.pop3.POP3Store si lo hacemos de esta forma.

Es importante destacar que los datos no viajan cifrados, por lo que alguien podría interceptarlos y ver nuestra clave. En el capítulo 6 se verá cómo solucionar esto.

3.3.3 Recuperación de mensajes

Una vez conectados a nuestro Store hay que abrir la carpeta **INBOX** que, como ya se comentó, es la única disponible en POP3. Para ello se debe crear un objeto

Folder a partir de la clase Store y que haga referencia a esta carpeta. A continuación se usa el método open y se expecificael modo en que se desea abrirla:

- Si sólo queremos leer los mensajes, sin borrarlos, se debe usar el modo de sólo lectura READ_ONLY.
- En caso de que se quieran borrar los mensajes se debe usar el modo READ_WRITE. En el capítulo 4 profundizaremos más sobre manejo de carpetas y borrado de mensajes.

Así pues, el código quedaría:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
```

Con la carpeta ya abierta resulta sumamente sencillo recuperamos los mensajes almacenados en ella. Para ello usaremos el método getMessages de la clase Folder, sin parámetros, que devuelve un *array* de mensajes de tipo Message. Este método también permite que se le pase como parámetro un array de enteros (ordinales) con los números de los mensajes que queremos recuperar, o también que le pasemos dos enteros que indican que se desean recuperar los mensajes almacenados entre esas dos posiciones, ambas inclusive. En el caso que nos ocupa emplearemos la primera fórmula:

```
Message [] mensajes = folder.getMessages();
```

3.3.4 Procesamiento de mensajes

El último paso consiste en procesar los mensajes de alguna forma; en este ejemplo los mensajes serán mostrados por pantalla, presentado por cada mensaje su asunto, remitente (en el caso de que haya), fecha y hora en que fue enviado (recordar que POP3 no nos permite usar el método getReceviedDate para ver la fecha y hora en que ha sido recibido) y su contenido:

```
for (int i=0; i < mensajes.length; i++) {
    System.out.println(
            "Mensaje " + i + ":\n" +
            "\tAsunto: " + mensajes[i].getSubject() + "\n" +
            "\tRemitente: " + mensajes[i].getFrom()[0] + "\n" +
            "\tFecha de Envío: " + mensajes[i].getSentDate() + "\n" +
            "\tContenido: " + mensajes[i].getContent() + "\n");
}
```

Otra forma de procesar los mensajes sería, por ejemplo, escribir cada mensaje en un fichero de texto. Para ello se puede usar el método writeTo, que escribe el contenido de un mensaje además de una serie de atributos como el asunto, remitente, etc. El código quedaría:

```
for (int i=0; i < mensajes.length; i++) {
     OutputStream os = new FileOutputStream ("c:/Temp/msj" + i + ".txt");
```

Método	Comportamiento
Address[] getFrom()	Devuelve un *array* en el que hay tantas direcciones como remitentes, null si el campo *from* no está presente en el mensaje o un *array* vacío si el mensaje no contiene ninguna dirección en el *from*.
Date getSentDate()	Devuelve la fecha en la que el mensaje se envió.
String getSubject()	Devuelve el asunto del mensaje.
Object getContent()	Devuelve el contenido del mensaje Si éste es de tipo texto plano devolverá un String y si es de tipo Multipart devolverá un objeto de esta clase.
javax.activation.DataHandler getDataHandler()	Devuelve un DataHandler con el que poder manejar el contenido del mensaje. Con un DataHandler es posible manejar contenidos complejos a través de un canal de entrada

Tabla 3.4 Métodos para recuperar atributos de un mensaje. Todos pueden elevar la excepción MessagingException. Los métodos de color azul son de la clase Message y los de color verde de la interfaz Part..

```
            mensajes[i].writeTo(os);
            os.close();
        }
```

En el caso de usar esta alternativa, hay que tener en cuenta que el método writeTo puede lanzar una excepción de tipo IOException, por lo que al final del bloque **try-catch** hay que añadir un nuevo **catch** para que capture esta excepción; además hay que importar el paquete java.io.

Por otro lado, nótese que los atributos de los mensajes han sido obtenidos usando métodos contrarios a los de la tabla 3.3. Entre otros, estos métodos de recuperación aparecen en la tabla 3.4.

El último paso consiste en cerrar el Folder y el Store creados. Al cerrar el Folder se lepasa como parámetro un valor boolean. Si este vale true se aplicarán los cambios realizados sobre los mensajes, es decir, si hemos marcado un mensaje como borrado entonces éste se borrará. En este ejemplo no se han marcado los mensajes de ninguna forma, así que le pasamos false como parámetro. En el apartado 4.2 se explicará esto con más profundidad. El código sería:

```
folder.close(false);
store.close();
```

En cuanto a las excepciones, podrían producirse en los siguientes casos (todas de tipo MessagingException a no ser que se indique lo contrario):

- El método getStore de la clase Session lanzará una excepción si no puede manejar el protocolo especificado.
- El método connect de la clase Store lanzará una excepción si falla al intentar conectarse al servidor.
- El método getFolder de la clase Store lanzará una excepción si se intenta obtener una carpeta de un servidor al que no estamos conectados.
- El método open de la clase Folder lanzará una excepción si se intenta abrir una carpeta que no existe.
- El método getMessages de la clase Folder lanzará una excepción si se intenta obtener mensajes de una carpeta que no existe.
- Los métodos getSubject, getSentDate y getFrom de la clase Message lanzarán una excepción si fallan al intentar acceder al asunto o la fecha de un mensaje.
- El método getContent de la clase Message también puede generar una excepción, pero esta vez de tipo IOException. Esta excepción será lanzada cuando se produzca un error mientras se está recuperando el contenido del mensaje.
- El método close de la clase Folder lanzará una excepción si se intenta cerrar una carpeta que no está abierta.
- El método close de la clase Store lanzará una excepción si se produce algún error mientras se está cerrando el Store.

Por todo esto, el bloque de instrucciones que abarca desde que se obtiene el Store hasta que se cierra, debe incluirse dentro de un bloque **try-catch** que recoja los dos tipos de excepciones que se pueden dar.

```
try {
        Store store = sesion.getStore("pop3");
        …
        store.close();
} catch (MessagingException me) {
        System.err.println (me.getMessage());
} catch (IOException ioe) {
        System.err.println (ioe.getMessage());
}
```

3.3.5 Ejemplo completo

A continuación se muestra el ejemplo completo de como recuperar los mensajes de correo electrónico:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
```

```
public class ObtenerCorreo {
  public static void main (String [] args) {
    if (args.length != 2) {
       System.out.println(    "Ha de enviar dos parámetros\n" +
                              "java ObtenerCorreo usuario clave");
       System.exit(1);
    }
    // Obtener el usuario y la contraseña recibidos como parámetros
    String usuario = args [0];
    String clave   = args [1];
    // Obtenemos las propiedades del sistema
    String popHost   = "pop.correo.yahoo.es";
    Properties props = System.getProperties();
    // Obtener una sesión con las propiedades anteriormente definidas
    Session sesion = Session.getDefaultInstance(props,null);
    // Capturar las excepciones
    try {
       // Crear un Store indicando el protocolo de acceso y
       // conectarse a él
       Store store = sesion.getStore("pop3");
       store.connect(popHost,usuario,clave);
       // Crear un Folder y abrir la carpeta INBOX en modo SOLO LECTURA
       Folder folder = store.getFolder("INBOX");
       folder.open(Folder.READ_ONLY);
       // Obtener los mensajes almacenados en la carpeta
       Message [] mensajes = folder.getMessages();
       // Procesar los mensajes
       for (int i=0; i < mensajes.length; i++) {
       System.out.println("Mensaje "           + i + ":\n" +
                  "\tAsunto: "        + mensajes[i].getSubject()  + "\n" +
                  "\tRemitente: "     + mensajes[i].getFrom()[0]  + "\n" +
                  "\tFecha de Envío: " + mensajes[i].getSentDate() + "\n" +
                  "\tContenido: "     + mensajes[i].getContent()  + "\n");
       }
       //Cerrar el Folder y el Store
       folder.close(false);
       store.close();
    } catch (MessagingException me) {
       System.err.println(me.toString());
    } catch (IOException ioe) {
       System.err.println(ioe.toString());
    }
  }
}
```

Capítulo 4

Uso de banderines y gestión de buzones

4.1 Introducción

En este capítulo abordaremos la gestión de mensajes en el servidor desde dos puntos de vista diferentes. Por un lado, es posible marcar los mensajes en el servidor con objeto de realizar con ellos determinadas operaciones o bien con el único de fin de dejar constancia de que han sido procesados de una determinada manera: han sido vistos, han sido respondidos, son mensajes nuevos (recibidos después de la última conexión), etc. Por otro lado, mediante el protocolo IMAP es posible mantener una estructura jerárquica de buzones en el servidor, lo que facilita el correo eléctronico vía WebMail.

4.2 Uso de banderines

En el epígrafe 3.3.4 se habló sobre la necesidad de borrar los mensajes una vez recuperados éstos, con objeto de que no permanezcan indefinidamente en el servidor. Para realizar esta operación es necesario el uso de banderines, modelados en JavaMail a través de la clase Flags.

Siguiendo con el ejemplo del apartado 3.3, para borrar los mensajes sólo es necesario hacer tres modificaciones:

- Sustituir la línea:
  folder.open(Folder.READ_ONLY)
  por la línea:
  folder.open(Folder.READ_WRITE).
  Con esto estaremos indicando que el estado de esta carpeta y su contenido pueden ser modificados.
- En el bucle en el que se procesan los mensajes se debe añadir una línea de código cuyo objetivo es marcar cada mensaje para que sea borrado:

```
for (int i=0; i < mensajes.length; i++) {
        System.out.println("Mensaje " + i + ":\n" +
            "\tAsunto: "         + mensajes[i].getSubject()  + "\n" +
            "\tRemitente: "      + mensajes[i].getFrom()[0]  + "\n" +
            "\tFecha de Envío: " + mensajes[i].getSentDate() + "\n" +
            "\tContenido: "      + mensajes[i].getContent()  + "\n");
        Mensajes[i].setFlag(Flags.Flag.DELETED,true);
```

```
}
```

- Por último, al cerrar el objeto de tipo Folder hay que pasarle como parámetro el valor lógico true, para indicar que se apliquen todas las marcas o banderines establecidos lo que, en el caso que nos ocupa, implica que se eliminen todos los mensajes marcados como borrados:

```
folder.close(true);
```

Como ya se ha visto el borrado de mensajes implica el uso de banderines (*flags*). Un banderín define el estado de un mensaje que está contenido en una carpeta. Existen diferentes tipos de banderines, de manera que es posible activar en cada mensaje banderines distintos. Los banderines (de la clase Flag) pueden ser definidos por el usuario, aunque lo más normal es usar los que vienen predefinidos en JavaMail a través de la clase Flags.Flag. Los banderines predefinidos y su significado se muestran a continuación en la tabla 4.1.

Banderín	Cometido
ANSWERED	El mensaje ha sido respondido.
DELETED	El mensaje debe borrarse.
DRAFT	El mensaje no es definitivo, sino tan sólo un borrador.
FLAGGED	No tiene un significado concreto. El programador puede usarlo para algún propósito específico definido por él mismo.
RECENT	El mensaje es nuevo en esa carpeta, es decir, ha llegado desde la última vez que fue abierta.
SEEN	Indica si el mensaje ha sido visto. Su valor se establece automáticamente cuando se recupera el contenido del mensaje.
USER	Indica que la carpeta que contiene a este mensaje soporta Flags definidos por el usuario.

Tabla 4.1 Banderines predefinidos en la clase Flags.Flag.

La existencia de todos estos banderines no implica que los servidores los soporten todos. Los servidores de tipo POP3 sólo suelen soportar el banderín de borradomientras que los servidores basados en IMAP soportan alguna más. Para saber qué banderines son soportados por un servidor se puede usar el método getPermanentFlags() de la clase Folder, que devuelve un objeto de la clase Flags que contiene todos los banderines soportados. Así, el código es el siguiente:

```
Flags banderasSoportadas = folder.getPermanentFlags();
```

y se puede comprobar cuáles son los banderines soportados con el código:

```
if (banderasSoportadas.contains(Flags.Flag.ANSWERED))
```

```
            System.out.println("Soporta el banderín ANSWERED\n");
        else
            System.out.println("No soporta el banderín ANSWERED\n");
        //
        if (banderasSoportadas.contains(Flags.Flag.DELETED))
            System.out.println("Soporta el banderín DELETED\n");
        else
            System.out.println("No soporta el banderín DELETED\n");
        // etc.
```

Como ya se ha dejado entrever en el caso del borrado, para establecer un banderín y comprobar si ha sido establecido o no, se usan los métodos setFlag e isSet respectivamente, ambos de la clase Message, por ejemplo:

```
mensaje.setFlag(Flags.Flag.FLAGGED,true);
if (mensaje.isSet(Flags.Flag.FLAGGED))
    System.out.println ("El banderín FLAGGED del mensaje ha sido activado");
```

4.2.1 Ejemplo completo

A continuación se muestra un ejemplo completo en el que se realizan las dos operaciones anteriores. En este ejemplo se usa el protocolo IMAP en lugar de POP3 para conectarse al servidor. Esto tiene la ventaja de que, además de ilustrar la conexión a IMAP, es posible usar más banderines que con un servidor POP3.

Acceder a un servidor usando IMAP es muy similar a hacerlo usando POP3, de hecho, sólo es necesario cambiar el nombre del protocolo de acceso, modificando la línea de código:

```
Store store = sesion.getStore("pop3");
```

por:

```
Store store = sesion.getStore("imap");
```

Así pues, el código completo quedaría:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class ManejoBanderas {
    public static void main (String [] args) {
        if (args.length != 2) {
          System.out.println(          "Ha de enviar dos parámetros\n" +
                                       "java ManejoBanderas usuario clave");
            System.exit(1);
    }
    // Obtener el usuario y la clave recibidos como parámetros
    String usuario = args [0];
    String clave   = args [1];
```

```
// Obtener las propiedades del sistema
String imapHost   = "imap.runbox.com";
Properties props = System.getProperties();
// Obtener una sesión con las propiedades anteriormente definidas
Session sesion = Session.getDefaultInstance(props,null);
// Capturar las excepciones
try {
   // Crear un Store indicando el protocolo de acceso y conectarse a él
   Store store = sesion.getStore("imap");
   store.connect(imapHost,usuario,clave);
   // Crear un Folder y abrir la carpeta INBOX en modo
   // LECTURA/ESCRITURA
   Folder folder = store.getFolder("INBOX");
   // Si se abriese en modo de sólo lectura no se podrían marcar los
// mensajes y por tanto parecerá que no se soporta ningún banderín
   folder.open(Folder.READ_WRITE);
   // Obtener las banderas soportadas y mostramos cuáles son
   Flags banderasSoportadas = folder.getPermanentFlags();
   if (banderasSoportadas.contains(Flags.Flag.ANSWERED))
      System.out.println("Soporta el banderín ANSWERED\n");
   else
      System.out.println("No soporta el banderín ANSWERED\n");
   if (banderasSoportadas.contains(Flags.Flag.DELETED))
      System.out.println("Soporta el banderín DELETED\n");
   else
      System.out.println("No soporta el banderín DELETED\n");
   if (banderasSoportadas.contains(Flags.Flag.DRAFT))
      System.out.println("Soporta el banderín DRAFT\n");
   else
      System.out.println("No soporta el banderín DRAFT\n");
   if (banderasSoportadas.contains(Flags.Flag.FLAGGED))
      System.out.println("Soporta el banderín FLAGGED\n");
   else
      System.out.println("No soporta el banderín FLAGGED\n");
   if (banderasSoportadas.contains(Flags.Flag.RECENT))
      System.out.println("Soporta el banderín RECENT\n");
   else
      System.out.println("No soporta el banderín RECENT\n");
   if (banderasSoportadas.contains(Flags.Flag.SEEN))
      System.out.println("Soporta el banderín SEEN\n");
   else
      System.out.println("No soporta el banderín SEEN\n");
   if (banderasSoportadas.contains(Flags.Flag.USER))
      System.out.println("Soporta el banderín USER\n");
   else
      System.out.println("No soporta el banderín USER\n");
   // Obtener el mensaje número 1 de todos los mensajes
```

```
            Message mensaje = folder.getMessage(1);
            // Establecer el banderín FLAGGED a true
            mensaje.setFlag(Flags.Flag.FLAGGED,true);
            // Comprobar si el banderín FLAGGED de este mensaje ha sido
            // establecido
            if (mensaje.isSet(Flags.Flag.FLAGGED))
               System.out.println ("El banderín FLAGGED ha sido activado");
            // Cerrar el Folder y el Store
            folder.close(true);
            store.close();
         } catch (MessagingException me) {
            System.err.println(me.toString());
         } catch (IndexOutOfBoundsException ioe) {
            System.err.println("No podemos acceder al mensaje número 1"
   + ioe.toString());
         }
      }
}
```

4.3 Gestión de buzones y carpetas

Un buzón de correo es el espacio físico que un servidor de correo reserva para cada usuario en sus unidades de disco. A pesar de que en cada buzón se almacena el correo perteneciente a un usuario, algunos servidores permiten compartir buzones, esto es, un mismo buzón puede pertenecer a varios usuarios.

Cada buzón se suele corresponder con un directorio. La forma de almacenar los mensajes depende del programa que los gestione, pudiéndose almacenar en un solo fichero (normalmente comprimido), o guardar cada mensaje en un archivo distinto.

En ocasiones, el concepto de buzón se usa como sinónimo de dirección de correo, lo que puede dar lugar a confusiones, puesto que las direcciones de correo son únicas mientras que un mismo buzón puede corresponderse con varias direcciones de correo.

Hay que destacar las distintas capacidades de tratamiento de buzones que existen en IMAP y en POP. Como ya se comentó en el epígrafe 1.5.3, en POP sólo hay acceso a un único buzón. En la mayoría de servidores este buzón se maneja a modo de carpeta y se accede a él con el nombre de **INBOX**. En IMAP, sin embargo, hay acceso a múltiples buzones que pueden manipularse usando JavaMail. Esto quiere decir que con IMAP es posible acceder a todas las carpetas disponibles en un buzón, mientras que con POP sólo se puede acceder a la carpeta **INBOX**.

Antes de ver un ejemplo de gestión de buzones es necesario comentar que la clase Folder, que usaremos como clase base para este ejemplo, representa una carpeta que contiene mensajes de correo, otras carpetas o ambas cosas. Todas las carpetas

contenidas en un objeto Folder serán descendientes de la carpeta **INBOX**, que se corresponde con la bandeja de entrada. En la tabla 4.2 aparecen algunos de los métodos de esta clase que usaremos en el próximo ejemplo. Algunos de los métodos que aparecen en esta tabla son abstractos, lo que implica que la clase Folder no se pueda usar directamente, sino indirectamente a través de sus clases hijas como pueda ser IMAPFolder.

4.3.1 Operaciones sobre las carpetas de un buzón

A continuación se mostrará un ejemplo que usa algunas de las capacidades que JavaMail ofrece para el tratamiento de buzones y carpetas. En este ejemplo es necesario un servidor basado en IMAP, ya que este tipo es el que permite tratar con distintas carpetas. Veremos como crear una carpeta, obtener las carpetas existentes en wl buzón de correo y sus atributos, ver las carpetas anidadas en otra carpeta, copiar mensajes entre carpetas, recuperar los mensajes de una carpeta distinta de la carpeta **INBOX** y borrar una carpeta. El poder crear carpetas, el número máximo de carpetas

Método	Comportamiento
boolean create(int type)	Crea una carpeta que será del tipo indicado usando las constantes de la clase Folder
boolean delete(boolean recurse)	Elimina la carpeta que llame a este método, eliminando o no sus subcarpetas en función de que se le pase true o false respectivamente
Folder getFolder(String name)	Devuelve la carpeta correspondiente al nombre indicado, si bien éste no tiene porqué existir. Para ver si existe se usa el método exists
Message[] getMessages()	Devuelve un *array* con todos los mensajes contenidos en esta carpeta
Folder[] list(String pattern)	Retorna las carpetas contenidas en esta carpeta que se correspondan con el patrón indicado. Usando "*" nos devuelve la estructura completa de carpetas incluida la que llama al método
void open(int mode)	Abre la carpeta en modo sólo lectura o lectura/escritura en función de que se le pase Folder.READ_ONLY o Folder.READ_WRITE respectivamente

Tabla 4.2 Métodos principales de la clase abstracta Folder. Todos pueden elevar la excepción MessagingException.

existentes o el poder crear carpetas anidadas son características que dependen del servidor que estemos usando.

4.3.1.1 Creación de una carpeta

Como ocurre en los ejemplos en los que se recuperaban los mensajes de un servidor, hay que obtener las propiedades del sistema, crear una sesión y un Store y conectarse a este último; por tanto el principio del código será el mismo que en los ejemplos anteriores. La única diferencia consiste en que hay que importar la clase IMAPFolder, que es sobre la que se basa todo el tratamiento posterior.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
import com.sun.mail.imap.IMAPFolder;
public class ObtenerCarpetas {
    public static void main (String [] args) {
        if (args.length != 2) {
            System.out.println(    "Ha de enviar dos parámetros\n" +
                "java ObtenerCorreoIMAP usuario clave");
            System.exit(1);
        }

        String usuario = args [0];
        String clave   = args [1];
        String popHost   = "imap.runbox.com";
        Properties props = System.getProperties();
        Session sesion = Session.getDefaultInstance(props,null);
        try {
            Store store = sesion.getStore("imap");
            store.connect(popHost,usuario,clave);
        …
```

Ahora hay que crear un Folder, pero a diferencia de los ejemplos anteriores, usaremos el método de la clase Store getDefaultFolder() que devuelve la raíz de la estructura de carpetas que contiene el buzón de correo. Este objeto Folder se posteriormente para ver las carpetas existentes en el buzón, pero no para recuperar mensajes, por tanto no es necesario abrirlo, es decir, no es necesario usar el método Folder.open():

```
Folder folder = store.getDefaultFolder();
```

La primera operación que pondremos a prueba será la creación de una carpeta. Antes de crearla, hay que comprobar que la carpeta raíz puede contener otras carpetas, para lo cual hay que usar el método getType() de la clase Folder y contrastarlo con la constante Folder.HOLDS_FOLDERS, utilizando para ello el

operador &. Si el resultado vale 0, entonces el objeto Folder no puede contener carpetas anidadas, y por tanto no podremos crear ninguna carpeta dentro:

```
if ((folder.getType() & Folder.HOLDS_FOLDERS) != 0) {
```

Una vez hecha esta comprobación, ya se puede pasar a crear la carpeta interna. Para esto basta con ejecutar dos instrucciones; en la primera se utiliza el método getFolder() que devuelve un objeto de este tipo ubicado en el camino (*path*) que le hayamos pasado como parámetro:

```
Folder universidadFolder = folder.getFolder("INBOX.Universidad");
```

Como puede verse en esta sentencia, la carpeta de la que cuelgan todas las demás es la carpeta INBOX, y cuando una carpeta de nombre **NuevaCarpeta** cuelga de otra la forma de referenciarla es **INBOX.NuevaCarpeta**; en nuestro caso, la carpeta creada se llama **Universidad**, por lo que el parámetro pasado es **"INBOX.Universidad"**. Una vez hecho esto, la segunda instrucción a realizar consiste en invocar el método create que hará que se cree definitivamente esta carpeta, indicándole el tipo, para lo cual usamos las constantes de la clase Folder: usaremos Folder.HOLD_MESSAGES para que la carpeta pueda contener mensajes:

```
universidadFolder.create(Folder.HOLDS_MESSAGES);
```

Si se desea que la carpeta pueda contener, a su vez, otras carpetas, será necesario añadir además la constantes Folder.HOLDS_FOLDERS:

```
universidadFolder.create(  Folder.HOLDS_MESSAGES &
                           Folder.HOLDS_FOLDERS);
```

4.3.1.2 Acceso a la estructura de carpetas de un buzón

Para conocer la estructura de carpetas existente a partir de **INBOX**, se usa el método list de la clase Folder. Este método recibe un patrón en forma de String. En este ejemplo usa el patrón "*" lo que devuelve todas las carpetas contenidas en la estructura jerárquica de**INBOX**; el resultado lo almacenaremos en un *array* de objetos Folder. Una vez hecho esto, basta con recorrer ese *array* y mostrar la información que se desee acerca de cada carpeta, como por ejemplo su nombre completo, sus atributos, y el nombre de su carpeta padre.

Para mostrar los atributos de una carpeta almacenada en un servidor IMAP es necesario hacer una conversión de tipo (*casting*) de cada objeto Folder del *array* en un objeto de la clase IMAPFolder; esto es necesario porque para mostrar los atributos hay que usar el método getAttributes() que devuelve un *array* de String, y que pertenece a la clase IMAPFolder, pero no a la clase Folder.

Para mostrar el nombre de la carpeta padre de un objeto Folder, es decir la carpeta de la que depende en la estructura jerárquica, es posible hacer uso del método getParent() que devuelve el objeto Folder padre; a continuación puede enviarse el mensaje getName() a dicho objeto para conocer su nombre. Si una carpeta no tiene padre no se mostrará nada.

Así pues, para ver la estructura jerárquica completa de carpetas almacenadas en el buzón se utiliza el siguiente código:

```
Folder [] listaCarpetas = folder.list("*");
for (int i = 0; i < listaCarpetas.length; i++) {
        System.out.println ("Carpeta: " + listaCarpetas[i].getFullName());
        IMAPFolder imf = (IMAPFolder)listaCarpetas[i];
        String [] atributos = imf.getAttributes();
        System.out.println ("\t" + "Atributos: ");
        for (int j= 0; j < atributos.length; j++)
                System.out.println ("\t" + atributos[j] + "\n");
        System.out.println (    "\t"+ "Carpeta Padre: " +
                                listaCarpetas[i].getParent().getName() + "\n");
```

Cuando el código se haya ejecutado la salida será:

```
Carpeta: INBOX.Universidad
        Atributos:
        \HasNoChildren
        Carpeta Padre: INBOX
Carpeta: INBOX.Trash
        Atributos:
        \HasNoChildren
        Carpeta Padre: INBOX
Carpeta: INBOX.Sent
        Atributos:
        \HasNoChildren
        Carpeta Padre: INBOX
Carpeta: INBOX
        Atributos:
        \Marked
        \HasChildren
        Carpeta Padre:
```

De donde se deduce que, gráficamente, que la estructura jerárquica de las carpetas contenidas en el buzón es la de la figura 4.3.

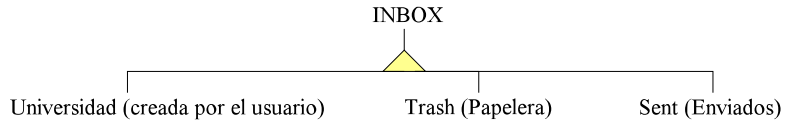

Figura 4.3 Jerarquía de carpetas a partir de **INBOX**.

En el resultado que aparece por pantalla puede apreciarse que el nombre de una carpeta seguido de un punto y de otro nombre de carpeta indica que la segunda está contenida en la primera. Además, se observa que las carpetas **Universidad**, **Trash** y **Sent** no contienen a otras carpetas, lo que viene indicado por su atributo **HasNoChildren**. Estas carpetas se corresponden con las que podemos ver en el servidor si se accede directamente vía WebMail, tal y como se muestra en la figura 4.4.

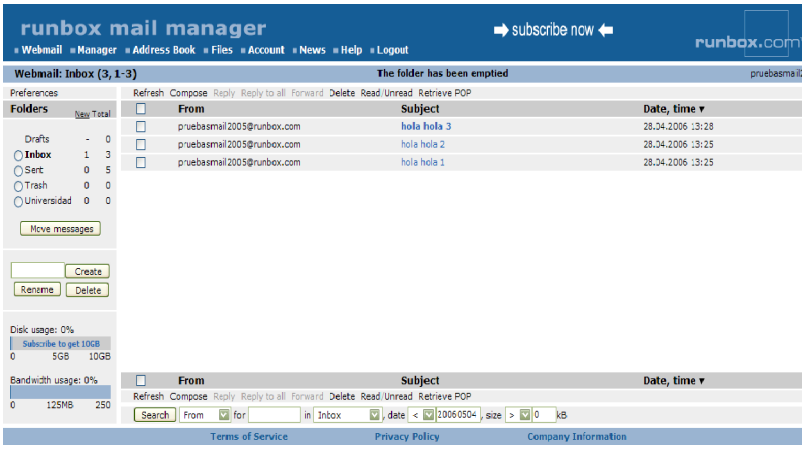

Figura 4.4 Vista de la estructura de carpetas vía WebMail

Como ya se ha comentado, no hay que confundir la estructura jerárquica de carpetas con la forma en que éstas y los mensajes están guardados en el espacio de almacenamiento físico del servidor. La estructura jerárquica es la forma que -usando JavaMail- las carpetas se nos presentan como usuarios, pero no es una representación literal del almacenamiento en disco del servidor.

4.3.1.3 Copia de mensajes entre carpetas

La siguiente operación que pondremos a prueba es copiar mensajes entre dos carpetas. Este proceso se realiza usando el método copyMessages (Message[] msgs, Folder folder) de la clase Folder, el cual copiará un *array* de mensajes pertenecientes al objeto Folder que hace la llamada, a la carpeta destino que se le pasa como parámetro.

Para ello, lo primero será crear un *array* sin inicializar de objetos de la clase Message, que debe tener tantas posiciones como mensajes se deseen copiar. En este caso copiaremos sólo dos mensajes, por lo que el *array* debe tener dos posiciones:

```
Message [] mensajesCopiar = new Message[2];
```

Para copiar los mensajes es necesario disponer de los objetos Folder de origen y destino. El destino será la carpeta **Trash**, o sea donde copiaremos los mensajes, y el origen la carpeta **INBOX**, de donde tomaremos los mensajes a copiar. Es importante destacar que hay que poner el nombre completo de la carpeta, es decir, no vale con poner **"Trash"**, sino que hay que especificar **"INBOX.Trash"**. Estas dos carpetas hay que abrirlas en el modo necesario: **INBOX** en modo sólo lectura, pues sólo vamos a copiar mensajes de él (no los vamos a eliminar) y el otro en modo

lectura/escritura, pues es donde vamos a copiar los mensajes. Una vez obtenidos los dos primeros mensajes de **INBOX** esta carpeta ya puede ser cerrada.

Por último, estos se añaden a la cola de la lista de mensajes de la carpeta Trash, usando el método copyMessages(...) que comentamos anteriormente:

```
Message [] mensajesCopiar = new Message[2];
Folder trashFolder = store.getFolder("INBOX.Trash");
trashFolder.open (Folder.READ_WRITE);
Folder INBOXFolder = store.getFolder("INBOX");
INBOXFolder.open (Folder.READ_ONLY);
mensajesCopiar[0] = INBOXFolder.getMessage(1);
mensajesCopiar[1] = INBOXFolder.getMessage(2);
INBOXFolder.copyMessages(mensajesCopiar, trashFolder);
INBOXFolder.close(false);
```

Para mostrar que, efectivamente, ha funcionado la copia de mensajes entre carpetas, a continuación recuperaremos los mensajes de la carpeta **Trash** y los mostraremos por pantalla de igual manera a como se hizo anteriormente con la carpeta **INBOX**, pero indicándo como nombre **INBOX.Trash**. Para hacer esto puede reutilizarse la variable trashFolder creada en el paso anterior:

```
Message [] mensajes = trashFolder.getMessages();
for (int i= 0; i < mensajes.length; i++) {
     System.out.println("Mensaje " + i + ":\n" +
               "\tAsunto: " + mensajes[i].getSubject()  + "\n" +
               "\tRemitente: " + mensajes[i].getFrom()[0]  + "\n" +
               "\tFecha de Envío: " + mensajes[i].getSentDate() + "\n" +
               "\tContenido: " + mensajes[i].getContent()  + "\n");
}
trashFolder.close(false);
```

4.3.1.4 Borrado de una carpeta

La última operación sobre la que centraremos nuestra atención será borrar una carpeta, concretamente la carpeta **Universidad**, que fue lo primero que se hizo en el apartado 4.3.1.1. Para ello sólo hay que aplicar el método delete sobre el objeto Folder usado anteriormente para crear la carpeta; este método borra la carpeta representada por el objeto que invoca al método, y tiene un parámetro de tipo boolean: se le pasa true si se quiere controlar también el borrado de sus subcarpetas, lo que en este ejemplo no es necesario pues **Universidad** no contiene carpetas anidadas:

```
universidadFolder.delete(false);
```

4.3.2 Ejemplo completo

A continuación se muestra el ejemplo completo con todas las operaciones realizadas anteriormente. Comentar únicamente que el tratamiento de las excepciones es el mismo que el de anteriores capítulos.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
import com.sun.mail.imap.IMAPFolder;


public class ObtenerCarpetas {
   public static void main (String [] args) {
      if (args.length != 2) {
         System.out.println(          "Ha de enviar dos parámetros\n" +
                                      "java ObtenerCorreoIMAP usuario clave");
         System.exit(1);
      }
   // Obtener el usuario y la clave recibidos como parámetros
      String usuario = args [0];
      String clave   = args [1];
   // Obtener las propiedades del sistema
      String popHost   = "imap.runbox.com";
      Properties props = System.getProperties();
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
      try {
      // Crear un Store indicando el protocolo de acceso y
      // conectarse a él
         Store store = sesion.getStore("imap");
         store.connect(popHost,usuario,clave);
      // Crear un objeto Folder y obtener la carpeta por defecto
         Folder folder = store.getDefaultFolder();
         if ((folder.getType() & Folder.HOLDS_FOLDERS) != 0) {
         //========================================
         // Crear una carpeta llamada Universidad
         //========================================
            Folder universidadFolder = Folder.getFolder("INBOX.Universidad");
            universidadFolder.create(Folder.HOLDS_MESSAGES);
         //========================================
         // Obtener las carpetas almacenadas bajo la carpeta por defecto
         //========================================
            Folder [] listaCarpetas = folder.list("*");
         // Nombre de cada carpeta y sus atributos
            for (int i = 0; i < listaCarpetas.length; i++) {
```

```
                    System.out.println("Carpeta:"   +
listaCarpetas[i].getFullName());
                IMAPFolder imf = (IMAPFolder)listaCarpetas[i];
                String [] atributos = imf.getAttributes();
                System.out.println ("\t" + "Atributos: ");
                for (int j= 0; j < atributos.length; j++)
                   System.out.println ("\t" + atributos[j] + "\n");
             //Carpeta padre de cada carpeta
                    System.out.println  ("\t"+  "Carpeta  Padre:  " +
listaCarpetas[i].getParent().getName() + "\n");
             }
          }
       //==========================
       // Copiar mensajes
       //==========================
          Message [] mensajesCopiar = new Message[2];
          Folder trashFolder = store.getFolder("INBOX.Trash");
          trashFolder.open (Folder.READ_WRITE);
          Folder INBOXFolder = store.getFolder("INBOX");
          INBOXFolder.open (Folder.READ_ONLY);
       // Concatenar a la lista de mensajes de la carpeta que le pasamos
       // como parámetro, el array de mensajes formado con los mensajes
       // de la carpeta origen
          mensajesCopiar[0] = INBOXFolder.getMessage(1);
          mensajesCopiar[1] = INBOXFolder.getMessage(2);
          INBOXFolder.copyMessages(mensajesCopiar,trashFolder);
          INBOXFolder.close(false);
       //===========================================
       // Obtener los mensajes de la carpeta INBOX.Trash
       //===========================================
          Message [] mensajes = trashFolder.getMessages();
       // Procesar los mensajes
          for (int i= 0; i < mensajes.length; i++) {
             System.out.println("Mensaje "           + i + ":\n" +
                "\tAsunto: "        + mensajes[i].getSubject()  + "\n" +
                "\tRemitente: "     + mensajes[i].getFrom()[0]  + "\n" +
                "\tFecha de Envío: " + mensajes[i].getSentDate() + "\n" +
                "\tContenido: "     + mensajes[i].getContent()  + "\n");
          }
          trashFolder.close(false);
       //==========================
       // Borrar una carpeta
       //==========================
          universidadFolder.delete(false);
          store.close();
       } catch (MessagingException me) {
          System.err.println(me.toString());
       } catch (IOException ioe) {
```

```
            System.err.println(ioe.toString());
        }
    }
}
```

4.4 Resumen de paquetes y clases

En este apartado, y a modo de resumen, se muestran los paquetes que componen JavaMail y se profundiza en las principales clases que los integran, si bien en los apartados anteriores hemos visto ya el uso de alguna de estas clases, y en los capítulos posteriores veremos otras nuevas.

En la tabla 4.3 pueden verse los paquetes que componen JavaMail y la descripción de cada uno de ellos, mientras que los siguientes epígrafes se centran en las clases más importantes y para qué se utilizan.

Paquete	Descripción
com.sun.mail.imap	Proporciona soporte para manejar el protocolo IMAP y poder acceder a un servidor de correo que use este protocolo.
com.sun.mail.pop3	Proporciona soporte para manejar el protocolo POP3 y poder acceder a un servidor de correo que use este protocolo.
com.sun.mail.smtp	Proporciona soporte para manejar el protocolo SMTP y poder usar un servidor basado en este protocolo para el envío de mensajes.
javax.mail	Contiene clases que modelan un sistema de correo, definiendo capacidades que son comunes a la mayoría de ellos. Este paquete es el más importante de los que componen JavaMail.
javax.mail.event	Define clases que se usan para el manejo de eventos producidos por las clases que componen JavaMail.
javax.mail.internet	Define características que son específicas de sistemas de correo que funcionan sobre la red Internet, y que se basan en las extensiones MIME.
javax.mail.search	Proporciona mecanismos para realizar búsquedas de mensajes según una serie de criterios, dentro de una carpeta.
javax.mail.util	Proporciona tres clases de utilidad que controlan distintos tipos de canales (*streams*).

Tabla 4.3 Paquetes principales de la API JavaMail

4.4.1 La clase Message

Message es una clase abstracta que modela un mensaje de correo electrónico mediante un conjunto de atributos que definen su cabecera, y un contenido que define su cuerpo. Entre los atributos de la cabecera están los típicos de *from*, *fecha de envío*, *asunto*, etc., que se insertan y se recuperan mediante los métodos de las tablas 3.3 y 3.4. Para hacer referencia a los destinatarios de un mensaje, la clase Message proporciona la subclase Message.RecipientType que incluye constantes que hacen referencia a *to*, *Bcc* y *Cc*.

Por defecto, el contenido es de tipo String conteniendo texto plano, pero se puede enviar otro tipo de información haciendo uso de otras clases que heredan de Message. Una de ellas es MimeMessage, que implementa el RFC822 y los estándares MIME, y proporciona una gran potencia al permitir, incluso, construir mensajes constituídos por bloques que contienen información de diferente índole: texto, gráficos, música, documentos de aplicaciones ofimáticas (WordPerfect, StarOffice, Macromedia Flash, etc.), etc.

Ni que decir tiene que esta clase se utiliza tanto para modelar los mensajes a enviar (que suelen crearse haciendo uso de MimeMessage), como para los que se reciben desde un buzón a través de la clase Folder.

Esta clase implementa la interfaz Part, que define un conjunto de características comunes a la mayoría de sistemas de correo, además de métodos para obtener y establecer el contenido del mensaje, y de algunos de los campos de la cabecera del mismo. En las tablas 3.3 y 3.4. puede distinguirse entre los métodos propios de Message y los de la interfaz Part.

4.4.2 La clase Session

Session define una sesión de conexión a una cuenta de correo. A este respecto, permite modificar opciones de configuración a la hora de conectarse y autenticarse con objeto de interactuar correctamente con el servidor de correo.

Esta clase no tiene un constructor público, por lo que para obtener una sesión de correo tendremos que usar uno de los dos métodos estáticos que proporciona Session: getDefaultInstance y getInstance. El primer método devuelve siempre el mismo objeto Session, esto es, no crea una sesión nueva cada vez por lo que facilita el que la sesión de correo sea compartida por varias aplicaciones. Es el método más utilizado porque consume pocos recursos y sumnistra la funcionalidad suficiente en la mayoría de los casos. El segundo método crea una sesión diferente cada vez que se invoca, por lo que no es compartida por distintas aplicaciones.

Sea cual sea el método usado, las características de configuración para la conexión han debido cargarse previamente en un objeto de tipo java.util.Properties, el cual hay que pasarle como primer parámetro de los dos que necesita. Estas propiedades se pueden obtener (y modificar posteriormente) a partir de las propiedades del sistema (usando el método System.getProperties() como ya se ha visto en capítulos anteriores) o creando un objeto vacío con el constructor Properties() y añadiéndole las propiedades que se necesiten. Desde un punto de vista pragmático, un objeto Properties no es más que una tabla de dispersión (tabla *hash*) que contiene pares (propiedad-valor). En la tabla 4.4 pueden verse las propiedades más importantes en relación con el correo electrónico, si bien para conocerlas todas puede consultarse la documentación oficial.

El segundo parámetro necesario para construir una sesión es un objeto de tipo Authenticator que puede ser null en caso de que el servidor no necesite autenticación. En el capítulo 6 se profundizará sobre este aspecto.

4.4.3 Las clases Store y Folder

Conceptualmente un objeto Store representa un buzón de correo dentro del servidor, es decir, la cuenta de correo a la que conectarsepara recibir y desde la que

Propiedad	Descripción
mail.debug	Hace que el programa se ejecute en modo depurador, es decir, muestre por pantalla todas las operaciones que realiza y los posibles errores. Por defecto no está activada. P.e. props.put ("mail.debug","true").
mail.from	Añade la dirección del remitente en el mensaje. P.e. props.put (“mail.from”, lenin@gmail.com).
mail.*protocolo*.host	Dirección del servidor para el protocolo especificado, es decir, el que se va a usar para enviar o recibir. P.e. props.put ("mail.smtp.host" ,"smtp.ono.com").
mail.*protocolo*.port	Número de puerto por el que escucha el servidor. P.e. props.put ("mail.pop3.port","995").
mail.*protocolo*.auth	Indica si vamos a autenticarnos en el servidor o no. P.e. props.put("mail.smtp.auth", "true").
mail.*protocolo*.user	Nombre de usuario cuando nos conectamos al servidor. P.e. props.put ("mail.pop3.user","pruebasmail2005").

Tabla 4.4 Propiedades más comunes al establecer una sesión de correo electrónico

enviar mensajes. Esta clase permite manipular la estructura de carpetas (objetos de tipo Folder) y los mensajes contenidos en cada una de ellas.

La conexión a un objeto Store se hace a partir de un objeto Session utilizando los métodos getStore en sus diferentes versiones (la m´s usual consiste en indicar sencillamente el protocolo que se desea usar para la conexión con el servidor especificado en las propiedades de la sesión).

Una vez conectados a un Store podremos obtener una carpeta determinada (con el método getFolder()) u obtener la carpeta por defecto (con el método getDefaultFolder()). La carpeta por defecto es la raíz de la estructura de carpetas almacenadas en la cuenta de correo. Esto se estudió desde un punto de vista práctico en el apartado 4.3.1.

Una sesión de correo debe finalizar cerrando el objeto Store mediante el método close().

Por último comentar que una sesión puede producir eventos cuando se intenta crear, borrar o renombrar alguna de sus carpetas; estos eventos pueden ser capturados a través de un objeto delegado de tipo FolderListener.

Una vez obtenido un objeto Store, se hace posible acceder a sus carpetas con objeto de manejar los mensajes de correo que éstas contienen. Así, la clase Folder representa una carpeta almacenada en el buzón de correo, y puede contener mensajes de correo, otras carpetas o ambas cosas, según se especifique en sus propiedades. La carpeta principal o carpeta por defecto es la carpeta **INBOX**, que se corresponde con la bandeja de entrada, y todas las demás carpetas son descendientes -directa o indirectamente- de la carpeta **INBOX**.

Es importante señalar también que el manejo de una carpeta depende del protocolo que usado. Por ejemplo, con el protocolo IMAP es posible acceder a cualquier carpeta dentro del buzón: bandeja de entrada (**INBOX**), papelera, enviados, etc., además de las carpetas definidas por el usuario. Estas carpetas se acceden siguiendo un método parecido al utilizado en los sistemas operativos, donde se utiliza una barra separadora entre el nombre de un directorio y el siguiente. En este caso se suele utilizar un punto ('.') como separador teniéndose, por ejemplo, **INBOX.Trash**. Por otro lado, utilizando el protocolo POP3 sólo es posible acceder a la bandeja de entrada, es decir, a la carpeta **INBOX**. En la tabla 4.2 apreciamos los métodos más importantes de la clase Folder.

4.4.4 La clase Transport

Esta es la clase encargada de enviar el correo a su destinatario. En la mayoría de las ocasiones se usará el método estático Transport.send(), si bien podría

obtenerse un objeto de esta clase usando el método getTransport() de la clase Session, sin parámetros, o indicándole el protocolo que usamos o la dirección del receptor del mensaje.

Para esto último, la clase Transport permite conectarse a un servidor y enviar los mensajes de la siguiente forma:

- Se obtiene un objeto Transport para el protocolo indicado:
  Transport transport = sesion.getTransport("smtp");
- Nos conectamos a él (habiendo añadido en las propiedades la dirección del servidor mediante props.put("mail.smtp.host",smtpHost)):
  transport.connect();
- Se envía el mensaje. Debe usarse getAllRecipients() sobre el mensaje para obtener todas las direcciones que previamente se hayan añadido en el mensaje:
  transport.sendMessage(mensaje,mensaje.getAllRecipients());
- Cerrar el objeto:
  transport.close();

Capítulo 5

Correo electrónico multiparte

5.1 Introducción

En este capítulo trataremos sobre los correos electrónicos que, o bien poseen ficheros adjuntos, o bien están formados por texto e imágenes en formato HTML. En cualquier caso, cuando se trabaja con correos de este tipo es necesario usar las extensiones MIME.

Como ya se comentó en el apartado 1.5.4, MIME está orientado a proporcionar facilidades para el intercambio de distintos tipos de contenidos, ya sea audio, video o imágenes en diferentes formatos, documentos procedentes de una amplia diversidad de aplicaciones ofimáticas, etc., además de proporcionar soporte para la internacionalización del lenguaje, es decir, poder incluir caracteres propios de distintos alfabetos, como es el caso de la letra eñe ('ñ') del idioma español. Así, con el estudio de los ejemplos de este capítulo se pondrán en evidencia las características que JavaMail proporciona para el manejo de los tipos MIME.

A modo de resumen podemos decir que las clases que más intervienen cuando se trabaja con correos en HTML o formados por adjuntos, son las clases Message, MimeMessage, MultiPart, MimeMultiPart, BodyPart, MimeBodyPart y la interfaz Part, que iremos viendo en los siguientes apartados. El proceso general

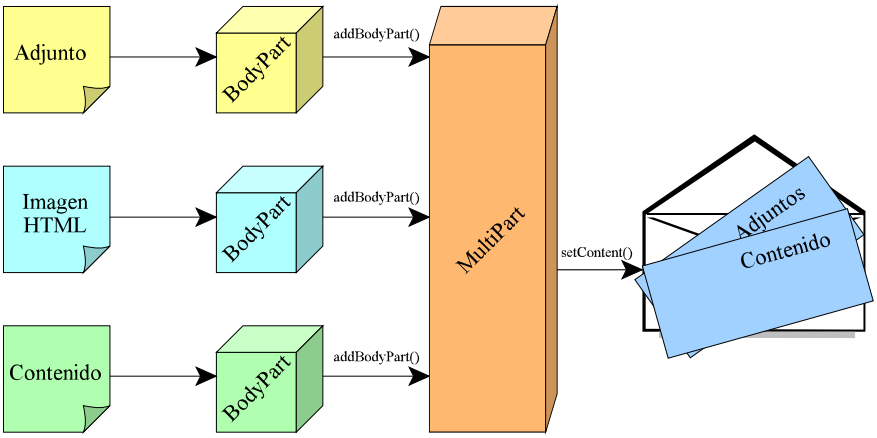

Figura 5.5 Composición de un mensaje con adjuntos o con HTML incrustado

es igual que en los ejemplos ya vistos en capítulos anteriores pero, además, tendremos que crear una serie de objetos de tipo BodyPart. Cada uno de estos objetos se corresponderá con uno de los adjuntos, con el texto del mensaje en sí o con una de las imágenes si se trata de correo HTML. Luego se van añadiendo a un objeto MultiPart y con éste se rellena el mensaje. En la figura 5.5 puede verse gráficamente este proceso.

5.2 Envío de mensajes HTML

En este epígrafe se mostrará cómo crear mensajes de correo en formato HTML. Para poder visualizar estos mensajes, es necesario disponer de un gestor de correo que sea capaz de manejar HTML, como por ejemplo Eudora, Microsoft Outlook, Thunderbird, etc.

Las dos ejemplos que aquí se tratan, toman un fichero HTML con imágenes (almacenadas en disco) y lo colocan como contenido del mensaje. La diferencia es que, en el primer ejemplo, no se envían las imágenes almacenadas en disco sino tan solo la URL de estas imágenes; en el segundo caso añadiremos las imágenes desde el disco y el documento HTML viajará completo por la red.

5.2.1 Mensaje HTML con referencia a una URL

En este ejemplo se muestra cómo enviar un mensaje de correo electrónico basado en HTML. Para ello usaremos un fichero de código HTML previamente descargado en disco. Las imágenes no estan contenidas en el propio fichero HTML sino que, por cada imagen, se especifica la URL que la contiene, es decir, la dirección de Internet donde se encuentra almacenada.

El primer paso será acceder a una página web y guardar su contenido, para usar este fichero como el contenido HTML a enviar en el correo. La dirección que usaremos en este ejemplo será http://www.google.com/, la dirección de Google en inglés. Una vez escrita esta dirección en un navegador, se selecciona la opción del menú **Archivo->Guardar Como**. A continuación se le da un nombre, por ejemplo **"Google.html"**, dejando el tipo de archivo que aparece por defecto que será **"Página Web, completa"**. En este ejemplo se asume que este fichero se almacena en el directorio **"c:\Temp"** aunque, obviamente, se puede almacenar en cualquier otro directorio. Este paso puede apreciarse en la figura 5.6.

El fichero que se acaba de grabar en el disco duro tiene modificadas sus URLs para hacer referencia a los ficheros locales que también se han grabado. El siguiente paso consiste en restituir la URL de la imagen que se va a referenciar, y que

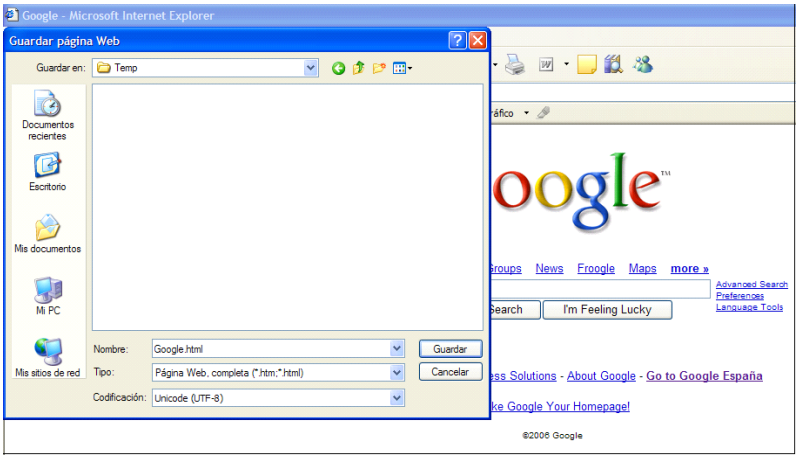

Figura 5.6 Descarga y almacenamiento de un fichero HTML.

es la del logotipo de Google; para conocer su URL original basta situar el cursor del ratón sobre la imagen, pulsar el botón derecho y seleccionar la opción de **"Propiedades"**; a continuación se selecciona la dirección que aparece en **"Dirección (URL)"** y se copia al portapapeles; véase la figura 5.7.

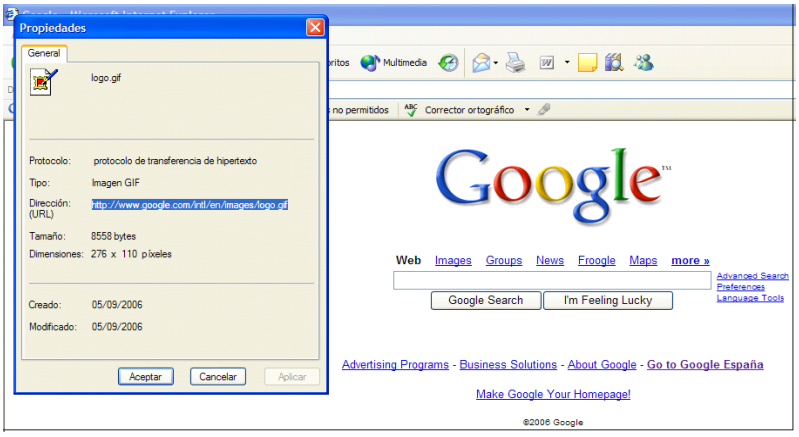

Figura 5.7 Cómo copiar la URL del logotipo de Google.

Si ahora se observa el contenido del directorio **"c:\Temp"**, en él se aprecia un fichero llamado **"Google.html"** y un directorio **"Google_archivos"**, que contiene la imagen que aparece en la página Web y que es referenciada por el primero. Como ya se comentó, al almacenar la página en el disco duro, todas las referencias a imágenes se han convertido en URLs locales, por lo que tendremos que convertirlas de nuevo en URLs universales. Para esto se abre el fichero **"Google.html"** con el navegador, y se selecciona la opción del menú **Ver->Código Fuente**; con esto se abre

una ventana del Bloc de Notas con el código HTML de la página. En este maremagnum de código, hay que sustituir en la línea donde pone:

```
<IMG height=110 alt=Google src="Google_archivos/logo.gif" width=276>
```

por

```
<IMG height=110 alt=Google
src="http://www.google.com/intl/en/images/logo.gif" width=276>
```

que es la URL de la imagen copiada en el paso anterior. Una vez hecho esto se selecciona **Archivo->Guardar** para almacenar los cambios.

Para que sea más fácil este proceso, usar en el Bloc de Notas la opción de menú **Editar->Buscar...**, y buscar la cadena **"src"**. En la figura 5.8 se muestra como debe quedar el fichero **"Google.html"**.

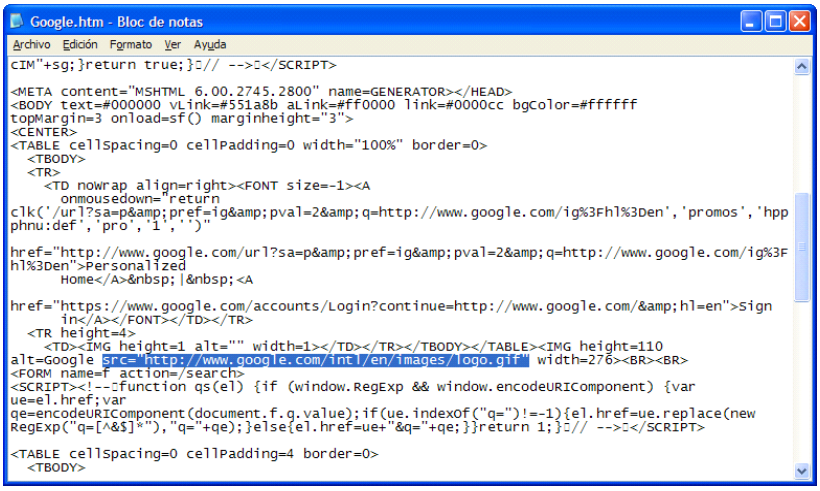

Figura 5.8 Modificación de la URL del logo de Google con el Bloc de Notas

Una vez realizados estos pasos, ya es posible proceder a construir el programa que envíe el mensaje de correo electrónico basado en HTML. El código es similar al del epígrafe 3.3.1 -en el que se enviaba un mensaje de texto plano- pero con algunas modificaciones. En este caso también se debe crear una sesión con las propiedades del sistema, habiendo añadido a estas propiedades la dirección del servidor SMTP que se va a usar, almacenándolo en la cadena smtpHost. Por tanto comenzamos importando las librerías necesarias y realizando estos pasos, que ya bien conocemos:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

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

```
if (args.length != 2) {
      System.out.println("Ha de enviar dos parámetros\n" +
           "java EnviarCorreo fromAddress toAddress");
      System.exit(1);
}
String from = args [0];
String to           = args [1];
String smtpHost  = "smtp.ono.com";
Properties props = System.getProperties();
props.put("mail.smtp.host",smtpHost);
Session sesion = Session.getDefaultInstance(props,null);
```

El paquete java.io se ha importado porque son necesarias las clases BufferedReader, FileReader y la clase IOException con objeto de leer el contenido del fichero **"Google.html"**.

El siguiente paso, que también es igual al de enviar un correo de texto plano, será crear un mensaje vacío y rellenar su asunto, el emisor y el receptor.

```
try {
     Message mensaje = new MimeMessage(sesion);
     mensaje.setSubject("Google en HTML");
     mensaje.setFrom(new InternetAddress(from));
     mensaje.addRecipient(Message.RecipientType.TO,
                          new InternetAddress(to));
```

Lo siguiente consiste en rellenar el contenido del mensaje pero, a diferencia de cuando se envía un correo de texto plano, no debe usarse el método setText() pasándole una cadena como contenido del mensaje, sino que el contenido del mensaje será el fichero HTML que antes se guardó; el método que hay que utilizar ahora es setContent() pues es necesario indicar que el mensaje contiene código HTML que debe interpretar el destinatario. En cualquier caso, para leer el fichero **"Google.html"** deben usarse las clases BufferedReader y FileReader.

La clase FileReader se usa para leer un flujo de caracteres desde un fichero, el cual es pasado como parámetro a su constructor, indicando su ruta y nombre. Si el constructor de esta clase falla porque no se encuentra el fichero indicado o porque no se puede abrir, se lanza una excepción de tipo FileNotFoundException que hereda de IOException; en caso contrario se devuelve un canal de entrada de caracteres que será, a su vez, el parámetro de entrada del constructor de la clase BufferedReader, que constituye un estrato que proporciona funcionalidades de *buffer* cuando la entrada/salida se produce desde un dispositivo de almacenamiento lento.

De la clase BufferedReader usaremos el método readLine() para leer líneas del flujo de caracteres de entrada devuelto por el FileReader. Este método lee línea por línea del flujo de entrada y devuelve cada una en forma de String, devolviendo null cuando llegue al final del flujo, es decir, cuando se haya alcanzado el final del

fichero. Este método puede lanzar una excepción de tipo IOException si se da un error de entrada/salida; esta excepción debe ser capturada y tratada por el programa.

Las líneas que se van leyendo del fichero (que, como ya se habrá adivinado, es **"c:/Temp/Google.html"**) se van concatenando en una cadena denominada fichero, hasta que el BufferedReader devuelva null, es decir, hasta que se haya leído el fichero completo:

```
String fichero = "";
String linea;
BufferedReader br = new BufferedReader (
                    new FileReader ("c:/Temp/Google.htm"));
while ((linea = br.readLine()) != null)
        fichero += linea;
br.close();
```

El último paso consiste en rellenar el mensaje con el contenido de este fichero. Para ello se usa el método setContent() de la clase Message (lo hereda de la interfaz Part). Este método recibe un primer parámetro de tipo Object que constituye el contenido del mensaje, en nuestro caso la cadena fichero. Como segundo parámetro hay que indicarle a través de un String de qué tipo MIME es el contenido que le hemos pasado como primer parámetro; en este caso se trata de texto en HTML que en MIME se corresponde con *text/html*:

```
mensaje.setContent(fichero,"text/html");
```

Por último, se envía el mensaje ya compuesto usando el método send() de la clase Transport:

```
Transport.send(mensaje);
```

debiéndose capturar todas las excepciones necesarias.

5.2.2 Ejemplo completo

Enviar un correo basado en HTML sin las imágenes incluidas en el propio mensaje, sino sólo referenciadas, tiene una serie de ventajas e inconvenientes. El mensaje ocupa menos espacio, pues la imagen no está contenida en el propio mensaje, lo que implica que se satura menos al servidor, el mensaje tarda menos en enviarse y además el código Java que envía el mensaje es bastante simple. El inconveniente es que la URL a la que referencia la imagen ha de estar disponible, o de lo contrario el destinatario no podrá verla cuando abra el mensaje.

A continuación se muestra el código completo:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
```

```
public class EnviarCorreoHTMLURL {
   public static void main (String [] args) {
      // Comprobar que el número de argumentos es el correcto
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                            "java EnviarCorreo fromAddress toAddress");
         System.exit(1);
      }
   // Obtener el from y el to recibidos como parámetros
      String from = args [0];
      String to    = args [1];
   // Obtener las propiedades del sistema y establecer el
   // servidor SMTP que vamos a usar
      String smtpHost  = "smtp.ono.com";
      Properties props = System.getProperties();
      props.put("mail.smtp.host",smtpHost);

   // Obtener una sesión con las propiedades anteriormente /definidas
      Session sesion = Session.getDefaultInstance(props,null);
      //Capturamos las excepciones
      try {
      // Crear un mensaje vacío
         Message mensaje = new MimeMessage(sesion);
      // Rellenar los atributos y el contenido
      // Asunto
         mensaje.setSubject("Google en HTML");
      // Emisor del mensaje
         mensaje.setFrom(new InternetAddress(from));
      //Receptor del mensaje
         mensaje.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));
      // Leer el fichero HTML
         String fichero = "";
         String linea;
      // Leer el fichero HTML
         String fichero = "";
         String linea;
         BufferedReader br = new BufferedReader (
                              new FileReader("c:/Temp/Google.htm"));
         while ((linea = br.readLine()) != null)
            fichero += linea;
         br.close();
      // Rellenar el mensaje con el fichero e indicar que se
      // trata de un fichero HTML
         mensaje.setContent(fichero,"text/html");
      // Enviar el mensaje
         Transport.send(mensaje);
      } catch (MessagingException me) {
```

```
                System.err.println(me.getMessage());
            } catch (IOException ioe) {
                System.err.println(ioe.getMessage());
            }
        }
    }
```

5.2.3 Mensaje HTML con imágenes integradas

En este epígrafe vamos a estudiar cómo enviar un mensaje basado en HTML pero, a diferencia del ejemplo anterior, en éste la imagen estará incrustada en el propio mensaje. Esta solución es un poco más compleja y sobrecarga más a los servidores, sobre todo en el caso de imágenes de gran tamaño, ya que éstas ocupan mucho espacio en disco, y hacen que los mensajes sean m´´as voluminosos y tarden más en ser enviados. Como ventaja, la imagen estará siempre accesible en el mensaje y por tanto siempre será visible por el destinatario.

El ejemplo con el que se va a trabajar es el mismo que en el epígrafe anterior. Así pues, los pasos básicos en este ejemplo consisten en crear dos objetos BodyPart en los que insertar el texto y la imagen respectivamente; luego se crea un objeto Multipart al que se añaden estos dos objetos y éste, a su vez, se utiliza para rellenar el contenido del mensaje. Ver figura 5.5.

Además, es necesario hacer que el fichero HTML referencie a una figura adjunta en lugar de una URL externa; para ello hay que modificar el código HTML del ejemplo anterior. Bäsicamente, es necesario cambiar la línea:

```
<IMG height=110 alt=Google
src="http://www.google.com/intl/en/images/logo.gif " width=276>
```

por:

```
<IMG height=110 alt=Google src="cid:figura1" width=276>
```

De esta manera, en vez de hacer que la imagen referencie a una URL, ésta referencia a una cabecera que debe incluirse en el mensaje cuando le añada la imagen. En la figura 5.9 puede verse cómo debe quedar el fichero **“Google.html”**.

Ahora comenzaremos a desarrollar el código de la aplicación. El inicio es el mismo que en las aplicaciones anteriores: obtener las propiedades del sistema, añadir el servidor SMTP a usar para enviar el mensaje y obtener una sesión con dichas propiedades:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
public class EnviarCorreoHTMLEmbebido {
    public static void main (String [] args) {
        if (args.length != 2) {
```

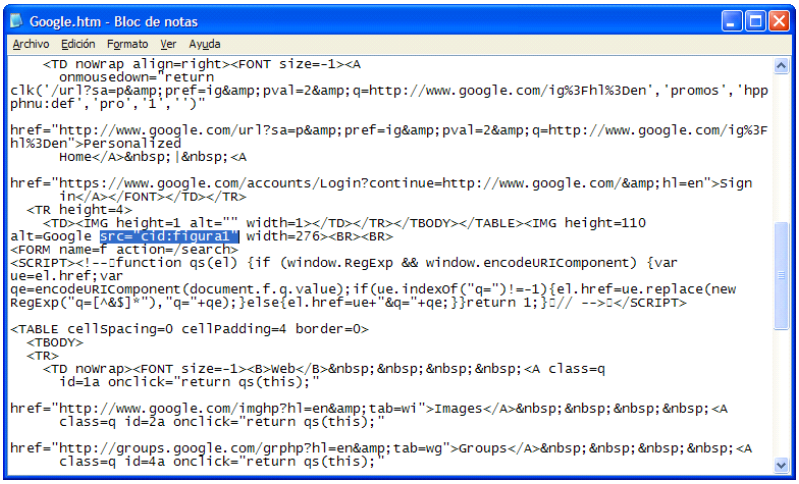

Figura 5.9 Modificación de **"Google.html"** para referenciar a un fichero adjunto en el mensaje

```
            System.out.println("Ha de enviar dos parámetros\n" +
                               "java EnviarCorreo fromAddress toAddress");
            System.exit(1);
        }
        String from = args [0];
        String to           = args [1];
        String smtpHost  = "smtp.ono.com";
        Properties props = System.getProperties();
        props.put("mail.smtp.host",smtpHost);
        Session sesion = Session.getDefaultInstance(props,null);
```

El paquete java.io se importa para usar las clases IOException, BufferedReader y FileReader. En el epígrafe anterior se vio la utilidad de estas clases.

El siguiente paso también es igual que en los ejemplos anteriores: crear un mensaje vacío y rellenar sus campos *asunto*, *emisor* y *receptor* del mensaje:

```
try {
    Message mensaje = new MimeMessage(sesion);
    mensaje.setSubject("Mensaje Google completo");
    mensaje.setFrom(new InternetAddress(from));
    mensaje.addRecipient(Message.RecipientType.TO,
                         new InternetAddress(to));
```

Y ahora llega lo interesante. El objetivo es crear un objeto Multipart que, como se comentó anteriormente, se usará para establecer el contenido del mensaje, habiendo insertado previamente en él la imagen del fichero HTML y el contenido de éste. Al constructor de este objeto le pasaremos la cadena **"related"** para indicar que

este MultiPart será de tipo *multipart/related*. Si quisiéramos que fuese de tipo *multipart/alternative* le pasaríamos **"alternative"**, y para obtener el tipo *multipart/mixed* se usa el mismo constructor pero sin parámetros. Como se comentó en el epígrafe 1.5.4 este tipo Multipart se usa cuando las partes del mensaje están relacionadas entre si, que es el caso que se da en este ejemplo entre la imagen del mensaje y su texto:

```
Multipart multipart = new MimeMultipart ("related");
```

Ahora es necesario crear objetos de tipo BodyPart con los que rellenar el Multipart. Para empezar, leeremos el contenido del fichero HTML y lo insertaremos en el primer objeto BodyPart, indicando que se trata de texto de tipo HTML, lo que en MIME se establece como **"text/html"**. Este proceso de leer el fichero es igual al del ejemplo anterior (apartado 5.2.1):

```
String fichero = "";
String linea;
BufferedReader br = new BufferedReader (
                            new FileReader ("c:/Temp/google.htm"));
while ((linea = br.readLine()) != null)
    fichero += linea;
br.close();
BodyPart texto = new MimeBodyPart ();
texto.setContent(fichero,"text/html");
```

Este BodyPart se debe insertar ahora en el Multipart, con lo que ya está listo el texto del fichero HTML:

```
multipart.addBodyPart(texto);
```

Ahora sólo queda incluir la imagen.

Para incluir la imagen creamos otro objeto, esta vez de tipo MimeBodyPart, pues su contenido es más complejo que simple texto. De hecho, esta clase incorpora el método attachFile(), cuya utilización facilitará enormemente nuestro propósito. Este método recibe como parámetro un String que se corresponde con la ruta completa y nombre del fichero que queremos añadir. Por último, debe especificarse que el campo *Content-ID* de la cabecera contendrá el valor **"<figura1>"**con lo que se indica que la imagen que aparece en el fichero HTML (que modificamos y pusimos como **src="cid:figura1"**), se corresponde con la que se ha incluído en esta MimeBodyPart. El nombre en sí de **"figura1"** no es significativo, lo importante es que tanto el fichero HTML como campo *Content-ID* hagan referencia al mismo nombre:

```
MimeBodyPart imagen = new MimeBodyPart();
imagen.attachFile("c:/Temp/Google_archivos/logo.gif");
imagen.setHeader("Content-ID","<figura1>");
multipart.addBodyPart(imagen);
```

El método attachFile() de la clase MimeBodyPart sólo está disponible a partir de la versión 1.4 de JavaMail, por lo que si se trabaja con una versión anterior no lo podremos emplear. En tal caso se debe sustituir la línea:

```
imagen.attachFile("c:/Temp/Google_archivos/logo.gif")
```

por:

```
DataSource fds = new FileDataSource("c:/Temp/Google_archivos/logo.gif");
imagen.setDataHandler(new DataHandler(fds));
```

en cuyo caso, además, habría que importar también el paquete javax.activation.

Para terminar el programa solo nos quedaría rellenar el mensaje con el Multipart que hemos creado y enviarlo.

```
            mensaje.setContent(multipart);
            Transport.send(mensaje);
        } catch (MessagingException e) {
            System.err.println(e.getMessage());
        } catch (IOException ioe) {
            System.err.println(ioe.toString());
        }
```

5.2.4 Ejemplo completo

A continuación se muestra el código completo de la aplicación que envía un mensaje HTML con una imagen incrustada.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class EnviarCorreoHTMLEmbebido {
   public static void main (String [] args) {
   // Comprobar que el número de argumentos es el correcto
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                                 "java EnviarCorreo fromAddress toAddress");
         System.exit(1);
      }
   // Obtener el from y el to recibidos como parámetros
      String from = args [0];
      String to    = args [1];
   // Obtener las propiedades del sistema y establecer el servidor
   // SMTP que vamos a usar
      String smtpHost  = "smtp.ono.com";
      Properties props = System.getProperties();
      props.put("mail.smtp.host",smtpHost);
   // Obtener una sesión con las propiedades anteriormente definidas
```

```
        Session sesion = Session.getDefaultInstance(props,null);
    // Capturar las excepciones
        try {
            Message mensaje = new MimeMessage(sesion);
        // Rellenar los atributos y el contenido
        // Asunto
            mensaje.setSubject("Mensaje Google Completo");
        // Emisor del mensaje
            mensaje.setFrom(new InternetAddress(from));
        // Receptor del mensaje
            mensaje.addRecipient(Message.RecipientType.TO,
                                 new InternetAddress(to));
        // Crear un Multipart de tipo multipart/related
            Multipart multipart = new MimeMultipart ("related");
        // Leer el fichero HTML
            String fichero = "";
            String linea;
            BufferedReader br = new BufferedReader (
                                    new FileReader ("c:/Temp/google.htm"));
            while ((linea = br.readLine()) != null)
               fichero += linea;
            br.close();
        // Rellenar el MimeBodyPart con el fichero e indicar que es un fichero HTML
            BodyPart texto = new MimeBodyPart ();
            texto.setContent(fichero,"text/html");
            multipart.addBodyPart(texto);
        // Procesar la imagen
            MimeBodyPart imagen = new MimeBodyPart();
            imagen.attachFile("c:/Temp/Google_archivos/logo.gif");
            imagen.setHeader("Content-ID","<figura1>");
            multipart.addBodyPart(imagen);
            mensaje.setContent(multipart);
        // Enviar el mensaje
            Transport.send(mensaje);
        } catch (MessagingException e) {
            System.err.println(e.getMessage());
        } catch (IOException ioe) {
            System.err.println(ioe.toString());
        }
    }
}
```

5.3 Mensajes con ficheros adjuntos

En este apartado estudiaremos cómo enviar y recibir mensajes que poseen ficheros adjuntos. La diferencia con los ejemplos anteriores estriba en que, en este

caso, el adjunto no está referenciado de manera explícita por el mensaje principal, sino que debe ser descargado explícitamente al sistema de ficheros del destinatario.

5.3.1 Envío de mensaje con ficheros adjuntos

En este ejemplo se muestra al usuario cómo enviar un mensaje de correo de texto plano que contiene dos adjuntos. La dinámica de este proceso es parecida a la de los epígrafes precedentes, siendo los pasos iniciales: obtener las propiedades del sistema, establecer el servidor SMTP a usar, crear un mensaje vacío y rellenar sus campos *remitente*, *destinatario* y *asunto*. A continuación se construye un BodyPart por cada adjunto y otro también para el texto del mensaje. Una vez listos estos BodyPart se añaden a un objeto Multipart, se rellena con éste el contenido del mensaje y se envía.

Los dos adjuntos que se usan en este ejemplo están almacenados en el directorio **c:/Temp**, y se llaman **adjunto.txt** y **logo.gif**. Obviamente se pueden usar cualesquiera otros ficheros.

Comenzamos importando los paquetes necesarios. Se importa el paquete de entrada/salida java.io para poder usar la clase File y por el método attachFile() de la clase MimeBodyPart, que puede lanzar la excepción IOException. El paquete javax.activation es necesario para poder usar las clases DataSource, FileDataSource y DataHandler cuyo funcionamiento ya se ha comentado en epígrafes anteriores pero se explicará mejor más adelante:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
import javax.activation.*;
```

Como siempre, se obtienen los parámetros de entrada, comprobando que el número de éstos es correcto: direcciones de correo de *from* y de *to*. Se obtienen las propiedades del sistema y les añadimos el servidor SMTP a usar. Por último, se crea una sesión utilizando estas propiedades:

```
public class EnviarCorreoAdjuntos {
    public static void main (String [] args) {
    // Comprobar que el número de argumentos es el correcto
        if (args.length != 2) {
            System.out.println("Ha de enviar dos parámetros\n" +
                                    "java EnviarCorreoAdjuntos fromAddress
toAddress");
            System.exit(1);
        }
    // Obtener el from y el to recibidos como parámetros
        String from = args [0];
```

```
        String to = args [1];
    // Obtener las propiedades del sistema y establecer el servidor
    // SMTP que se va a usar
        String smtpHost  = "smtp.ono.com";
        Properties props = System.getProperties();
        props.put("mail.smtp.host",smtpHost);
    // Obtener una sesión con las propiedades anteriormente definidas
        Session sesion = Session.getDefaultInstance(props,null);
```

Se crea ahora un mensaje vacío y se rellena su *asunto*, el *emisor* y el *receptor*:

```
//Capturar las excepciones
try {

    Message mensaje = new MimeMessage(sesion);
    mensaje.setSubject("Hola Mundo con adjuntos");
    mensaje.setFrom(new InternetAddress(from));
    mensaje.addRecipient(Message.RecipientType.TO,
                         new InternetAddress(to));
```

Lo siguiente que hay que hacer es crear un objeto Multipart. En éste deben insertarse tres BodyPart que se corresponden con el texto del mensaje y con los dos adjuntos respectivamente. Al usar el constructor de la clase MimeMultipart sin argumentos se obtiene directamente un objeto MimeMultipart vacío y con el tipo MIME *multipart/mixed*; este tipo MIME se usa cuando las partes del mensaje son independientes pero están agrupadas en un orden, como es el caso de este ejemplo:

```
Multipart mp = new MimeMultipart();
```

Ahora se crea el BodyPart que contiene el texto del mensaje. Para ello es suficiente con crear un BodyPart vacío y usar el método setText() de la interfaz Part (implementada por la clase Message) para añadir el texto del mensaje. Este método añade el texto como tipo *text/plain* (texto plano) sin necesidad de tener que especificarlo. Una vez hecho esto se añade dicho BodyPart al Multipart:

```
BodyPart texto = new MimeBodyPart ();
texto.setText ("Este es el cuerpo del mensaje");
mp.addBodyPart (texto);
```

Si en vez de texto plano hubiésemos querido enviar texto en formato HTML sólo tendríamos que realizar los mismos pasos que en los ejemplos 5.2.1 y 5.2.3, leer el contenido del fichero y añadirlo al BodyPart usando el método setContent().

Ahora procederemos a añadir el primer adjunto al mensaje, para lo cual se crea un objeto MimeBodyPart en vez de BodyPart porque necesitamos el método attachFile() que pertenece a esta clase. Como ya se ha visto, este método permite añadir un fichero al MimeBodyPart indicando sólo su ruta y nombre en forma de String. Una vez adjunto el fichero al BodyPart añadimos éste en el Multipart:

```
MimeBodyPart adjunto1 = new MimeBodyPart ();
adjunto1.attachFile("c:/Temp/adjunto.txt");
```

```
mp.addBodyPart(adjunto1);
```

Ahora se añade el segundo adjunto al mensaje. No obstante, como el método attachFile() sólo está disponible a partir de la versión 1.4 de JavaMail, y si se emplea una versión anterior no podremos usarlo, vamos a estudiar la alternativa que se usaba en versiones anteriores. Para ello se crea otro BodyPart y un objeto de la clase File, indicándole la ruta completa y el nombre del fichero a adjuntar. Un objeto File no es más que un modelo de abstracción de un fichero, que facilita las operaciones con él:

```
BodyPart adjunto2 = new MimeBodyPart ();
File path = new File ("c:/Temp/logo.gif");
```

A continuación se crea un objeto de la clase DataSource (fuente de datos) que proporciona mecanismos de acceso sobre un conjunto de datos de cualquier tipo, en este caso el fichero adjunto. Como esta clase es abstracta, se debe utilizar una instancia de la clase FileDataSource, que hereda de ella y que provee servicios de clasificación de datos. El constructor de esta clase recibe como parámetro el objeto File creado anteriormente. Finalmente queda añadir el adjunto al BodyPart, para lo cual usamos el método setDataHandler() de la interfaz Part, que simplemente añade al objeto que lo llame, el DataHandler (manejador de datos) que se le pase como parámetro, y que será quien se encargue de operar sobre los datos y añadirlos en el BodyPart. El constructor de DataHandler recibe como parámetro la fuente de datos sobre la que tiene que operar, en este caso el DataSource creado:

```
DataSource source = new FileDataSource(path);
adjunto2.setDataHandler(new DataHandler(source));
```

Por último, no se nos debe olvidar indicar el nombre del adjunto usando, para ello, el método setFileName() de la interfaz Part. A este método se le pasa como parámetro el nombre del fichero usando el método getName() de la clase File, que devuelve el nombre del fichero sin la ruta. Con esto el adjunto ya está insertado correctamente en el BodyPart, y sólo queda añadir éste al Multipart.

```
adjunto2.setFileName(path.getName());
mp.addBodyPart(adjunto2);
```

Una vez completado el Multipart, ya se puede rellenar el contenido del mensaje usando el método setContent(), y enviar el mensaje:

```
mensaje.setContent(mp);
//Enviamos el mensaje
Transport.send(mensaje);
```

5.3.2 Ejemplo completo

A continuación se muestra el código completo de este programa que envía un mensaje de texto plano con dos adjuntos.

```
import java.util.Properties;
import javax.mail.*;
```

```
import javax.mail.internet.*;
import java.io.*;
import javax.activation.*;

public class EnviarCorreoAdjuntos {
   public static void main (String [] args) {
   // Comprobar que el número de argumentos es el correcto
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                  "java EnviarCorreoAdjuntos fromAddress toAddress");
         System.exit(1);
      }
   // Obtener el from y el to recibidos como parámetros
      String from = args [0];
      String to   = args [1];
   // Obtener las propiedades del sistema y establecer el servidor
   // SMTP que se va a usar
      String smtpHost  = "smtp.ono.com";
      Properties props = System.getProperties();
      props.put("mail.smtp.host",smtpHost);
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
   // Capturar las excepciones
      try {
         Message mensaje = new MimeMessage(sesion);
      // Rellenar los atributos y el contenido
      // Asunto
         mensaje.setSubject("Hola Mundo con adjuntos");
      // Emisor del mensaje
         mensaje.setFrom(new InternetAddress(from));
      // Receptor del mensaje
         mensaje.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));
         Multipart mp = new MimeMultipart();
      // Texto del mensaje
         BodyPart texto = new MimeBodyPart ();
         texto.setText ("Este es el cuerpo del mensaje");
         mp.addBodyPart (texto);
      // Adjuntar el primer fichero
         MimeBodyPart adjunto1 = new MimeBodyPart ();
         adjunto1.attachFile("c:/Temp/adjunto.txt");
         mp.addBodyPart(adjunto1);
      // Adjuntar el segundo fichero
         BodyPart adjunto2 = new MimeBodyPart ();
         File path = new File ("c:/Temp/logo.gif");
         DataSource source = new FileDataSource(path);
         adjunto2.setDataHandler(new DataHandler(source));
```

```
            // Nombre de fichero que aparecerá en el adjunto
               adjunto2.setFileName(path.getName());
               mp.addBodyPart(adjunto2);
               mensaje.setContent(mp);
            // Enviar el mensaje
               Transport.send(mensaje);
            } catch (MessagingException e) {
               System.err.println(e.getMessage());
            } catch (IOException ioe) {
               System.err.println(ioe.toString());
            }
         }
      }
```

5.3.3 Recepción de mensaje con ficheros adjuntos

En este apartado se va a estudiar cómo descargar mensajes de texto plano que contengan adjuntos, o bien mensajes basados en HTML con una imagen inrustada en el mensaje o bien con código HTML únicamente; con ello, se cubren todas las posibilidades de envío de mensajes vistas en los apartados anteriores.

El primer paso consiste en averiguar si el mensaje es de texto, ya sea texto plano o texto en formato HTML, o si se trata de un mensaje con adjuntos o imágenes incrustadas en el propio mensaje. En este último caso el mensaje estará formado por un objeto Multipart que contendrá varios BodyPart, que habrá que ir procesando adecuadamente. Ver figura 5.5.

Este ejemplo es parecido del epígrafe 3.3, aunque en este caso el código se vuelve un poco más complejo, pues trata más casos que cuando sólo se reciben mensajes de texto plano. Así pues, el comienzo del código es similar al del citado epígrafe, por lo que no lo comentaremos en profundidad, si bien tiene dos modificaciones. La primera consiste en la necesidad de importar el paquete javax.activation, necesario para almacenar el mensaje en disco cuando se trate de un mensaje HTML sin imágenes incluidas en el propio mensaje, es decir, sólo texto en formato HTML. Más adelante se verá su modo de empleo. La otra modificación es que, por motivos de claridad, se procesará exclusivamente el primer mensaje, es decir, el mensaje que lleva más tiempo en la bandeja de entrada. Comenzamos importando los paquetes necesarios:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
import javax.activation.*;
```

A continuación se comprueba que el número de argumentos es el correcto, se obtienen y se recuperan también las propiedades del sistema, creándose una sesión a partir de ellas:

```
public class ObtenerCorreoAdjuntos {
    public static void main (String [] args) {
        if (args.length != 2) {
            System.out.println("Ha de enviar dos parámetros\n" +
                               "java ObtenerCorreoAdjuntos usuario clave");
            System.exit(1);
        }
        String usuario = args [0];
        String clave   = args [1];
        String popHost   = "pop.runbox.com";
        Properties props = System.getProperties();
        Session sesion = Session.getDefaultInstance(props,null);
```

Lo siguiente será crear un objeto Store con el que acceder al buzón de correo, indicando el protocolo de acceso (POP3 en este ejemplo), y conectarse a él, indicando el nombre del servidor POP, y el nombre de usuario y contraseña:

```
try {
        Store store = sesion.getStore("pop3");
        store.connect(popHost,usuario,clave);
```

Una vez conectados al servidor, ya se puede abrir la carpeta **INBOX** en modo de sólo lectura y, como ya se ha comentado, obtener el primer mensaje:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message mensaje = folder.getMessage(1);
```

El contenido del mensaje se recupera usando el método getContent() (que puede lanzar una excepción IOException, lo que nos obligó a importar el paquete java.io):

```
Object o = mensaje.getContent();
```

Ahora hay que ver qué tipo de mensaje es el que se ha recuperado, para lo cual puede hacerse uso de la palabra clave de Java **instanceof**, que dice si un objeto es una instancia de un clase concreta o no. En el caso de que así sea devuelve **true**. Si el mensaje es de texto, ya sea texto plano o texto en formato HTML, sin imágenes contenidas en él, su contenido será de tipo String:

```
if (o instanceof String) {
```

Una vez visto si el mensaje es de texto, hemos de comprobar de cual de los dos tipos de texto citados anteriormente se trata; para ello se usa el método getContentType(), que pertenece a la interfaz Part y es implementado por la clase Message (en este caso nuestro mensaje de correo es de este tipo). Con este método se obtiene una cadena con el tipo del contenido del mensaje. Esta cadena además de indicar el tipo de texto, muestra información sobre el conjunto de caracteres (por

ejemplo *text/html; charset=Cp1252*) y como en este caso esta información no nos interesa, usaremos el método indexOf() para ver si esa cadena contiene a la cadena **"text/html"**, sin atender al resto de la información. indexOf() devolverá **-1** si **"text/html"** no estuviese contenida en la cadena devuelta por getContentType(), en cuyo caso deduciremos que se trata de un mensaje de texto, pero no HTML, por lo que debe ser tratado como texto plano:

```
if (mensaje.getContentType().indexOf("text/html") != -1) {
```

Para tratar los mensajes de texto en formato HTML se debe usar el método getDataHandler(), que devuelve un objeto de la clase DataHandler asociado al contenido de este mensaje. Este método pertenece a la interfaz Part que es implementada por la clase Message. La clase DataHandler permite operar con el contenido del mensaje y, en este ejemplo, permitirá escribir el contenido del mensaje en un canal de salida, usando el método writeTo(OutputStream os), que puede lanzar la excepción IOException. Para obtener este OutputStream con asociado a un fichero en el que escribir, se debe usar la clase FileOutputStream. Esta clase se usa cuando se desea escribir un flujo de datos en un fichero, que es lo que haremos en este ejemplo. En este caso el fichero está almacenado en el directorio **"c:/Temp"**, y se llama **"contenido.html"**:

```
DataHandler dh = mensaje.getDataHandler();
OutputStream os = new FileOutputStream ("c:/Temp/contenido.html");
dh.writeTo(os);
os.close();
```

Si el mensaje es de texto, pero no se trata de texto en formato HTML, entonces debe ser texto plano, por lo que directamente se visualizará por pantalla:

```
else
        System.out.println (o);
```

Si el mensaje que estamos tratando no es un mensaje de texto, entonces debe comprobarse si se trata de un mensaje de tipo Multipart. Si es así, el mensaje será un mensaje con adjuntos (uno o varios) o un mensaje HTML con imágenes (una o varias) incluidas en el mensaje. En este caso se obtiene el número de partes que componen el mensaje y, una vez averiguado de qué tipo es cada una, se procesan convenientemente. Como ya se sabe que el mensaje es de tipo Multipart, se le puede hacer un *casting* al contenido del mensaje para obtener el objeto Multipart y, usando su método getCount(), es posible conocer el número de partes (objetos BodyPart) que componen el mensaje a procesar:

```
else if (o instanceof Multipart) {
        Multipart mp = (Multipart)o;
        int numPart = mp.getCount();
        for (int i=0; i < numPart; i++) {
```

Cada uno de los BodyPart que componen el mensaje se pueden ir recuperando mediante el método getBodyPart() de la clase Multipart. Este método irá devolviendo estos objetos en función del índice del bucle, ya que están

almacenados dentro del objeto Multipart cada uno en una posición, como si fuese un *array*:

```
Part part = mp.getBodyPart(i);
```

Ahora debe conocerse la disposición o naturaleza del BodyPart recuperado,para lo que se usa el método getDisposition() de la interfaz Part. La disposición consiste en un String que informa de cómo debe presentarse al usuario el objeto de tipo Part, es decir, sirve para diferenciar entre el contenido en sí del mensaje, los adjuntos y las imágenes. Esta cadena puede valer null, en cuyo caso este BodyPart porta el contenido del mensaje, o ser distinta de null e igual (ignorando mayúsculas y minúsculas) a la cadena ATTACHMENT (se trata de un adjunto) o a la cadena INLINE (se trata de una imagen incrustada en el mensaje). Estas cadenas están presentes como campos de la interfaz Part:

```
String disposition = part.getDisposition();
```

Si la disposición vale null se estará procesando un BodyPart que tiene el contenido del mensaje. En este ejemplo se contemplan tres posibles casos para tratar el contenido cuando la disposición vale null:

- *multipart/alternative*
- *text/plain*
- *text/html*

Para hacer estas diferenciaciones, se obtiene el tipo MIME del BodyPart y se comprueba si es de tipo *multipart/alternative* o *text/plain*, usando el método isMimeType() de la interfaz Part, que informa de si el BodyPart se corresponde o no con el tipo MIME que se le pasa como parámetro en forma de String:

```
if (disposition == null) {
        if(part.isMimeType("multipart/alternative")||
                part.isMimeType("text/plain")) {
```

En este ejemplo hemos tenido que contemplar el tipo MIME *multipart/alternative* (versiones alternativas de la misma información) porque algunos servidores de correo establecen de este tipo el contenido de los mensajes con adjuntos cuando los envían. Cuando se trata con un BodyPart que se corresponde con este tipo, estamos tratando en realidad con un Multipart que contiene varios BodyPart con la misma información, presentada de distintas formas. Por tanto hay que hacer un *casting* al contenido del BodyPart para convertirlo en MultiPart y extraer el primer BodyPart que lo compone (y posteriormente su contenido), es decir, la primera versión de la información (generalmente en texto plano, que es el formato que se desea obtener) y almacenar su contenido en una cadena para visualizarlo a continuación por pantalla:

```
String cuerpoMensaje;
if (part.isMimeType("multipart/alternative")) {
        Multipart mp2 = (Multipart) part.getContent();
        Part part2    = mp2.getBodyPart(0);
        cuerpoMensaje = (String)part2.getContent();
```

```
}
```

En caso de que la disposición sea null y el tipo MIME *text/plain* sólo hay que almacenar su contenido en un String. Y, tanto si se trata de *multipart/alternative* o de *text/plain*, el contenido del mensaje debe visualizarse por pantalla:

```
else
        cuerpoMensaje = (String)part.getContent();
        System.out.println(cuerpoMensaje);
```

El último caso a contemplar cuando la disposición es null es cuando el contenido del mensaje est en formato HTML. En este caso, al ser un mensaje multiparte, estará formado por el texto de un mensaje HTML junto con imágenes incrustadas. Al igual que hicimos cuando se trataba de un mensaje de texto con formato HTML hay que asegurarse de que el tipo MIME es *text/html*. Si es así, se convierte el objeto Part en un objeto MimeBodyPart para poder usar su método saveFile(), que se encarga de salvar su contenido en disco, en el fichero que se le indique como parámetro:

```
else {
        if (part.getContentType().indexOf("text/html") != -1) {
                MimeBodyPart mbp = (MimeBodyPart)part;
                mbp.saveFile("c:/Temp/contenido.html");
        }
}
```

El método saveFile() solo está disponible a partir de la versión 1.4 de JavaMail, por lo que si se usa una versión anterior hay que sustituir el código anterior por las siguientes líneas (el *casting* anterior tampoco sería ya necesario):

```
DataHandler dh = part.getDataHandler();
OutputStream os = new FileOutputStream ("c:/Temp/contenido.html");
dh.writeTo (os);
os.close();
```

Si la disposición de esta parte del mensaje es distinta de null, es que no se trata del cuerpo principal, por lo que hay que comprobar si se está frente a un adjunto o ante una imagen embebida en el mensaje. En cualquiera de los dos casos se puede usar el método getFileName() de la interfaz Part para obtener su nombre y grabarlo; si no tiene nombre lo llamaremos **"adjunto"** + **posición** que ocupa este BodyPart dentro del Multipart, usando para ello el índice del bucle:

```
else if ((disposition != null) &&
        (disposition.equalsIgnoreCase(Part.ATTACHMENT)||
         disposition.equalsIgnoreCase(Part.INLINE)))) {
        String nombrePart = part.getFileName();
        if (nombrePart == null)
                nombrePart = "adjunto" + i;
```

y, al igual que se hacía antes, convertimos el objeto Part en un objeto MimeBodyPart para poder usar el método saveFile() de esta clase:

```
                    MimeBodyPart mbp = (MimeBodyPart)part;
                    mbp.saveFile("c:/Temp/" + nombrePart);
```

Por último, no debe olvidarse cerrar tanto la carpeta como el almacén:

```
folder.close(false);
store.close();
```

5.3.4 Ejemplo completo

A continuación se muestra el código completo de este ejemplo .

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
import javax.activation.*;

public class ObtenerCorreoAdjuntos {
   public static void main (String [] args) {
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                            "java ObtenerCorreoAdjuntos usuario clave");
         System.exit(1);
      }
   // Obtener el usuario y la clave recibidos como parámetros
      String usuario = args [0];
      String clave   = args [1];
   // Obtener las propiedades del sistema
      String popHost   = "pop.correo.yahoo.es";
      Properties props = System.getProperties();
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
   // Capturar las excepciones
      try {
      // Crear un Store indicando el protocolo de acceso y nos
      // conectarse a él
         Store store = sesion.getStore("pop3");
         store.connect(popHost,usuario,clave);
      // Crear un Folder y abrimos la carpeta INBOX en modo SOLO
      //LECTURA
         Folder folder = store.getFolder("INBOX");
         folder.open(Folder.READ_ONLY);
      // Obtener el mensaje número 1 de los almacenados en el Folder
         Message mensaje = folder.getMessage(1);
      // Obtener el contenido del mensaje y vemos de que tipo es
         Object o = mensaje.getContent();
```

```
// Si el mensaje es de texto sin adjuntos, ya sea texto HTML o
// texto plano, el contenido será de tipo String
   if (o instanceof String) {
   // Si entra por aquí se trata de texto HTML
      if (mensaje.getContentType().indexOf("text/html") != -1) {
         DataHandler dh = mensaje.getDataHandler();
         OutputStream os =
                        new FileOutputStream("c:/Temp/contenido.html");
         dh.writeTo(os);
         os.close();
      } else
   // Si entra por aquí se trata de texto plano
         System.out.println (o);
   }
// Si entra por aquí es que se trata de un mensaje con adjunto/s o
// HTML con imágenes embebidas en el mensaje
   else if (o instanceof Multipart) {
   // Se sabe que el contenido del mensaje es de tipo Multipart,
   // así que se le hace un casting para obtener un objeto de este
   // tipo
      Multipart mp = (Multipart)o;
   // Recorrer todos los Part que componen el mensaje
      int numPart = mp.getCount();
      for (int i=0; i < numPart; i++) {
      // Con cada parte del mensaje hay que ver la disposición que
      // tiene
         Part part = mp.getBodyPart(i);
         String disposition = part.getDisposition();
      // Si la disposición es null probablemente se trate del
      // contenido en sí del mensaje
         if (disposition == null) {
            if (part.isMimeType("multipart/alternative")
                ||part.isMimeType("text/plain")) {
               String cuerpoMensaje;
               if (part.isMimeType("multipart/alternative")) {
                  Multipart mp2 = (Multipart) part.getContent();
                  Part part2    = mp2.getBodyPart(0);
                  cuerpoMensaje = (String)part2.getContent();
               }else
                  cuerpoMensaje = (String)part.getContent();
            }
         // Si entra por aquí se trataría de la parte del texto de
         // un correo HTML con imágenes embebidas
            else {
               if (part.getContentType().indexOf("text/html") != -1) {
                  MimeBodyPart mbp = (MimeBodyPart)part;
```

```
                        mbp.saveFile("c:/Temp/contenido.html");
                    }
                }
            }
            // Si entra por aquí es que se trata de un adjunto o de una
            // imagen embebida en el mensaje
            else if ((disposition != null) &&
                    (disposition.equalsIgnoreCase(Part.ATTACHMENT) ||
                    (disposition.equalsIgnoreCase(Part.INLINE)))) {
                String nombrePart = part.getFileName();
                if (nombrePart == null)
                    nombrePart = "adjunto" + i;
                // Procesar el adjunto o imagen
                MimeBodyPart mbp = (MimeBodyPart)part;
                mbp.saveFile("c:/Temp/" + nombrePart);
            }
        }
      }
      folder.close(false);
      store.close();
    } catch (MessagingException me) {
      System.err.println(me.toString());
    } catch (IOException ioe) {
      System.err.println(ioe.toString());
    }
  }
}
```

Capítulo 6

Seguridad

6.1 Visión general

En este capítulo nos centraremos en conceptos relacionados con la seguridad de la información en las comunicaciones. Como este campo abarca multitud de conceptos que se salen de nuestros objetivos principales, enfocaremos nuestra atención hacia dos casos de estudio concretos y qué soluciones proporciona JavaMail para abordarlos. Estos casos son:

- autenticación frente a un servidor para enviar correo a través de él, y
- obtener correo de un servidor mediante una conexión segura.

6.2 Autenticación frente a un servidor

En el epígrafe 3.2 se vio cómo enviar un mensaje de texto plano, usando para ello un servidor SMTP que no necesitaba autenticación. Sin embargo, actualmente algunos servidores no nos permiten enviar un correo sin más y nos exigen esta autenticación, esto es, antes de poder enviar el mensaje hay que demostrarle al servidor que somos usuarios registrados, y que disponemos de una cuenta abierta en él con su correspondiente nombre de usuario y contraseña.

En este apartado se estudiará cómo enviar un mensaje de correo a través de un servidor que sí requiere de esta autenticación. El ejemplo será muy similar al del epígrafe 3.2, por lo que sólo nos centraremos en lo que lo diferencia de éste y, finalmente, se mostrará el código completo que realiza la tarea.

6.2.1 Clases Authenticator y PasswordAuthentication

Para hacer las cosas bien, se van a definir dos variables de clase de tipo String llamadas **user** y **password**. Estas variables serán estáticas (static) pues serán usadas desde el método estático main(). El lector puede imaginar para qué sirven ambas:

```
private static String user,password;
```

Estas variables se inicializarán con los parámetros que recibe el método main() por lo que, en este ejemplo, es necesario controlar que en vez de dos parámetros como en el ejemplo 3.2, se recibirán cuatro:

```
if (args.length != 4) {
    System.out.println("Ha de enviar dos parámetros\n" +
                       "java EnviarCorreoAutenticacion " +
                       "fromAddress toAddress user password");
    System.exit(1);
}
String from = args[0];
String to = args[1];
user = args[2];
password = args[3];
```

Resulta imprescindible añadir a las propiedades del sistema una propiedad para indicarle al servidor que sí vamos a autenticarnos en él. Esto se hace añadiendo la siguiente línea en el código:

```
props.put("mail.smtp.auth", "true");
```

La principal diferencia entre enviar un correo sin autenticarse y haciéndolo es, además de indicarlo en las propiedades, que al obtener la sesión con el método getDefaultInstance() de la clase Session es necesario pasar como segundo parámetro un objeto de la clase Authenticator.

La clase Authenticator sirve para representar a objetos que proporcionan cualquier autenticación en una conexión de red. Para usarla hay que codificar una clase que herede de ella, pues Authenticator es abstracta; básicamente, el único método que hay que implementar es getPasswordAuthentication(), que es invocado automáticamente por el sistema para conocer la información de autenticación. Este método simplemente devolverá un objeto de la clase PasswordAuthentication que puede crearse de manera sencilla usando el nombre de usuario y la clave que recibimos como parámetros en el método main(). Esta clase es tan sólo un repositorio o lugar de almacenamiento, que se usa para almacenar un nombre de usuario junto a su clave, datos que se pasan a su constructor en el momento de la creación. En la tabla 6.1 podemos ver a modo de resumen esta clase.

Para realizar los pasos que se han explicado hay que crear una clase anidada,

Método	Descripción
PasswordAuthentication (String userName, String password)	Construye un objeto PasswordAuthentication con el nombre de usuario y clave pasados como parámetro
String getUserName()	Devuelve el nombre de usuario almacenado en el objeto
String getPassword()	Devuelve la clave almacenada en el objeto

Tabla 6.1 Métodos principales de la clase PasswordAuthentication.

llamada MiAutenticador, dentro de la clase principal que envía el correo. Esta clase será la que extenderá a Authenticator y devolverá el objeto PasswordAuthentication. La clase debe ser estática pues es llamada desde el método estático main():

```
private static class MiAutenticador extends Authenticator {
        public PasswordAuthentication getPasswordAuthentication() {
                return new PasswordAuthentication(user, password);
        }
}
```

Una vez creada esta clase, desde el método main() sólo hay que crear un objeto Authenticator usando la clase anidada recién creada. Por último se obtiene la sesión usando el método getDefaultInstance() de la clase Session, pero sin pasar null como segundo parámetro -pues requerimos autenticarnos- sino que recibirá el objeto Authenticator creado:

```
Authenticator auth = new MiAutenticador();
Session sesion = Session.getDefaultInstance(props,auth);
```

6.2.2 Ejemplo completo

A continuación se muestra el ejemplo completo de cómo enviar un mensaje de correo usando un servidor SMTP que requiere autenticación.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class EnviarCorreoAutenticacion {
   private static String user,password;

// Creación de la clase interna
   private static class MiAutenticador extends Authenticator {
      public PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(user, password);
      }
   }

   public static void main (String [] args) {
   // Comprobar que el número de argumentos es el correcto
      if (args.length != 4) {
         System.out.println("Ha de enviar dos parámetros\n" +
                            "java EnviarCorreoAutenticacion" +
                            "fromAddress toAddress user password");
         System.exit(1);
```

```
        }
        String from = args[0];
        String to   = args[1];
    // Inicializar las dos variables de clase
        user       = args[2];
        password       = args[3];
    // Obtener las propiedades del sistema y establecer el servidor
    // SMTP que se va a usar
        String smtpHost  = "smtp.correo.yahoo.es";
        Properties props = System.getProperties();
        props.put("mail.smtp.host",smtpHost);
    // Esta línea indica que vamos a autenticarnos en el servidor SMTP
        props.put("mail.smtp.auth", "true");
        Authenticator auth = new MiAutenticador();
    // Obtener una sesión con las propiedades anteriormente definidas
        Session sesion = Session.getDefaultInstance(props,auth);
    // Capturar las excepciones
        try {
        // Crear un mensaje vacío
            Message mensaje = new MimeMessage(sesion);
        // Rellenar los atributos y el contenido
        // Asunto
            mensaje.setSubject("Hola Mundo autenticado");
        // Emisor del mensaje
            mensaje.setFrom(new InternetAddress(from));
        // Receptor del mensaje
            mensaje.addRecipient(Message.RecipientType.TO,
                                 new InternetAddress(to));
        // Rellenar el cuerpo del mensaje
            mensaje.setText("Este es el cuerpo del mensaje");
        // Enviar el mensaje
            Transport.send(mensaje);
        } catch (MessagingException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

6.3 Conexión segura con SSL

En este epígrafe nos centraremos en cómo descargar mensajes de un servidor que requiere conexión segura a través del protocolo SSL (*Secure Socket Layer*). Este protocolo se encarga de proporcionar seguridad en las comunicaciones mediante el cifrado de los datos que se intercambian entre el servidor y el cliente, además de proporcionar autenticación desde el punto de vista del servidor, integridad en los mensajes y, opcionalmente, autenticación de cliente para conexiones TCP/IP. Existe

otro protocolo similar, denominado TLS (*Transport Layer Security*) y desarrollado por el IETF (*Internet Engineering Task Force*) como evolución del SSL, por lo que son muy similares y muchas veces se hace referencia a uno u otro de forma indistinta.

Java implementa estos protocolos a través de la API JSSE (*Java Secure Socket Extension*). Este manual no tiene por objetivo el estudio de esta API en profundidad, por lo que sólo nos centraremos en el uso de funciones relacionadas con SSL en JavaMail. Para ver más información sobre JSSE se puede consultar la dirección http://java.sun.com/products/jsse.

Descargar mensajes de un servidor usando una conexión segura es muy similar a hacerlo de forma convencional, como veíamos en el epígrafe 3.3. En este ejemplo se mostrarán las diferencias que aparecen y al final del epígrafe se mostrará el código completo que realiza esta tarea.

6.3.1 Proveedores de seguridad y propiedades

En primer lugar hay que añadir el proveedor de servicios criptográficos de Sun para SSL (aunque puede añadirse cualquier otro que pueda resultar conveniente). Un proveedor SSL se encarga de supervisar todas las operaciones que se realicen a través de este protocolo. Para añadir el proveedor de Sun se puede utilizar el método addProvider() de la clase java.security.Security, que se encarga de centralizar todas las propiedades y los métodos más comunes relacionados con la seguridad. Así pues, la importación y la utilización quedan de la forma:

```
import java.security.Security;
...
Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
```

El proveedor también puede añadirse estáticamente en el fichero de políticas de seguridad del JRE, denominado **java.security** (de hecho suele estar añadido por defecto), incluyendo una línea de la forma:

```
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
```

en la que el número **2** es tan sólo un número en la secuencia de proveedores de servicios de seguridad que haya especificada en este fichero.

El siguiente paso consiste en establecer las propiedades de la conexión SSL. En la tabla 6.2, se muestran algunas de estas propiedades.

Dado que se pretende utilizar el protocolo POP3, deben añadirse las siguientes líneas de asignación de propiedades:

```
props.put("mail.pop3.socketFactory.class",
                                "javax.net.ssl.SSLSocketFactory");
props.put("mail.pop3.socketFactory.fallback", "false");
props.put("mail.pop3.port", "995");
props.put("mail.pop3.socketFactory.port", "995");
```

Nombre	Descripción
mail.*protocolo*.socketFactory.class	Permite especificar qué clase es la encargada de manejar los *sockets*. Dicha clase debe heredar de javax.net.SocketFactory. En el caso que nos ocupa se usa SSLSocketFactory, que emplea *sockets* basados en SSL.
mail.*protocolo*.socketFactory.fallback	Esta propiedad adopta un valor *booleano* que indica al sistema si debe volver o no a una conexión no segura si falla la conexión segura.
mail.*protocolo*.port mail.*protocolo*.socketFactory.port	Estas dos propiedades se añaden para indicar el número de puerto que usa la versión segura del protocolo. Ambos valores suelen ser iguales.
mail.*protocolo*.connectiontimeout	Con esta propiedad se indica que si la conexión no se realiza en el tiempo especificado -en milisegundos-, ésta debe ser cancelada.

Tabla 6.2 Propiedades de configuración de una conexión SSL.

Si en vez de POP3 se trabajase con un servidor basado en IMAP sólo hay que sustituir en estas propiedades la palabra **"pop3"** por **"imap"**:

```
props.put("mail.imap.socketFactory.class",
                                "javax.net.ssl.SSLSocketFactory");
props.put("mail.imap.socketFactory.fallback", "false");
props.put("mail.imap.port", "995");
props.put("mail.imap.socketFactory.port", "995");
```

6.3.2 Ejemplo completo

Como puede observarse, este ejemplo es muy parecido al de recepción de correo sin usar una conexión segura, con la diferencia de que hemos de añadir el proveeedor de Sun para SSL y establecer una serie de propiedades. A continuación puede verse el ejemplo completo en el que se recuperan los mensajes almacenados en un servidor POP3 que requiere una conexión segura.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
```

```
import java.security.Security;

public class ObtenerCorreoConSegura {
   public static void main (String [] args) {
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                                        "java ObtenerCorreo usuario clave");
         System.exit(1);
      }
   // Obtener el usuario y la clave recibidos como parámetros
      String usuario = args [0];
      String clave   = args [1];
   // Añadir el proveedor SSL de Sun
      Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
   // Obtener las propiedades del sistema
      String popHost   = "pop.gmail.com";
      Properties props =  System.getProperties();
   // Añadir las propiedades necesarias
      props.put("mail.pop3.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
      props.put("mail.pop3.socketFactory.fallback", "false");
      props.put("mail.pop3.port", "995");
      props.put("mail.pop3.socketFactory.port", "995");
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
   // Capturar las excepciones
      try {
      // Crear un Store indicando el protocolo de acceso y conectarse a
      // él
         Store store = sesion.getStore("pop3");
         store.connect(popHost,usuario,clave);
      // Crear un Folder y abrir la carpeta INBOX en modo SOLO LECTURA
         Folder folder = store.getFolder("INBOX");
         folder.open(Folder.READ_ONLY);
      // Obtener los mensajes almacenados en el Folder
         Message [] mensajes = folder.getMessages();
      // Procesar los mensajes
         for (int i= 0; i < mensajes.length; i++) {
            System.out.println("Mensaje " + i + ":\n" +
               "\tAsunto: " + mensajes[i].getSubject()  + "\n" +
               "\tRemitente: " + mensajes[i].getFrom()[0]  + "\n" +
               "\tFecha de Envío: " + mensajes[i].getSentDate() + "\n" +
               "\tContenido: " + mensajes[i].getContent()  + "\n");
         }
      // Cerrar el Folder y el Store
         folder.close(false);
         store.close();
```

```
        } catch (MessagingException me) {
            System.err.println(me.toString());
        } catch (IOException ioe) {
            System.err.println(ioe.toString());
        }
    }
}
```

6.3.3 Certificados de seguridad

Por desgracia, bajo ciertas circunstancias, el código propuesto puede dar problemas y elevarse una excepción SSLException con el texto ***"untrusted server cert chain"*** (que viene a decir algo así como "el certificado del servidor no es de confianza"). Ésta se debe a que el certificado de seguridad del servidor de correo electrónico no está instalado en el almacén de claves en que se está ejecutando nuestro programa Java. Para corregir el problema pueden adoptarse dos estrategias:

- Incluir el certificado del servidor de correo en nuestro almacén de claves, normalmente ubicado en el directorio **\jre\lib\security\cacerts**.
- Modificar el sistema que se encarga de verificar si un certificado es de confianza o no.

La segunda estrategia es más general y es la que se va a abordar a continuación. El mensaje de error anterior -caso de haberse producido- habrá sido elevado por un objeto de tipo TrustManager, que es el que controla si un certificado es o no de confianza. El TrustManager por defecto hace las cosas bien y mira en el directorio especificado anteriormente para ver si existe o no en nuestra máquina la clave del servidor de correo y, al no encontrarlo, produce el fallo. Lo que aquí se va a hacer, consiste en crear un objeto que hereda de X509TrustManager -padre de TrustManager- que siempre informa favorablemente de la confianza de cualquier certificado, incluso aunque su nombre sea distinto al del servidor. El código es tan sencillo como:

```
import com.sun.net.ssl.X509TrustManager;
import java.security.cert.X509Certificate;
public class TrustManagerSimple implements X509TrustManager {
    public boolean isClientTrusted( X509Certificate[] cert) {
        return true;
    }
    public boolean isServerTrusted( X509Certificate[] cert) {
        return true;
    }
    public X509Certificate[] getAcceptedIssuers() {
        return new X509Certificate[0];
    }
}
```

Claro está que esto supone un agujero de seguridad que, en condiciones normales, debe corregirse; de hecho, lo usual suele ser mostrar un diálogo al usuario en el que se le pide confirmación para confiar en el certificado y, si el usuario acepta, lo almacena localmente de forma automática (esto suelen hacerlo los navegadores web).

Hay un problema añadido que se debe a que la propiedad mail.pop3.socketFactory.class requiere un valor que debe ser una clase que hereda de SSLSocketFactory pero que, ahora, debe hacer uso del gestor de certificados recién creado, esto es: TrustManagerSimple. Por ello, hay que crear también nuestra propia clase SSLSocketFactory que crea los *sockets* seguros, y registrarla convenientemente.

```
import com.sun.net.ssl.*;
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;
public class SSLSocketFactorySimple extends SSLSocketFactory {
   private SSLSocketFactory factory;
   public SSLSocketFactorySimple () {
      System.out.println( "SSLSocketFactorySimple inicializado");
      try {
         SSLContext sslcontext = SSLContext.getInstance( "TLS");
         sslcontext.init( null, // No se necesita KeyManager
                  new TrustManager[] { new TrustManagerSimple()},
                  new java.security.SecureRandom());
         factory = ( SSLSocketFactory) sslcontext.getSocketFactory();
      } catch( Exception ex) {
         ex.printStackTrace();
      }
   }
   public static SocketFactory getDefault() {
      return new SSLSocketFactorySimple();
   }
    public Socket createSocket( Socket socket, String s, int i, boolean flag) throws
IOException {
      return factory.createSocket( socket, s, i, flag);
   }
   public Socket createSocket( InetAddress inaddr, int i, InetAddress inaddr1, int j)
throws IOException {
      return factory.createSocket( inaddr, i, inaddr1, j);
   }
   public Socket createSocket( InetAddress inaddr, int i) throws IOException {
      return factory.createSocket( inaddr, i);
```

```
    }
     public Socket createSocket( String s, int i, InetAddress inaddr, int j) throws
IOException {
       return factory.createSocket( s, i, inaddr, j);
    }
    public Socket createSocket( String s, int i) throws IOException {
       return factory.createSocket( s, i);
    }
    public String[] getDefaultCipherSuites() {
      return factory.getSupportedCipherSuites();
    }
    public String[] getSupportedCipherSuites() {
      return factory.getSupportedCipherSuites();
    }
}
```

Por último, en lugar de registrar al proveedor de *sockets* como:

```
props.put("mail.pop3.socketFactory.class",
                                    "javax.net.ssl.SSLSocketFactory");
```

debe hacerse como:

```
Security.setProperty( "ssl.SocketFactory.provider",
                                    "SSLSocketFactorySimple");
```

Capítulo 7

Acuse de recibo y prioridades. Búsquedas

7.1 Introducción

En este capítulo se cubren los últimos flecos interesantes de la API JavaMail. En concreto, se analiza cómo enviar y recibir mensajes de diferentes prioridades y/o con acuse de recibo, así como las posibilidades de búsquedas de mensajes en el propio servidor y sin necesidad de efectuar descargas a las aplicaciones clientes. Esto último puede tener una gran utilidad en servidores IMAP, en los que los mensajes pueden clasificarse en carpetas antes de ser accedidos por los clientes.

7.2 Acuse de recibo

El acuse de recibo (en inglés, *acknowledgment of receipt* o simplemente *acknowledgment*) es tan sólo una confirmación que recibe el emisor en el momento en que el receptor accede al mensaje. Cuando el receptor abra el mensaje le aparecerá una ventana en la que se le indicará que el emisor ha solicitado que se confirme la recepción de ese mensaje, teniendo el receptor potestad para decidir si quiere que el emisor reciba esa confirmación o no. Esta ventana suele ser gestionada por el propio cliente de correo electrónico. La figura 7.1 muestra un ejemplo de esta ventana.

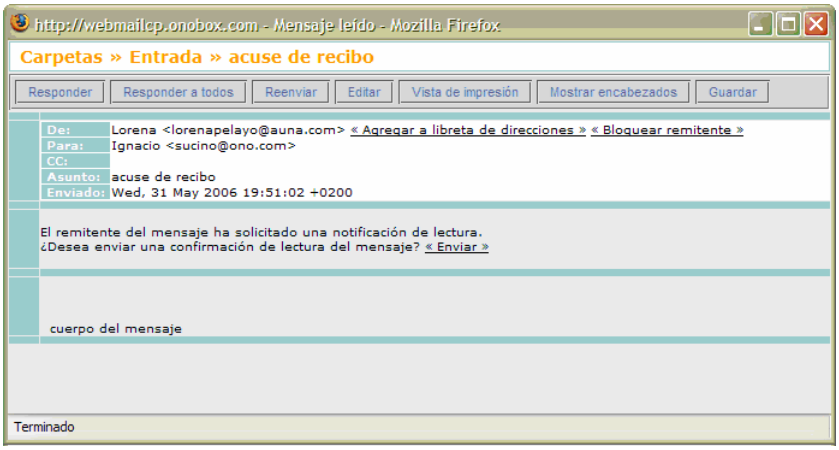

Figura 7.1 Ventana cliente para el acuse de recibo.

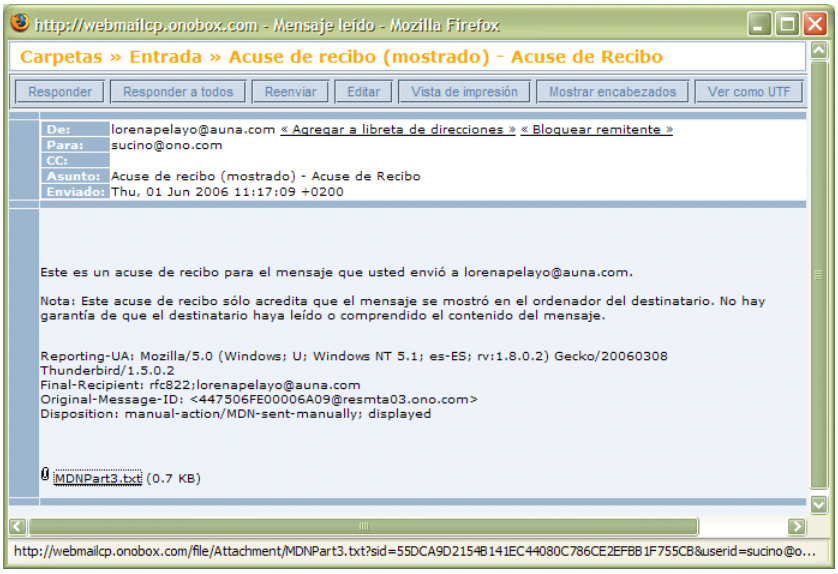

Figura 7.2 Confirmación de acuse de recibo

Si, cuando el receptor vea esta ventana, selecciona «Enviar», el emisor recibirá un mensaje con el que sabrá que el correo que envió ha sido recibido. El mensaje podría ser parecido al que se muestra en la figura 7.2.

Hay que destacar que algunos servidores de correo no soportan esta característica por lo que, un mensaje de correo con acuse de recibo enviado a una cuenta alojada en uno de estos servidores será tratado como cualquier otro, ignorándose toda la información relacionada con el acuse de recibo.

Manejar el acuse de recibo con JavaMail es extremadamente sencillo, ya que tan sólo hay que añadir un campo en la cabecera del mensaje. Como se comentó en el apartado 1.3, la cabecera (o sobre) de un mensaje contiene la información necesaria para que se complete la transmisión y entrega del mismo. Dentro de esta información se encuentran los campos *to*, *from*, *subject*, *Cc* (*Carbon Copy*), etc.

Para añadir la cabecera se usa el método addHeader() de la interfaz Part, implementada por la clase Message. Este método recibe como parámetros dos cadenas, la primera con el nombre del campo de la cabecera y la segunda con el valor de éste. Como primer parámetro hay que pasarle exactamente la cadena **"Disposition-Notification-To"**, que significa que el mensaje contiene una confirmación de lectura, y como segundo parámetro una cadena con la dirección a la que hay que mandar esa confirmación, que normalmente suele ser la dirección del remitente (*from*). No obstante, si se desea que la confirmación llegue a otra dirección,

sólo hay que colocar ésta como segundo parámetro, es decir, no es obligatorio poner aquí la misma dirección que en el *from*:

```
mensaje.addHeader("Disposition-Notification-To",from);
```

Si en vez de mandar un acuse de recibo, lo que se desea es comprobar si un mensaje lo tiene o no, se debe usar la siguiente línea de código:

```
String [] acuseRecibo =
        mensajes[i].getHeader ("Disposition-Notification-To");
```

donde acuseRecibo[0] contiene la dirección a la que hay que enviar la notificación de lectura, o bien null si el mensaje no contiene acuse de recibo.

Por último, es importante hacer notar que el manejo del acuse de recibo y de la prioridad (que se verá en el siguiente epígrafe) son muy similares, pues en ambos casos se trata de incorporar propiedades a la cabecera del mensaje.

7.2.1 Ejemplo completo

A continuación se muestra el código completo de cómo enviar un mensaje de texto plano con acuse de recibo.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class EnviarCorreoAcuseRecibo {
   public static void main (String [] args) {
   // Comprobar que el número de argumentos es el correcto
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                  "java EnviarCorreoAcuseRecibo fromAddress toAddress");
         System.exit(1);
      }
   // Obtener el from y el to recibidos como parámetros
      String from = args [0];
      String to = args [1];
   // Obtener las propiedades del sistema y establecer el servidor SMTP
   // que se va a usar
      String smtpHost  = "smtp.ono.com";
      Properties props = System.getProperties();
      props.put("mail.smtp.host",smtpHost);
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
   // Capturar las excepciones
      try {
      // Crear un mensaje vacío
         Message mensaje = new MimeMessage(sesion);
```

```
        // Rellenar los atributos y el contenido
        // Asunto
            mensaje.setSubject("Hola Mundo con acuse de recibo");
        // Emisor del mensaje
            mensaje.setFrom(new InternetAddress(from));
        // Receptor del mensaje
            mensaje.addRecipient(Message.RecipientType.TO,
                new InternetAddress(to));
        // Rellenar el cuerpo del mensaje
            mensaje.setText("Este es el cuerpo del mensaje");
        // Acuse de recibo
            mensaje.addHeader("Disposition-Notification-To",from);
        // Enviar el mensaje
            Transport.send(mensaje);
        } catch (MessagingException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

7.3 Prioridad

La prioridad de un mensaje es simplemente un indicativo que se añade al mensaje en el momento de enviarlo, para que el destinatario sepa la importancia de éste. Esto no quiere decir que por modificar la prioridad de un mensaje, éste vaya a transportarse más rápido o se vaya a leer antes que cualquier otro mensaje sino que, simplemente, al ver su prioridad el destinatario decidirá si lo abre primero o no. Hay que indicar que en la mayoría de las ocasiones no se suele usar, y los mensajes se envían con la prioridad que tengan establecida por defecto.

En este epígrafe se tratan por separado dos casos:

- Establecer la prioridad de un mensaje a la hora de enviarlo
- Obtener la prioridad de un mensaje cuando se recibe

7.3.1 Prioridad en el envío de mensajes

Establecer la prioridad de un mensaje usando JavaMail es muy sencillo, y tan sólo hay que añadir una línea de código. Para ver cómo funciona esto, nos centraremos una vez más en el ejemplo en el que se envía un mensaje de texto plano (epígrafe 3.2) y cambiaremos la prioridad al mensaje. Para ello, y al igual que en el apartado 7.2, es necesario emplear el método addHeader() de la interfaz Part, implementada por la clase Message. Este método recibe como parámetros dos cadenas: como primer parámetro se le pasa la cadena **"X-Priority"**, que significa que

se desea establecer la prioridad del mensaje, y como segundo parámetro una cadena con el valor de dicha prioridad.

JavaMail maneja cinco valores de prioridad, correspondientes a los números del 1 al 5 en forma de String, donde el número 1 representa la prioridad más alta y el 5 la prioridad más baja. Será responsabilidad del programa que reciba el mensaje interpretar estas prioridades y mostrarlas al usuario de una forma adecuada. Si al usar addHeader() se especifica como **valorPrioridad** una cadena alfanumérica distinta de éstas (por ejemplo "0", "6", "a"), será equivalente a añadir un valor de prioridad 3, es decir, prioridad Normal. En la tabla 7.1 pueden verse estos valores y su significado.

Valor de prioridad	Comportamiento
"1"	La más alta
"2"	Alta
"3"	Normal
"4"	Baja
"5"	La más baja
Valor distinto a los anteriores	Normal

Tabla 7.1 Valores de prioridad en el envío de mensajes.

Así pues, para modificar la prioridad basta añadir la siguiente línea de código:

```
mensaje.addHeader("X-Priority", "valorPrioridad");
```

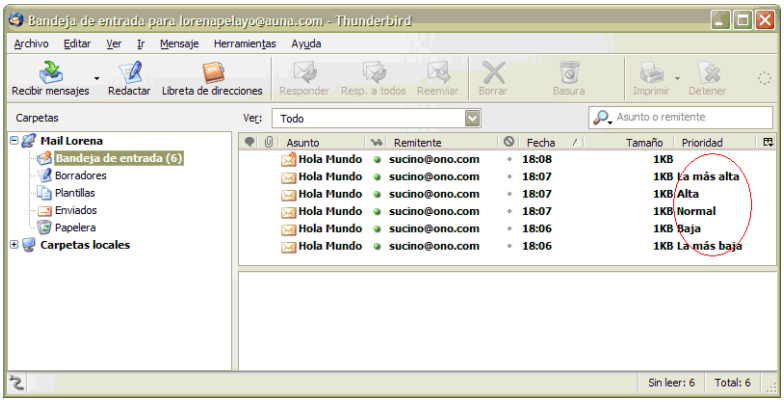

Figura 7.3 Gestión de prioridades en Mozilla Thunderbird.

En la figura 7.3 puede apreciarse el resultado de enviar 5 mensajes, con prioridad 5, 4, 3 ,2 y 1 respectivamente y un mensaje en el que no establecemos la prioridad, y como maneja la prioridad de estos mensajes el gestor de correo Mozilla Thunderbird (versión 1.5.0.4). Puede observarse cómo cada mensaje aparece con la prioridad indicada en la tabla 7.1, excepto el mensaje en el que no se establece la prioridad, que no tiene ningún mensaje asociado a ésta.

7.3.2 Ejemplo completo

A continuación puede verse el código completo de cómo enviar un mensaje de texto plano con la prioridad más alta.

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class EnviarCorreoPrioridad {
   public static void main (String [] args) {
   // Comprobamos que el número de argumentos es el correcto
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                  "java EnviarCorreoPrioridad fromAddress toAddress");
         System.exit(1);
      }
   // Obtener el from y el to recibidos como parámetros
      String from = args [0];
      String to    = args [1];
   // Obtener las propiedades del sistema y establecer el servidor SMTP que
   // se va a usar
      String smtpHost  = "smtp.auna.com";
      Properties props = System.getProperties();
      props.put("mail.smtp.host",smtpHost);
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
   // Capturar las excepciones
      try {
      // Crear un mensaje vacío
         Message mensaje = new MimeMessage(sesion);
      // Rellenar los atributos y el contenido
      // Asunto
         mensaje.setSubject("Hola Mundo");
      // Emisor del mensaje
         mensaje.setFrom(new InternetAddress(from));
      // Receptor del mensaje
         mensaje.addRecipient(Message.RecipientType.TO,
                                    new InternetAddress(to));
```

```
        // Rellenar el cuerpo del mensaje
            mensaje.setText("Este es el cuerpo del mensaje");
        // Añadir la prioridad
                        /*
                         1 => La más alta
                         2 => Alta
                         3 => Normal
                         4 => Baja
                         5 => La más baja
                        */
            mensaje.addHeader("X-Priority", "1");
        // Enviar el mensaje
            Transport.send(mensaje);
        } catch (MessagingException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

7.3.3 Prioridad en la recepción de mensajes

Ya se ha visto cómo establecer la prioridad en el envío de mensajes, por lo que ahora toca estudiar cómo obtener la prioridad en la recepción, una vez que el mensaje ha sido recuperado del servidor de correo.

Para llevar a cabo esta tarea nos basaremos en el ejemplo del epígrafe 3.3, en el que se recuperaban mensajes de texto plano. En este caso, una vez recuperarados los mensajes, por cada uno de ellos se mostrará su prioridad, además de los demás campos.

En esta ocasión se debe usar el método complementario a addHeader(), esto es, el método getHeader() de la interfaz Part de la que hereda la clase Message, a la que pertenecen los mensajes recuperados del servidor. Este método recibe como único parámetro una cadena, que es el nombre del campo de la cabecera del mensaje que se desea obtener. Como ya se ha comentado, la cabecera de un mensaje son los campos *from*, *to*, etc., que sirven para encaminarlo a través de la red y que llegue a su destino. Este método devuelve un *array* de String, con la cadena o cadenas que forman el contenido del campo de la cabecera solicitado:

```
String [] prioridad = mensajes[i].getHeader ("X-Priority");
```

Si el mensaje no tiene prioridad establecida este método devuelve el valor null, y al visualizar por pantalla la prioridad del mensaje mostraremos la cadena **"No tiene prioridad establecida"**. En caso de que el mensaje tenga prioridad, y por tanto el método getHeader() no devuelva null, el array devuelto sólo contendrá una cadena,

con lo que habrá que acceder a la posición número cero del *array* y mostrarla por pantalla.

```
String [] prioridad = mensajes[i].getHeader ("X-Priority");
String prioridadMensaje;
if (prioridad != null)
        prioridadMensaje = prioridad[0];
else
        prioridadMensaje = "No tiene prioridad establecida";
```

De esta forma la prioridad se mostraría por pantalla de la forma numérica 1, 2 , 3, 4 y 5. Para dar una interpretación más clara a estos valores, se puede usar una sentencia de selección if en cascada, para visualizar por cada valor su correspondiente significado. Hemos de tener en cuenta que cuando se tiene como valor de prioridad cualquier valor alfanumérico, distinto de **"1"**, **"2"**, **"3"**, **"4"** o **"5"**, como por ejemplo **"a"**, **"6"**, etc., será equivalente a disponer del valor **"3"**, es decir, prioridad normal; por tanto la rama de la sentencia if que controla el **"3"**, o cualquier otra cadena distinta de **"1"**, **"2"**, **"4"** o **"5"**, será la misma.

Hay que tener en cuenta también, que cada gestor de correo puede establecer la prioridad de forma diferente. Por ejemplo, si se envía un mensaje desde Mozilla Thunderbird con la prioridad más alta, al obtener ésta con JavaMail devolverá la cadena **"1 (Highest)"**. Por tanto hay de comprobar que la prioridad más alta contenga un 1, no que sea igual a 1. A su vez, con esta comprobación surge el problema de que, si por ejemplo se recibe un valor **"15"** de prioridad, y tan sólo se comprueba si contiene un **"1"**, tal comprobación devolverá **true** y parecerá que la prioridad es la más alta, cuando en realidad se corresponde con la normal. Para evitar esto puede hacerse uso de la clase StreamTokenizer tal y como aparece a continuación:

```
String [] prioridad = mensajes[i].getHeader ("X-Priority");
String prioridadMensaje="Normal";
if (prioridad != null) {
        StreamTokenizer st= new StreamTokenizer(new StringReader(prioridad[0]));
        try{
                if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                        switch((int)st.nval){
                        case 1: prioridadMensaje="La más alta"; break;
                        case 2: prioridadMensaje="Alta"; break;
                        case 4: prioridadMensaje="Baja"; break;
                        case 5: prioridadMensaje="La más baja"; break;
                        }
                }catch(IOException x){
                        // Se asume que la prioridad es normal.
        }
}
```

7.3.4 Ejemplo completo

A continuación se muestra el ejemplo completo de cómo obtener los mensajes almacenados en el servidor de correo mostrando su prioridad.

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class ObtenerCorreoPrioridad {
   public static void main (String [] args) {
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                                    "java ObtenerCorreoPrioridad usuario clave");
         System.exit(1);
      }
   // Obtener el usuario y la clave recibidos como parámetros
      String usuario = args [0];
      String clave   = args [1];
   // Obtener las propiedades del sistema
      String popHost = "pop.correo.yahoo.es";
      Properties props = System.getProperties();
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
   // Capturar las excepciones
      try {
      // Crear un Store indicando el protocolo de acceso y conectarse a él
         Store store = sesion.getStore("pop3");
         store.connect(popHost,usuario,clave);
      // Crear un Folder y abrir la carpeta INBOX en modo SOLO LECTURA
         Folder folder = store.getFolder("INBOX");
         folder.open(Folder.READ_ONLY);
      // Obtener los mensajes almacenados en el Folder
         Message [] mensajes = folder.getMessages();
      // Procesar los mensajes
         for (int i= 0; i < mensajes.length; i++) {
      // Obtener la prioridad de cada mensaje
            String [] prioridad = mensajes[i].getHeader ("X-Priority");
            String prioridadMensaje="Normal";
            if (prioridad != null) {
               StreamTokenizer st= new StreamTokenizer(
                                          new StringReader(prioridad[0]));
               try{
                  if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                     switch((int)st.nval){
                        case 1: prioridadMensaje="La más alta"; break;
```

```
                        case 2: prioridadMensaje="Alta"; break;
                        case 4: prioridadMensaje="Baja"; break;
                        case 5: prioridadMensaje="La más baja"; break;
                    }
                }catch(IOException x){
                // Se asume que la prioridad es normal.
                }
            }// Visualizar por pantalla los mensajes
            System.out.println("Mensaje "           + i + ":\n" +
                "\tAsunto: " + mensajes[i].getSubject()  + "\n" +
                "\tRemitente: " + mensajes[i].getFrom()[0]  + "\n" +
                "\tFecha de Envío: " + mensajes[i].getSentDate() + "\n" +
                "\tContenido: " + mensajes[i].getContent()  + "\n" +
                "\tPrioridad: " + prioridadMensaje           + "\n");
        }
      // Cerrar el Folder y el Store
        folder.close(false);
        store.close();
      } catch (MessagingException me) {
        System.err.println(me.toString());
      } catch (IOException ioe) {
        System.err.println(ioe.toString());
      }
    }
}
```

7.4 Búsquedas

JavaMail incorpora la capacidad de buscar mensajes que cumplan ciertas características dentro de una carpeta concreta. Esta búsqueda se hace directamente en el servidor, sin necesidad de que los mensajes sean descargados a la aplicación cliente.

Así pues, para realizar las búsquedas en JavaMail se utilizan las clases del paquete javax.mail.search. Concretamente, en lugar de descargar los mensajes de un objeto Folder, a través del método getMessages(), se empleará el método search() también de la clase Folder. Este método está sobrecargado y se puede usar con las dos signaturas que proporciona y que se muestran en la tabla 7.2.

Para realizar búsquedas es necesario construir uno o varios criterios de filtrado, usando para ello la clase SearchTerm y sus clases herederas, que implementan métodos de coincidencia de búsqueda específicos. Se trata de una clase abstracta, que tiene un sólo método, con la cabecera:

```
public abstract boolean match(Message msg)
```

Este método, una vez construido un criterio de búsqueda, indica para cada mensaje que recibe como parámetro si concuerda o no con el criterio a que representa.

Método	Comportamiento
Message[] search(SearchTerm term)	Devuelve todos los mensajes almacenados en la carpeta que recibe el mensaje, y que concuerden con el criterio de búsqueda expresado en el parámetro *term*.
Message[] search(SearchTerm term, Message[] msgs)	De los mensajes que recibe como segundo parámetro, devuelve un subconjunto formado por aquellos mensajes que concuerden con el criterio de búsqueda expresado en el primer parámetro.

Tabla 7.2 Métodos para búsqueda de mensajes que proporciona la clase Folder.

7.4.1 La clase SearchTerm

Como la clase SearchTerm es abstracta, es necesario usar sus subclases para construir expresiones de búsqueda. Algunas de estas subclases (en concreto, sus constructores) aparecen en la tabla 7.4.

Antes de continuar, vamos a mostrar un par de ejemplos de búsquedas con JavaMail; para ellos suponemos que la variable folder posee el objeto Folder en el que se desea buscar los mensajes. En el primer ejemplo, simplemente se crea un término de búsqueda que localiza los mensajes cuyo asunto contenga la palabra **"hola"**, para lo cual es necesario hacer uso del constructor de la claseque busca en el campo *subject*, es decir, la clase SubjectTerm, y luego descargar los mensajes del Folder que coincidan con ese término:

```
SearchTerm stAsunto = new SubjectTerm ("hola");
Message [] mensajesAsunto = folder.search(stAsunto);
```

El segundo ejemplo, consiste en recuperar los mensajes que tienen un tamaño mayor de 1024 bytes. Para ello, se usa la clase SizeTerm, que es subclase de la clase abstracta ComparisonTerm. SizeTerm se usa para realizar comparaciones en cuanto a lo que el tamaño del mensaje se refiere. Usaremos el constructor de esta clase indicándole como primer parámetro el tipo de comparación que queremos realizar, en este caso **mayor que**, para lo que se proporciona la constante ComparisonTerm.GT (*Greater Than* o mayor que). Como segundo parámetro se le pasa el tamaño en bytes:

```
SearchTerm stSize = new SizeTerm (ComparisonTerm.GT, 1024);
```

Constructor de subclase	Objetivo de la subclase
AndTerm(SearchTerm t1, SearchTerm t2)	Este constructor crea un término de búsqueda que hace la operación lógica *and* entre los dos términos de búsqueda pasados como parámetros.
AndTerm(SearchTerm[] t)	Este constructor crea un término de búsqueda que hace la operación lógica *and* entre todos los términos de búsqueda pasados como parámetros en el *array t*.
OrTerm(SearchTerm t1, SearchTerm t2)	Este constructor crea un término de búsqueda que hace la operación lógica *or* entre los dos términos de búsqueda que pasados como parámetros.
OrTerm(SearchTerm[] t)	Este constructor crea un término de búsqueda que hace la operación lógica *or* entre todos los términos de búsqueda pasados como parámetros en el *array t*.
NotTerm(SearchTerm t)	Este constructor crea un término de búsqueda que hace la operación lógica *not* con respecto al término de búsqueda pasado como parámetro.

Tabla 7.3 Constructores de subclases de SearchTerm que permiten construir términos de búsqueda en los que interviene los operadores lógicos *and*, *or* y *not*.

```
Message [] mensajesTamaño = folder.search(stSize);
```

La clase abstracta ComparisonTerm proporciona constantes enteras que permiten realizar comparaciones para diferentes tipos de datos:

- Si un campo es mayor que otro dado (ComparisonTerm.EQ),
- Si es mayor o igual (ComparisonTerm.GE),
- Mayor que (ComparisonTerm.GT),
- Menor o igual (ComparisonTerm.LE),
- Menor que (ComparisonTerm.LT),
- No igual (ComparisonTerm.NE).

En el ejemplo anterior ha quedado ilustrada la utilización de esta clase.

También es posible construir términos de búsqueda en los que intervienen operadores lógicos. Las clases que lo permiten se usan para ver si se cumplen dos o más criterios de búsqueda a la vez (usando la clase AndTerm), si se cumple alguno de los criterios de búsqueda de entre dos o más posibilidades (usando la clase OrTerm) o si no se da un criterio de búsqueda (usando la clase NotTerm). En la tabla 7.3 aparecen estas clases.

Para ilustrar la utilización de estas clases realizaremos una búsqueda en la que se den las dos condiciones vistas anteriormente, es decir, recuperar los mensajes

Constructor de subclase	Objetivo de la subclase
BodyTerm (String texto)	Este constructor crea un término de búsqueda que busca en el cuerpo del mensaje la cadena pasada como parámetro.
FromStringTerm(String texto)	Este constructor crea un término de búsqueda que busca en el campo *from* del mensaje la cadena pasada como parámetro.
RecipientStringTerm(Message.RecipientType tipo, String texto)	Este constructor crea un término de búsqueda que busca en el campo *to*, *cc* o *bcc* del mensaje la cadena pasada como segundo parámetro, en función de que el primer parámetro sea Message.RecipientType.TO, Message.RecipientType.CC o Message.RecipientType.BCC respectivamente.
SubjectTerm (String texto)	Este constructor crea un término de búsqueda que busca en el campo *subject* del mensaje la cadena pasada como parámetro.
SizeTerm(int comparador, int tamaño)	Este constructor crea un término de búsqueda que busca aquellos mensajes de tamaño mayor menor o igual a su parámetro *tamaño*. El parámetro *comparador* identifica el tipo de comparación que se desea realizar.

Tabla 7.4 Constructores de subclases de SearchTerm que permiten construir término de búsqueda

que contengan la cadena **"hola"** en el campo *asunto*, y además tengan un *tamaño* mayor de 1024 bytes. Para ello, se usan los dos términos de búsqueda creados anteriormente, y se realiza la operación lógica *and* entre ellos usando la clase AndTerm:

```
SearchTerm stAnd = new AndTerm (stAsunto,stSize);
Message [] mensajesAnd = folder.search(stAnd);
```

7.4.2 Ejemplo completo

Para terminar el capítulo, veremos el ejemplo completo de cómo realizar las tres búsquedas mencionadas.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.mail.search.*;
import java.io.*;

public class Busquedas {
   public static void main (String [] args) {
      if (args.length != 2) {
         System.out.println("Ha de enviar dos parámetros\n" +
                                        "java Busquedas usuario clave");
         System.exit(1);
      }
   // Obtener el usuario y la clave recibidos como parámetros
      String usuario = args [0];
      String clave   = args [1];
   // Obtener las propiedades del sistema
      String popHost   = "pop.correo.yahoo.es";
      Properties props = System.getProperties();
   // Obtener una sesión con las propiedades anteriormente definidas
      Session sesion = Session.getDefaultInstance(props,null);
   // Capturar las excepciones
      try {
      // Crear un Store indicando el protocolo de acceso y conectarse a él
         Store store = sesion.getStore("pop3");
         store.connect(popHost,usuario,clave);
      // Crear un Folder y abrir la carpeta INBOX en modo SOLO LECTURA
         Folder folder = store.getFolder("INBOX");
         folder.open(Folder.READ_ONLY);
      // Buscar los mensajes que en el campo asunto contengan la palabra hola
         SearchTerm stAsunto = new SubjectTerm ("hola");
         Message [] mensajesAsunto = folder.search(stAsunto);
      // Procesar los mensajes
         System.out.println ("Mensajes que contienen en el campo asunto la" +
                                        "palabra hola \n");
         for (int i= 0; i < mensajesAsunto.length; i++) {
            System.out.println("Mensaje "           + i + ":\n" +
              "\tAsunto: " + mensajesAsunto[i].getSubject()  + "\n" +
              "\tRemitente: " + mensajesAsunto[i].getFrom()[0]  + "\n" +
              "\tFecha de Envío: "+ mensajesAsunto[i].getSentDate() + "\n" +
              "\tContenido: " + mensajesAsunto[i].getContent()  + "\n");
```

```
            }
        // Buscar los mensajes que tengan un tamaño mayor de 1024 bytes
            SearchTerm stSize = new SizeTerm (ComparisonTerm.GT, 1024);
            Message [] mensajesTamanyo = folder.search(stSize);
        // Procesar los mensajes
            System.out.println ("Mensajes con un tamaño mayor de 1024 bytes\n");
            for (int i= 0; i < mensajesTamanyo.length; i++) {
                System.out.println("Mensaje "            + i + ":\n" +
                  "\tAsunto: " + mensajesTamanyo[i].getSubject()  + "\n" +
                  "\tRemitente: " + mensajesTamanyo[i].getFrom()[0]  + "\n" +
                  "\tFecha de Envío: " + mensajesTamanyo[i].getSentDate() + "\n" +
                  "\tTamaño: " + mensajesTamanyo[i].getSize()     + "\n" +
                  "\tContenido: " + mensajesTamanyo[i].getContent()  + "\n");
            }
        // Buscar los mensajes que en el campo asunto contengan la palabra hola
        // y tengan un tamaño mayor de 1024 bytes
            SearchTerm stAnd = new AndTerm (stAsunto,stSize);
            Message [] mensajesAnd = folder.search(stAnd);
        // Procesar los mensajes
            System.out.println ("Mensajes que contienen en el campo asunto la" +
                "palabra hola y \n tienen un tamaño mayor de 1024 bytes \n");
            for (int i= 0; i < mensajesAnd.length; i++) {
                System.out.println("Mensaje "            + i + ":\n" +
                  "\tAsunto: " + mensajesAnd[i].getSubject()  + "\n" +
                  "\tRemitente: " + mensajesAnd[i].getFrom()[0]  + "\n" +
                  "\tFecha de Envío: " + mensajesAnd[i].getSentDate() + "\n" +
                  "\tTamaño: " + mensajesTamanyo[i].getSize() + "\n" +
                  "\tContenido: " + mensajesAnd[i].getContent()  + "\n" );
            }
        // Cerrar el Folder y el Store
            folder.close(false);
            store.close();
        } catch (MessagingException me) {
            System.err.println(me.toString());
        } catch (IOException ioe) {
            System.err.println(ioe.toString());
        }
    }
}
```

Java a Tope: JavaMail

(JavaMail en ejemplos)

El presente volumen aborda uno de los tipos de comunicaciones más antiguos utilizados en internet: el correo electrónico. Ello se hace desde la perspectiva del programador en Java, de tal manera que es posible incorporar en un programa las funcionalidades necesarias para enviar y recibir mensajes.

Los aspectos tratados en los distintos capítulos abordan la mayoría de situaciones reales con que puede encontrarse un programador: desde el envío de mensajes escritos en lenguaje HTML con imágenes incrustadas, hasta la gestión de carpetas en servidores IMAP, pasando por el establecimiento de prioridades y acuses de recibo, o la inclusión de adjuntos en los mensajes. Los aspectos relativos a la seguridad también son tratados, tanto desde la perspectiva de la seguridad en las transmisiones como de la autenticación por parte de los usuarios.

Los ejemplos completos que se encuentran al final de cada epígrafe suponen una gran ayuda para aquellos lectores que no se quieran preocupar del porqué, sino sólo del cómo y que sólo requieren una solución a sus necesidades. Para los demás, cada ejemplo se ve acompañado de una extensa explicación, así como de tablas y resúmenes que contemplan las amplias capacidades que ofrece la API JavaMail.

ISBN 84-690-0697-5