Flujos de E/S y Red

Índice

Copyright © 2008-2009 Depto. CCIA All rights reserved.

1. Entrada/Salida

Los programas muy a menudo necesitan enviar datos a un determinado destino, o bien leerlos de una determinada fuente externa, como por ejemplo puede ser un fichero para almacenar datos de forma permanente, o bien enviar datos a través de la red, a memoria, o a otros programas. Esta entrada/salida de datos en Java la realizaremos por medio de *flujos (streams)* de datos, a través de los cuales un programa podrá recibir o enviar datos en serie.

1.1. Flujos de datos de entrada/salida

Existen varios objetos que hacen de flujos de datos, y que se distinguen por la finalidad del flujo de datos y por el tipo de datos que viajen a través de ellos. Según el tipo de datos que transporten podemos distinguir:

- Flujos de caracteres
- Flujos de *bytes*

Dentro de cada uno de estos grupos tenemos varios pares de objetos, de los cuales uno nos servirá para leer del flujo y el otro para escribir en él. Cada par de objetos será utilizado para comunicarse con distintos elementos (memoria, ficheros, red u otros programas). Estas clases, según sean de entrada o salida y según sean de caracteres o de *bytes* llevarán distintos sufijos, según se muestra en la siguiente tabla:

	Flujo de entrada / lector	Flujo de salida / escritor
Caractéres	`_Reader`	`_Writer`
Bytes	`_InputStream`	`_OutputStream`

Donde el prefijo se referirá a la fuente o sumidero de los datos que puede tomar valores como los que se muestran a continuación:

`File_`	Acceso a ficheros
`Piped_`	Comunicación entre programas mediante tuberías (*pipes*)
`String_`	Acceso a una cadena en memoria (solo caracteres)
`CharArray_`	Acceso a un *array* de caracteres en memoria (solo caracteres)
`ByteArray_`	Acceso a un *array* de *bytes* en memoria (solo *bytes*)

Además podemos distinguir los flujos de datos según su propósito, pudiendo ser:

- Canales de datos, simplemente para leer o escribir datos directamente en una fuente o sumidero externo.
- Flujos de procesamiento, que además de enviar o recibir datos realizan algún

Copyright © 2008-2009 Depto. CCIA All rights reserved.

procesamiento con ellos. Tenemos por ejemplo flujos que realizan un filtrado de los datos que viajan a través de ellos (con prefijo `Filter`), conversores datos (con prefijo `Data`), *bufferes* de datos (con prefijo `Buffered`), preparados para la impresión de elementos (con prefijo `Print`), etc.

Un tipo de filtros de procesamiento a destacar son aquellos que nos permiten convertir un flujo de *bytes* a flujo de caracteres. Estos objetos son `InputStreamReader` y `OutputStreamWriter`. Como podemos ver en su sufijo, son flujos de caracteres, pero se construyen a partir de flujos de *bytes*, permitiendo de esta manera acceder a nuestro flujo de *bytes* como si fuese un flujo de caracteres.

Para cada uno de los tipos básicos de flujo que hemos visto existe una superclase, de la que heredaran todos sus subtipos, y que contienen una serie de métodos que serán comunes a todos ellos. Entre estos métodos encontramos los métodos básicos para leer o escribir caracteres o *bytes* en el flujo a bajo nivel. En la siguiente tabla se muestran los métodos más importantes de cada objeto:

InputStream	**read**(), **reset**(), **available**(), **close**()
OutputStream	**write**(int b), **flush**(), **close**()
Reader	**read**(), **reset**(), **close**()
Writer	**write**(int c), **flush**(), **close**()

A parte de estos métodos podemos encontrar variantes de los métodos de lectura y escritura, otros métodos, y además cada tipo específico de flujo contendrá sus propios métodos. Todas estas clases se encuentran en el paquete `java.io`. Para más detalles sobre ellas se puede consultar la especificación de la API de Java.

1.2. Entrada, salida y salida de error estándar

Al igual que en C, en Java también existen los conceptos de entrada, salida, y salida de error estándar. La entrada estándar normalmente se refiere a lo que el usuario escribe en la consola, aunque el sistema operativo puede hacer que se tome de otra fuente. De la misma forma la salida y la salida de error estándar lo que hacen normalmente es mostrar los mensajes y los errores del programa respectivamente en la consola, aunque el sistema operativo también podrá redirigirlas a otro destino.

En Java esta entrada, salida y salida de error estándar se tratan de la misma forma que cualquier otro flujo de datos, estando estos tres elementos encapsulados en tres objetos de flujo de datos que se encuentran como propiedades estáticas de la clase `System`:

	Tipo	**Objeto**
Entrada estándar	`InputStream`	`System.`**`in`**
Salida estándar	`PrintStream`	`System.`**`out`**

Copyright © 2008-2009 Depto. CCIA All rights reserved.

Salida de error estándar	`PrintStream`	`System.err`

Para la entrada estándar vemos que se utiliza un objeto `InputStream` básico, sin embargo para la salida se utilizan objetos `PrintWriter` que facilitan la impresión de texto ofreciendo a parte del método común de bajo nivel `write` para escribir *bytes*, dos métodos más: `print` y `println`. Estas funciones nos permitirán escribir cualquier cadena, tipo básico, o bien cualquier objeto que defina el método `toString` que devuelva una representación del objeto en forma de cadena. La única diferencia entre los dos métodos es que el segundo añade automáticamente un salto de línea al final del texto impreso, mientras que en el primero deberemos especificar explícitamente este salto.

Para escribir texto en la consola normalmente utilizaremos:

```
System.out.println("Hola mundo");
```

En el caso de la impresión de errores por la salida de error de estándar, deberemos utilizar:

```
System.err.println("Error: Se ha producido un error");
```

Además la clase `System` nos permite sustituir estos flujos por defecto por otros flujos, cambiando de esta forma la entrada, salida y salida de error estándar.

Truco

Podemos ahorrar tiempo si en Eclipse en lugar de escribir `System.out.println` escribimos simplemente `sysout` y tras esto pulsamos *Ctrl + Espacio*.

1.3. Acceso a ficheros

Podremos acceder a ficheros bien por caracteres, o bien de forma binaria (por *bytes*). Las clases que utilizaremos en cada caso son:

	Lectura	**Escritura**
Caracteres	`FileReader`	`FileWriter`
Binarios	`FileInputStream`	`FileOutputStream`

Para crear un lector o escritor de ficheros deberemos proporcionar al constructor el fichero del que queremos leer o en el que queramos escribir. Podremos proporcionar esta información bien como una cadena de texto con el nombre del fichero, o bien construyendo un objeto `File` representando al fichero al que queremos acceder. Este objeto nos permitirá obtener información adicional sobre el fichero, a parte de permitirnos realizar operaciones sobre el sistema de ficheros.

A continuación vemos un ejemplo simple de la copia de un fichero carácter a carácter:

```
public void copia_fichero() {
```


Copyright © 2008-2009 Depto. CCIA All rights reserved.

```
  int c;
  try {
    FileReader in = new FileReader("fuente.txt");
    FileWriter out = new FileWriter("destino.txt");

    while( (c = in.read()) != -1) {
      out.write(c);
    }

    in.close();
    out.close();

  } catch(FileNotFoundException e1) {
    System.err.println("Error: No se encuentra el fichero");
  } catch(IOException e2) {
    System.err.println("Error leyendo/escribiendo fichero");
  }
}
```

En el ejemplo podemos ver que para el acceso a un fichero es necesario capturar dos excepciones, para el caso de que no exista el fichero al que queramos acceder y por si se produce un error en la E/S.

Para la escritura podemos utilizar el método anterior, aunque muchas veces nos resultará mucho más cómodo utilizar un objeto `PrintWriter` con el que podamos escribir directamente líneas de texto:

```
public void escribe_fichero() {
  FileWriter out = null;
  PrintWriter p_out = null;

  try {
    out = new FileWriter("result.txt");
    p_out = new PrintWriter(out);
    p_out.println(
      "Este texto será escrito en el fichero de salida");

  } catch(IOException e) {
    System.err.println("Error al escribir en el fichero");
  } finally {
    p_out.close();
  }
}
```

1.4. Acceso a los recursos

Hemos visto como leer y escribir ficheros, pero cuando ejecutamos una aplicación contenida en un fichero JAR, puede que necesitemos leer recursos contenidos dentro de este JAR.

Para acceder a estos recursos deberemos abrir un flujo de entrada que se encargue de leer su contenido. Para ello utilizaremos el método `getResourceAsStream` de la clase `Class`:

```
InputStream in = getClass().getResourceAsStream("/datos.txt");
```


Copyright © 2008-2009 Depto. CCIA All rights reserved.

De esta forma podremos utilizar el flujo de entrada obtenido para leer el contenido del fichero que hayamos indicado. Este fichero deberá estar contenido en el JAR de la aplicación.

Especificamos el carácter '`/`' delante del nombre del recurso para referenciarlo de forma relativa al directorio raíz del JAR. Si no lo especificásemos de esta forma se buscaría de forma relativa al directorio correspondiente al paquete de la clase actual.

1.5. Codificación de datos

Si queremos guardar datos en un fichero binario deberemos codificar estos datos en forma de *array* de *bytes*. Los flujos de procesamiento `DataInputStream` y `DataOutputStream` nos permitirán codificar y descodificar respectivamente los tipos de datos simples en forma de *array* de *bytes* para ser enviados a través de un flujo de datos.

Por ejemplo, podemos codificar datos en un *array* en memoria (`ByteArrayOutputStream`) de la siguiente forma:

```
String nombre = "Jose";
String edad = 25;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);

dos.writeUTF(nombre);
dos.writeInt(edad);
dos.close();
baos.close();

byte [] datos = baos.toByteArray();
```

Podremos descodificar este *array* de *bytes* realizando el procedimiento inverso, con un flujo que lea un *array* de *bytes* de memoria (`ByteArrayInputStream`):

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = new DataInputStream(bais);
String nombre = dis.readUTF();
int edad = dis.readInt();
```

Si en lugar de almacenar estos datos codificados en una *array* en memoria queremos guardarlos codificados en un fichero, haremos lo mismo simplemente sustituyendo el flujo canal de datos `ByteArrayOutputStream` por un `FileOutputStream`. De esta forma podremos utilizar cualquier canal de datos para enviar estos datos codificados a través de él.

1.6. Serialización de objetos

Si queremos enviar un objeto a través de un flujo de datos, deberemos convertirlo en una serie de *bytes*. Esto es lo que se conoce como serialización de objetos, que nos permitirá leer y escribir objetos.

Copyright © 2008-2009 Depto. CCIA All rights reserved.

Para leer o escribir objetos podemos utilizar los objetos `ObjectInputStream` y `ObjectOutputStream` que incorporan los métodos `readObject` y `writeObject` respectivamente. Los objetos que escribamos en dicho flujo deben tener la capacidad de ser *serializables*.

Serán *serializables* aquellos objetos que implementan la interfaz `Serializable`. Cuando queramos hacer que una clase definida por nosotros sea *serializable* deberemos implementar dicho interfaz, que no define ninguna función, sólo se utiliza para identificar las clases que son *serializables*. Para que nuestra clase pueda ser *serializable*, todas sus propiedades deberán ser de tipos de datos básicos o bien objetos que también sean *serializables*.

Un uso común de la serialización se realiza en los *Transfer Objetcs*. Este tipo de objetos deben ser serializables para así poderse intercambiar entre todas las capas de la aplicación, aunque se encuentren en máquinas diferentes.

2. Red

En este apartado veremos como establecer una comunicación en red mediante URLs. En Java también se pueden establecer conexiones de red a más bajo nivel mediante *sockets* (protocolos TCP y UDP), pero muchas veces los puertos a los que accedemos se encuentran cortados por *firewalls* intermedios, por lo que estos tipos de conexiones no serán adecuados para acceder a servidores en Internet.

Además las APIs de Java nos ofrecen numerosas facilidades para trabajar con URLs que sigan protocolos estándar (como por ejemplo HTTP), tanto en el lado del cliente como en el servidor, por lo que será recomendable establecer la comunicación entre nuestro cliente y nuestro servidor utilizando estos protocolos.

Java nos permite además utilizar mecanismos de comunicación por red de más alto nivel como por ejemplo RMI que nos permite invocar métodos de objetos remotos. Esto lo veremos con más detalle en módulos posteriores.

2.1. Acceso a URLs

Una URL (*Uniform Resource Locator*) es una cadena utilizada para localizar un recurso en Internet. Dentro de la URL podemos distinguir varias componentes:

```
protocolo://servidor[:puerto]/recurso
```

Por ejemplo, en el caso de la dirección `http://www.ua.es/es/index.html` lo que se hará será acceder al servidor `www.ua.es` mediante protocolo HTTP y solicitar el recurso `/es/index.html`. El puerto por defecto es el `80`, pero si el servidor Web atendiese en un puerto distinto a este deberíamos especificarlo también en la URL.

En Java tenemos el objeto `URL` que se encargará de representar las URLs. Podemos

Copyright © 2008-2009 Depto. CCIA All rights reserved.

construir un objeto `URL` a partir del nombre completo de la URL:

```
URL url = new URL("http://www.ua.es/es/index.html");
```

Dado que muchas veces se especifican links relativos, será de ayuda contar con un segundo constructor que nos permita crear URLs a partir de la dirección base donde nos encontremos y de la dirección relativa solicitada:

```
URL url = new URL(direccion_base, direccion_relativa);
```

Aquí la dirección relativa puede referirse a un recurso alojado en el servidor donde nos encontremos o bien a un destino dentro de la web donde estamos, referenciado mediante `#nombre_destino`.

Existen más constructores de esta clase, permitiéndonos por ejemplo construir una URL dando cada elemento (protocolo, servidor, puerto, recurso) por separado. Siempre que creemos una URL deberemos capturar la excepción `MalformedURLException` que se producirá en el caso de estar mal construida.

```
try {
        URL url = new URL("http://www.ua.es/es/index.html");
} catch(MalformedURLException e) {
        System.err.println("Error: URL mal construida");
}
```

La clase `URL` proporciona métodos para obtener información sobre la URL que representa.

2.2. Lectura del contenido

Para leer desde la dirección URL representada por el objeto deberemos obtener un flujo de entrada provinente de ella. Para obtener este flujo utilizaremos el método `openStream` del objeto `URL`.

```
InputStream in = url.openStream();
```

Una vez obtenido este flujo de entrada podremos leer de él o bien transformarlo a otro tipo de flujo como por ejemplo a un flujo de caracteres o de procesamiento.

Esto puede ser suficiente en muchos casos, en los que sólo necesitemos leer el contenido de un recurso localizado mediante dicha URL. Por ejemplo, si la URL corresponde a una página web, al leer de ella obtendremos el código de dicha página web.

Sin embargo, muchas veces necesitaremos poder enviar información al servidor y acceder a una serie de propiedades sobre la respuesta que hemos obtenido del mismo. Para hacer esto deberemos crear una conexión con la URL, y utilizar esta conexión para enviar o recibir información.

2.3. Conexión con la URL

Copyright © 2008-2009 Depto. CCIA All rights reserved.

Podemos crear una conexión con la URL utilizando el método `openConnection` del objeto `URL` que nos devolverá un objeto del tipo `URLConnection`. Estableciendo una conexión podremos leer o escribir datos en la URL.

La clase `URLConnection` es una clase abstracta, que define de forma genérica una conexión con una URL cualquiera. En realidad el objeto que habremos obtenido será una subclase de `URLConnection` especializada en el protocolo con el que estemos trabajando. Por ejemplo si trabajamos con HTTP, en realidad habremos obtenido un objeto `HttpURLConnection`.

Cuando accedamos a una URL, primero se enviará un mensaje de petición al servidor al que hace referencia la URL para solicitar la información deseada. En este mensaje de petición se podrán incluir una serie de propiedades con información y un bloque de contenido que queramos enviar al servidor. En estas propiedades que incluimos como cabecera podremos especificar por ejemplo información sobre el agente de usuario, como el tipo de cliente que está accediendo al servidor y el idioma de este cliente. Además podemos incluir cualquier contenido que queramos al cuerpo del mensaje.

Una vez se envíe el mensaje de petición al servidor, éste nos devolverá un mensaje de respuesta. En esta respuesta podremos encontrar también una serie de cabeceras con información sobre la misma, y un bloque de contenido que anteriormente hemos visto cómo leer.

Todo este proceso de composición del mensaje de petición, envío del mensaje, recepción del mensaje de respuesta, y análisis del mismo lo realiza internamente la clase `URLConnection`. Nosotros podremos utilizar esta clase para configurar el mensaje de petición y obtener la información del mensaje de respuesta de forma sencilla.

Una vez creada la conexión, ésta pasará por tres estados:

- **Configuración**: No se ha establecido la conexión, todavía no se ha enviado el mensaje de petición. Este será el momento en el que deberemos añadir la información necesaria a las cabeceras del mensaje de petición.
- **Conectada**: El mensaje de petición ya se ha enviado, y se espera recibir una respuesta. En este momento podremos leer las cabeceras o el contenido de la respuesta.
- **Cerrada**: La conexión se ha cerrado y ya no podemos hacer nada con ella.

La conexión nada más crearse se encuentra en estado de configuración. Pasará automáticamente a estado conectada cuando solicitemos cualquier información sobre la respuesta.

Enviar o recibir datos

Para enviar o recibir el contenido de los mensajes de petición y respuesta respectivamente deberemos hacer lo siguiente:

- Crear la conexión

Copyright © 2008-2009 Depto. CCIA All rights reserved.

```
URLConnection con = url.openConnection();
```

- Si vamos a enviar datos a través de ella, establecer la capacidad de salida con:

```
con.setDoOutput(true);
```

- Si vamos a escribir en la petición, obtener el flujo de salida con:

```
OutputStream out = con.getOutputStream();
```

- Si vamos a leer la respuesta, obtener el flujo de entrada con:

```
InputStream in = con.getInputStream();
```

- Leer o escribir en los flujos de entrada y de salida obtenidos como vimos en capítulos anteriores.

Al obtener los flujos para escribir o leer en la conexión estaremos haciendo que pase a estado conectado, por lo que no podremos continuar configurando las propiedades de la petición después de hacer esto. Estas propiedades de configuración las deberemos establecer previamente a la apertura de estos flujos, como veremos a continuación.

Mensaje de petición

En fase de configuración podremos añadir una serie de propiedades al mensaje de petición para configurarlo. Estas propiedades serán parejas <*clave, valor*>. Las añadiremos con el siguiente método:

```
con.setRequestProperty(nombre, valor);
```

Por ejemplo, podemos mandar las siguiente cabeceras:

```
con.setRequestProperty("IF-Modified-Since",
        "22 Sep 2002 08:00:00 GMT");
con.setRequestProperty("Content-Language", "es-ES");
```

Con esto estaremos diciendo al servidor que queremos que nos devuelva una respuesta sólo si ha sido modificada desde la fecha indicada, y además le estamos comunicando datos sobre el cliente, como que el lenguaje del cliente es español de España.

Cabeceras de la respuesta

Una vez hayamos terminado de configurar el mensaje de petición, podemos pasar a estado conectado. En este estado, además de leer el contenido del mensaje de respuesta, podremos obtener información sobre esta respuesta consultando una serie de cabeceras incluidas en este mensaje.

También podemos utilizar este objeto para leer las cabeceras que nos ha devuelto la respuesta. Nos ofrece métodos para leer una serie de cabeceras estándar de HTTP como los siguientes:

Copyright © 2008-2009 Depto. CCIA All rights reserved.

`getLength`	`content-length`	Longitud del contenido, o `-1` si la longitud es desconocida
`getType`	`content-type`	Tipo MIME del contenido devuelto
`getEncoding`	`content-encoding`	Codificación del contenido
`getExpiration`	`expires`	Fecha de expiración del recurso
`getDate`	`date`	Fecha de envío del recurso
`getLastModified`	`last-modified`	Fecha de última modificación del recurso

Puede ser que queramos obtener otras cabeceras, como por ejemplo cabeceras propias no estándar. Para ello tendremos una serie de métodos que obtendrán las cabeceras directamente por su nombre:

```
String valor = con.getHeaderField(nombre);
int valor = con.getHeaderFieldInt(nombre, default);
long valor = con.getHeaderFieldDate(nombre, default);
```

De esta forma podemos obtener el valor de la cabecera o bien como una cadena, o en los datos que sean de tipo fecha (valor `long`) o enteros también podremos obtener su valor directamente en estos tipos de datos.

Podremos acceder a las cabeceras también a partir de su índice:

```
String valor = con.getHeaderField(int indice);
String nombre = con.getHeaderFieldKey(int indice);
```

Podemos obtener de esta forma tanto el nombre como el valor de la cabecera que ocupa un determinado índice.

Tanto los métodos que obtienen un flujo para leer o escribir en la conexión, como estos métodos que acabamos de ver para obtener información sobre la respuesta producirán una transición al estado conectado.

Ejemplo

Vamos a ver un ejemplo de como enviar datos al servidor, y posteriormente leer la respuesta. Cuando enviemos datos será conveniente proporcionar el tamaño y el tipo MIME de este contenido para que pueda ser interpretado correctamente. Estos datos se incluirán como propiedades de la petición.

```
// Creamos la URL
URL url = new URL("http://jtech.ua.es/chat/enviar");

// Creamos la conexion
URLConnection con = url.openConnection();

// Activamos la salida de datos en la conexión
con.setDoOutput(true);
```


Copyright © 2008-2009 Depto. CCIA All rights reserved.

```
// Escribimos los datos en un buffer en memoria
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
dos.writeUTF(nick);
dos.writeUTF(msg);
dos.close();

// Establecemos las propiedades de tipo y tamaño del contenido
con.setRequestProperty("Content-Length",
String.valueOf(baos.size()));
con.setRequestProperty("Content-Type", "application/octet-stream");


// Abrimos el flujo de salida para enviar los datos al servidor
OutputStream out = con.getOutputStream();
baos.writeTo(out);

// Abrimos el flujo de entrada para leer la respuesta obtenida
InputStream in = con.getInputStream();
```

3. Javamail

3.1. Introducción

JavaMail proporciona un conjunto de clases abstractas que definen los objetos e interfaces necesarias para implementar un sistema de e-mail.Los proveedores JavaMail que implementan el API proporcionan la funcionalidad necesaria para establecer la comunicación a través de protocolos concretos. Concretamente, la implementación de Sun permite manejar los protocolos SMTP (Simple Mail Transfer Protocol), IMAP (Internet Message Access Protocol), MIME (Multipurspose Internet Mail Extensions) y Post Office Protocol 3 (POP3).

En terminología JavaMail, usando los protocolos podemos implementar o bien un **Transport** o un **Store**. El primero se refiere a un servicio con capacidad para enviar mensajes a su destino (usualmente con SMTP), mientras que el segundo es un servicio con el que hay conectar para descargar mensajes que han sido enviados a nuestro buzón (p.e. POP3 o IMAP) .

3.2. Instalación y prueba

En cuanto a la instalación, lo primero es descargar la extensión JavaMail de java.sun.com/products/javamail. La última versión es la 1.4.1. A continuación copiaremos el fichero `mail.jar`. en el directorio de extensiones *$JAVA_HOME/jre/lib/ext*, o bien como librería de nuestro proyecto. Seguidamente instalaremos JAF (JavaBeans

Copyright © 2008-2009 Depto. CCIA All rights reserved.

Activation Framework) tras descargarlo de java.sun.com/beans/glasgow/jaf.html. Copiaremos el fichero `activation.jar` en el directorio de extensiones.

> **Nota**
> JAF ya viene incluido en JDK 1.6, de forma que si utilizamos esta versión de JDK, o una posterior, no será necesario instalar JAF por separado.

3.3. Configuración de la sesión

La clase **Session** representa una sesión para el envío de mensajes. Podemos configurar esta sesión con datos de la conexión (protocolo, host o servidor de correo, puerto, etc), que se proporcionarán mediante un objeto **java.util.Properties**.

```
Properties props = new Properties();
props.put("mail.transport.protocol",  "smtp");
props.put("mail.smtp.host", "mail.alu.ua.es");
props.put("mail.smtp.port", "25");
Session session = Session.getInstance(props);
```

Un forma alternativa de crear la sesión es:

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

en la que se crea una sesión compartida, mientras que usando **getInstance()** creamos una sesión no-compartida. El argumento **null** es un autentificador que veremos después. La diferencia básica entre una sesión compartida y otra no-compartida es que en el primer caso, cuando llamamos de nuevo para crear una sesión de este tipo los parámetros que les pasemos en esta ocasión son ignorados y se devuelve la misma sesión. Es habitual compartir sesión si todos los clientes, p.e., utilizan un mismo servidor SMTP. En este sentido la sesión puede compartirse por múltiples clases que se estén ejecutando simultáneamente en la MV.

También podemos activar el modo depuración en la sesión mediante el método `session.setDebug(true)`.

3.4. Composición de los mensajes

Nos ocupamos ahora de introducir los mecanismos básicos para la construcción de los mensajes (más tarde volveremos sobre este punto para tratar la construcción de mensajes más complejos, p.e. con attachments). La clase **Message** es una clase abstracta que proporciona un contenedor para mensajes, esto es, permite construir mensajes con todos sus atributos. En términos generales, un mensaje consta de una cabecera (con información como el asunto, emisor, receptor/es, etc) y un contenido propiamente dicho.

Lo más habitual es utilizar la subclase **MimeMessage** que permite entender tipos MIME en donde los encabezados no se restringen a caracteres ASCII. Hay dos tipos de

Copyright © 2008-2009 Depto. CCIA All rights reserved.

constructor

```
Message msg = new MimeMessage(session);
Message msgBis = new MimeMessage(msg);
```

En el primer caso se obtiene un mensaje vacío a partir de los datos de la sesión, mientras que el segundo (constructor de copia) se crea una nueva instancia con las mismas propiedades y contenido que el mensaje que se le pasa por argumento.

Una vez creado el mensaje, p.e. vacío, el método **setFrom()** fija la dirección origen del mensaje (from) haciendo uso de objetos de la clase de direcciones **javax.mail.internet.InternetAddress** sublclase de la clase abstracta **javax.mail.Address**.

```
if (from != null)
    msg.setFrom(new InternetAddress(from));
else
    msg.setFrom();
```

El método **setRecipients()** especifica los receptores del mensaje (to, cc, bcc):

```
msg.setRecipients(Message.RecipientType.TO,
InternetAddress.parse(to, false));
```

EL método **setSubject()** establece el asunto del mensaje.

```
msg.setSubject(subject);
```

Para rellenar el contenido del mensaje hay que tener en cuenta si se trata de un mensaje de solo-de-texto o no. En caso de que sea solo texto se dispone del método **setText()** que recibe una cadena de entrada. Si como en el ejemplo el contenido se lee de un stream entonces esta cadena se va construyendo a medida que se leen las líneas y una vez terminado el stream se hace la llamada al método. Si por el contrario estamos incluyendo tipos MIME, como HTML en el contenido, utilizaremos el método **setContent()**:

```
String contenido = "<HTML><HEAD><TITLE> Hola</TITLE></HEAD><BODY>
Texto<BODY></HTML>"
msg.setContent(contenido, "text/html");
```

Finalmente, para fijar el encabezado hacemos uso de **setHeader()**, inclueyendo el nombre de la clase java desde la que se envía el mail, y para incluir la fecha de envío usaremos **setSentDate()**:

```
String mailer = "Java Mailer";
...
msg.setHeader("X-Mailer", mailer);
msg.setSentDate(new Date());
```

Otra funcionalidad es establecer la dirección a la que responder. Usaremos el método **setReplyTo()**, al que le pasamos un array de direcciones.

```
Adress[] direcciones = ....
msg.setReplyTo(direcciones);
```


Copyright © 2008-2009 Depto. CCIA All rights reserved.

Finalmente, hay que reseñar que los métodos **setXXX** tienen sus contrapartidas **getXXX** como veremos más adelante, salvo **getMessageID()** que devuelve el identificador de un mensaje lo cual es bastante útil para gestionar jerarquías de mensajes.

3.5. Envío de mensajes

Una vez construido el mensaje utilizamos la clase **Transport** para enviarlo, normalmente a través del protocolo SMTP. Para ello utilizamos los métodos **send()** al cual le pasamos o bien un único argumento (el objeto **Message** en cuestión) o bien dicho objeto seguido de un array de direcciones (objetos **Address**) con lo que se sustituyen los destinatarios previamente establecidos.

```
Transport.send(msg);
```

Alternativamente podemos utilizar un objeto **Transport**, obtenido a partir de la sesión actual, y llamar a **sendMessage()**:

```
msg.saveChanges(); // send() incluye una llamada a este método.
Transport tr = session.getTransport("smtp");
tr.connect(host, usuario, password);
tr.sendMessage(msg, msg.getAllRecipients());
tr.close();
```

lo cual tiene la ventaja de que aprovechamos la misma conexión al servidor para enviar muchos mensajes, mientras que con **send()** se establece una conexión diferente para cada uno de ellos.

Si el servidor de correo saliente utiliza SMTP con SSL, debemos activar previamente TLS en la configuración de la sesión con:

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

3.6. Mensajes con adjuntos

Hasta ahora hemos hecho referencia al contenido de los mensajes sin entrar en detalle en el complejo modelo de contenido utilizado en JavaMail. Pero conocer dicho modelo es necesario si queremos manejar attachments.

En términos generales, un mensaje está compuesto de múltiples partes, cada una de las cuales es una **javax.mail.BodyPart** o una **javax.mail.internet.MimeBodyPart** y todas las partes pueden combinarse en un contenedor llamado **javax.mail.MultiPart** (o bien un **javax.mail.internet.MimeMultiPart**).

Veamos el código de una aplicación sencilla para enviar un attachment. Lo primero, tras crear la sesión es definir el mensaje:

```
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
```


Copyright © 2008-2009 Depto. CCIA All rights reserved.

```
message.addRecipient(Message.RecipientType.TO, new
InternetAddress(to));
message.setSubject("Mensaje con adjuntos");
```

Seguidamente, creamos la parte de texto y la rellenamos

```
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Aqui te envio el fichero");
```

Entonces creamos un contenedor y le añadimos el cuerpo de texto

```
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);
```

Entonces comenzamos a construir la parte que contendrá el attachment:

```
messageBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);
```

Finalmente se pone el contenido del mensaje y se envía:

```
message.setContent(multipart);
Transport.send(message);
```

Si, por ejemplo queremos enviar un mensaje HTML de tal forma que a su vez referencia a una imagen, deberemos hacer los siguiente, una vez hayamos abierto la sesión, creado el mensaje e inicializado los encabezamientos:

1. Crear una **BodyPart** para el contenido HTML y rellenar este contenido (incluyendo la referencia a la imágen).

```
BodyPart messageBodyPart = new MimeBodyPart();
String htmlText = "<H1>Hola</H1>" +
      "<img src=\"cid:imagen.gif\">";
messageBodyPart.setContent(htmlText, "text/html");
```

2. Crear un **MultiPart** para combinar para que contenga por un lado la parte HTML y por otro la imágen que se incorporará como attachment, e incorporarle la parte HTML:

```
MimeMultipart multipart = new MimeMultipart("related");
multipart.addBodyPart(messageBodyPart);
```

3. Crear el **BodyPart** para el attachment y asociarla a un **DataHandler**:

```
messageBodyPart = new MimeBodyPart();
DataSource fds = new FileDataSource(file);
messageBodyPart.setDataHandler(new DataHandler(fds));
```

4. Añadir una cabecera para el HTML:

```
messageBodyPart.setHeader("Content-ID","<imagen.gif>");
```

5. Añadir la parte del attachment a la **MultiPart**, asociar esta al mensaje y enviarlo:

```
multipart.addBodyPart(messageBodyPart);
```


Copyright © 2008-2009 Depto. CCIA All rights reserved.

```
message.setContent(multipart);
Transport.send(message);
```

3.7. Autentificación

Las clases **Authenticator** y **PasswordAuthenticator** nos permiten controlar el acceso al servidor de mail. La primera es una clase abstracta por lo que utilizamos la segunda para crear instancias. Concretamente, la autentificación tiene lugar al construir el objeto sesión. De hecho el segundo parámetro que se le pasa al constructor de la sesión es un objeto de una subclase de **Authenticator** que debe implementar el método **getPasswordAuthentication()** que devuelve un objeto **PasswordAuthentication**:

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

public class Autentificador extends Authenticator {

  public PasswordAuthentication getPasswordAuthentication() {
    String username, password;

    Scanner scan = new Scanner(System.in);
    System.out.print("Usuario: ");
    username = scan.nextLine();
    System.out.print("Password: ");
    password = scan.nextLine();

    return new PasswordAuthentication(username, password);
  }

}
```

En este método incluimos el código necesario para solicitar el password. Posteriormente antes de crear una sesión haremos lo siguiente

```
...
Authenticator aut = new Autentificador();
Session session = Session.getDefaultInstance(props, aut);
```

Si queremos, por ejemplo, autentificarnos con el servidor de salida, deberemos especificarlo en la configuración antes de crear la sesión:

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

3.8. Recepción de mensajes y gestión de carpetas

Hasta ahora hemos visto cómo enviar mensajes. Para leerlos del servidor empezaremos definiendo un objeto sesión, y a partir de él llamaremos al método **getStore()** indicando el protocolo de lectura (p.e. pop3 o imap). Seguidamente conectaremos, a través del método **connect()** con el servidor de mail una vez hayamos indicado el host (p.e. mail.alu.ua.es) el usuario y su password correspondiente:

Copyright © 2008-2009 Depto. CCIA All rights reserved.

```
Store store = session.getStore("pop3");
store.connect(host, username, password);
```

Seguidamente, se obtiene una o varias carpetas (objetos **Folder**) y se accede a los mensajes:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
...
```

Dentro de las carpetas podremos realizar operaciones como buscar mensajes, copiar o mover mensajes a otra carpeta, etc.

Copyright © 2008-2009 Depto. CCIA All rights reserved.

Copyright © 2008-2009 Depto. CCIA All rights reserved.