Capitolo 17

Networking

Lorenzo Bettini

Introduzione

Si sente spesso affermare che Java è "il linguaggio di programmazione per Internet". Effettivamente la maggior parte del grande successo e della diffusione di Java è dovuta a questo, vista soprattutto l'importanza sempre maggiore che Internet sta assumendo. Java è quindi particolarmente adatto per sviluppare applicazioni che devono fare uso della rete. Ciò non deve indurre a pensare che con Java si scrivono principalmente solo Applet, per animare e rendere più carine e interattive le pagine web. Con Java si possono sviluppare vere e proprie applicazioni che devono girare in rete interagendo con più computer (le cosiddette *applicazioni distribuite*).

Non si dimentichi che un altro fattore determinante per il suo successo è l'indipendenza dalla piattaforma, ottenuta grazie all'utilizzo del bytecode. Il linguaggio astrae da problemi di portabilità come il byte ordering, e quindi anche il programmatore non deve preoccuparsi dei classici problemi di interoperabilità cross-platform.

A questo punto il programmatore di una applicazione network based non deve preoccuparsi di scrivere ex novo particolari librerie o funzioni per le operazioni di base, ma può dedicarsi totalmente ai dettegli veri e propri dell'applicazione.

Inoltre ciò che rende Java un linguaggio adatto per il networking sono le classi definite nel pacchetto java.net che sarà analizzato in questo capitolo, in cui, oltre alla descrizione delle varie classi e dei rispettivi metodi, saranno forniti anche semplici esempi estendibili e funzionanti.

Socket

Le classi di networking incapsulano il paradigma *socket* presentato per la prima volta nella Berkeley Software Distribution (BSD) della University of California at Berkeley.

Una socket è come una porta di comunicazione e non è molto diversa da una presa elettrica: tutto ciò che è in grado di comunicare tramite il protocollo standard TCP/IP può collegarsi ad una socket e comunicare tramite questa porta, allo stesso modo in cui un qualsiasi apparecchio che funziona a corrente può collegarsi a una presa elettrica e sfruttare la tensione messa a disposizione. Nella "rete" gestita dalle socket, invece dell'elettricità, viaggiano pacchetti TCP/IP. Tale protocollo e le socket forniscono quindi un'astrazione che permette di far comunicare dispositivi diversi che utilizzano lo stesso protocollo.

Quando si parla di networking, ci si imbatte spesso nel termine *client-server*. Si tratta in realtà di un paradigma: un'entità (spesso un programma) *client* per portare a termine un particolare compito richiede dei servizi ad un'altra entità (anche questa spesso è un programma): un *server* che ha a disposizione delle risorse da condividere. Una tale situazione si ritrova spesso nell'utilizzo quotidiano dei computer (anche senza saperlo): un programma che vuole stampare qualcosa (client) richiede alla stampante (server) l'utilizzo di tale risorsa.

Il server è una risorsa costantemente disponibile, mentre il client è libero di scollegarsi dopo che è stato servito. Tramite le socket inoltre un server è in grado di servire più client contemporaneamente.

Alcuni esempi di client-server molto noti sono:

Telnet : se sulla nostra macchina si ha disposizione il programma Telnet (programma client), è possibile operare su un computer remoto come si opera su un computer locale. Questo è possibile se sulla macchina remota è presente un programma server in grado di esaudire le richieste del client Telnet;

FTP : tramite un client FTP si possono copiare e cancellare files su un computer remoto, purché qui sia presente un server FTP;

Web : il browser è un client web, che richiede pagine web ai vari computer su cui è installato un web server, che esaudirà le richieste spedendo la pagina desiderata.

Come si è detto, tipicamente, sia il server che il client sono dei programmi che possono girare su macchine diverse collegate in rete. Il client deve conoscere l'indirizzo del server e il particolare protocollo di comunicazione utilizzato dal server. L'indirizzo in questione è un classico *indirizzo IP*.

Un client, quindi, per comunicare con un server usando il protocollo TCP/IP dovrà per prima cosa creare una socket con tale server, specificando l'indirizzo IP della macchina su cui il server è in esecuzione e il numero di porta sulla quale il server è in ascolto. Il concetto di *porta* permette ad un singolo computer di servire più client contemporaneamente: su uno stesso computer possono essere in esecuzione server diversi, in ascolto su porte diverse. Se si vuole un'analogia si può pensare al fatto che più persone abitano nella medesima via, ma a numeri civici diversi. In questo caso i numeri civici rappresenterebbero le porte.

Un server "rimarrà in ascolto" su una determinata porta finché un client non creerà una socket con la macchina del server, specificando quella porta. Una volta che il collegamento con il server, tramite la socket è avvenuto, il client può iniziare a comunicare con il server, sfruttando la socket creata. A collegamento avvenuto si instaura un protocollo di livello superiore che dipende da quel particolare server: il client deve utilizzare quel protocollo di comunicazione, per richiedere servizi al server.

Il numero di porta è un intero compreso fra 1 e 65535. Il TCP/IP riserva le porte minori di 1024 a servizi standard. Ad esempio la porta 21 è riservata all'FTP, la 23 al Telnet, la 25 alla posta elettronica, la 80 all'HTTP (il protocollo delle pagine web), la 119 ai news server. Si deve tenere a mente che una porta in questo contesto non ha niente a che vedere con le porte di una macchina (porte seriali, parallele, ecc.), ma è un'astrazione utile per smistare informazioni a più server in esecuzione su una stessa macchina.

Si presentano adesso le classi messe a disposizione da Java nel pacchetto java.net per la gestione di comunicazioni in rete.

La classe InetAddress

Come si sa, un indirizzo Internet è costituito da 4 numeri (da 0 a 255) separati ciascuno da un punto. Spesso però, quando si deve accedere a un particolare host, invece di specificare dei numeri, si utilizza un nome, che corrisponde a tale indirizzo (p.e.: www.myhost.it). La traduzione dal nome all'indirizzo numerico vero e proprio è compito del servizio *Domain Name Service*, abbreviato con DNS.

Senza entrare nei dettagli di questo servizio, basti sapere che la classe InetAddress mette a disposizione diversi metodi per astrarre dal particolare tipo di indirizzo specificato (a numeri o a lettere), occupandosi essa stessa di effettuare le dovute traduzioni.

Inoltre c'è un ulteriore vantaggio: la scelta di utilizzare un indirizzo numerico a 32 bit non fu a suo tempo una scelta molto lungimirante; con l'immensa diffusione che Internet ha avuto e sta avendo, si è molto vicini ad esaurire tutti i possibili indirizzi che si possono ottenere con 32 bit (oltretutto diversi indirizzi sono riservati e quindi il numero di indirizzi possibili si riduce ulteriormente); si stanno pertanto introducendo degli indirizzi a 128 bit che, da questo punto di vista, non dovrebbero più dare tali preoccupazioni.

Le applicazioni che utilizzeranno indirizzi Internet tramite la classe InetAddress saranno portabili dal punto di vista degli indirizzi, in modo completamente trasparente.

Descrizione classe

```
public final class InetAddress extends Object implements Serializable
```

Costruttori

La classe non mette a disposizione nessun costruttore: l'unico modo per creare un oggetto InetAddress prevede l'utilizzo di metodi statici, descritti di seguito.

Metodi

```
public static InetAddress getByName(String host) throws UnknownHostException
```

Restituisce un oggetto InetAddress rappresentante l'host specificato nel parametro host. L'host può essere specificato sia come nome, che come indirizzo numerico. Se si specifica null come parametro, ci si riferisce all'indirizzo di default della macchina locale.

```
public static InetAddress[] getAllByName(String host) throws UnknownHostException
```

Tale metodo è simile al precedente, ma restituisce un array di oggetti InetAddress: spesso alcuni siti web molto trafficati registrano lo stesso nome con indirizzi IP diversi. Con questo metodo si otterranno tanti InetAddress quanti sono questi indirizzi registrati.

```
public static InetAddress getLocalHost() throws UnknownHostException
```

Viene restituito un InetAddress corrispondente alla macchina locale. Se tale macchina non è registrata, oppure è protetta da un firewall, l'indirizzo è quello di loopback: 127.0.0.1.

Tutti questi metodi possono sollevare l'eccezione UnknownHostException se l'indirizzo specificato non può essere risolto (tramite il DNS).

```
public String getHostName()
```

Restituisce il nome dell'host che corrisponde all'indirizzo IP dell'InetAddress. Se il nome non è ancora noto (ad esempio se l'oggetto è stato creato specificando un indirizzo IP numerico), verrà cercato tramite il DNS; se tale ricerca fallisce, verrà restituito l'indirizzo IP numerico (sempre sotto forma di stringa).

```
public String getHostAddress()
```

Simile al precedente: restituisce però l'indirizzo IP numerico, sotto forma di stringa, corrispondente all'oggetto InetAddress.

```
public byte[] getAddress()
```

L'indirizzo IP numerico restituito sarà sotto forma di array byte. L'ordinamento dei byte è *high byte first* (che è proprio l'ordinamento tipico della rete).

Un'Applet potrà costruire un oggetto InetAddress solo per l'host dove si trova il web server dal quale l'Applet è stata scaricata, altrimenti verrà generata un'eccezione: SecurityException.

Un esempio

Con tale classe a disposizione è molto semplice scrivere un programma in grado di tradurre nomi di host nei corrispettivi indirizzi numerici e viceversa. Al programma che segue basterà

passare una stringa contenente o un nome di host o un indirizzo IP numerico e si avranno in risposta le varie informazioni.

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

public class HostLookup {
 public static void main(String args[]) {
    // prima si stampano i dati relativi
    // alla macchina locale...
    try {
       InetAddress LocalAddress = InetAddress.getLocalHost();
       System.out.println("host locale : "
                          + LocalAddress.getHostName() + ", IP : "
                          + LocalAddress.getHostAddress());
    } catch(UnknownHostException e) {
       System.err.println("host locale sconosciuto!");
       e.printStackTrace();
    }

    // ...poi quelli dell'host specificato
    if(args.length != 1) {
       System.err.println("Uso: HostLookup host");
    } else {
       try {
          System.out.println("Ricerca di " + args[0] + "...");
          InetAddress RemoteMachine = InetAddress.getByName(args[0]);
          System.out.println("Host Remoto : "
+ RemoteMachine.getHostName() + ", IP : "
+ RemoteMachine.getHostAddress() );
       } catch(UnknownHostException ex) {
          System.out.println("Ricerca Fallita " + args[0]);
       }
    }
 }
}
```

URL

Tramite un URL (*Uniform Resource Locator*) è possibile riferirsi alle risorse di Internet in modo semplice e uniforme. Si ha così a disposizione una forma intelligente e pratica per identificare o indirizzare in modo univoco le informazioni su Internet. I browser utilizzano gli URL per recuperare le pagine web. Java mette a disposizione alcune classi per utilizzare gli URL; sarà così possibile, ad esempio, inglobare nelle proprie applicazioni funzioni tipiche dei web browser.

Esempi tipici di URL sono:

```
http://www.myweb.com:8080/webdir/webfile.html
ftp://ftp.myftpsite.edu/pub/programming/tips.tgz
```

Un URL consiste di 4 componenti:

1. il protocollo separato dal resto dai due punti (esempi tipici di protocolli sono http, ftp, news, file, ecc.);

2. il nome dell'host, o l'indirizzo IP dell'host, che è delimitato sulla sinistra da due barre (//), e sulla destra da una sola (/), oppure da due punti (:);

3. il numero di porta, separato dal nome dell'host sulla sinistra dai due punti, e sulla destra da una singola barra. Tale componente è opzionale, in quanto, come già detto, ogni protocollo ha una porta di default;

4. il percorso effettivo della risorsa che richiediamo. Il percorso viene specificato come si specifica un path sotto Unix. Se non viene specificato nessun file, la maggior parte dei server HTTP aggiunge automaticamente come file di default index.html.

Descrizione classe

```
public final class URL extends Object implements Serializable
```

Costruttori

La classe ha molti costruttori, poiché vengono considerati vari modi di specificare un URL.

```
public URL(String spec) throws MalformedURLException
```

L'URL viene specificato tramite una stringa, come ad esempio:

```
http://www.myweb.it:80/foo.html
```

```
public URL(URL context, String spec) throws MalformedURLException
```

L'URL viene creato combinando un URL già esistente e un URL specificato tramite una stringa. Se la stringa è effettivamente un URL assoluto, allora l'URL creato corrisponderà a tale stringa; altrimenti l'URL risultante sarà il percorso specificato in spec, relativo all'URL context. Ad esempio se context è http://www.myserver.it/ e spec è path/index.html, l'URL risultante sarà http://www.myserver.it/path/index.html.

```
public URL(String protocol, String host, int port,String file) throws MalformedURLException
```

Con questo costruttore si ha la possibilità di specificare separatamente ogni singolo componente di un URL.

```
public URL(String protocol, String host, String file) throws MalformedURLException
```

Simile al precedente, ma viene usata la porta di default del protocollo specificato.

Un'eccezione MalformedURLException viene lanciata se l'URL non è specificato in modo corretto (per quanto riguarda i vari componenti).

Metodi

Di questa classe fanno parte diversi metodi che permettono di ricavare le varie parti di un URL.

```
public int getPort()
public String getProtocol()
public String getHost()
public String getFile()
```

Il significato di questi metodi dovrebbe essere chiaro: restituiscono un singolo componente dell'oggetto URL.

```
public String toExternalForm()
```

Restituisce una stringa che rappresenta l'URL

```
public URLConnection openConnection() throws IOException
```

Restituisce un oggetto URLConnection (sarà trattato di seguito), che rappresenta una connessione con l'host dell'URL, secondo il protocollo adeguato. Tramite questo oggetto, si può accedere ai contenuti dell'URL.

```
public final InputStream openStream() throws IOException
```

Apre una connessione con l'URL, e restituisce un input stream. Tale stream può essere utilizzato per leggere i contenuti dell'URL.

```
public final Object getContent() throws IOException
```

Questo metodo restituisce un oggetto di classe Object che racchiude i contenuti dell'URL. Il tipo reale dell'oggetto restituito dipende dai contenuti dell'URL: se si tratta di un'immagine, sarà un oggetto di tipo Image, se si tratta di un file di testo, sarà una String. Questo metodo compie diverse azioni, invisibili all'utente, come stabilire la connessione con il server, inviare una richiesta, processare la risposta, ecc.

Un esempio

Ovviamente per ogni protocollo ci dovrà essere un appropriato gestore. Il JDK fornisce di default un gestore del protocollo HTTP, e quindi l'accesso alle informazioni web è alquanto semplice.

Nel caso dell'HTTP, ad esempio una chiamata al metodo openStream, il gestore del protocollo HTTP, invierà una richiesta al web server specificato con l'URL, analizzerà le risposte del server, e restituirà un input stream dal quale è possibile leggere i contenuti del particolare file richiesto. Richiedere un file a un server web è molto semplice, ed è illustrato nell'esempio seguente, che mostra anche l'utilizzo di altri metodi della classe.

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

public class HTTP_URL_Reader {
 public static void main(String args[]) throws IOException {
    if(args.length < 1)
       throw new IOException("Sintassi : HTTP_URL_Reader URL");

    URL url = new URL(args[0]);

    System.out.println("Componenti dell'URL");
    System.out.println("URL    : " + url.toExternalForm());
    System.out.println("Protocollo : " + url.getProtocol());
    System.out.println("Host       : " + url.getHost());
    System.out.println("Porta      : " + url.getPort());
    System.out.println("File     : " + url.getFile());

    System.out.println("Contenuto dell'URL :");

    // lettura dei dati dell'URL
    InputStream iStream = url.openStream();
    DataInputStream diStream = new DataInputStream(iStream);



String line ;
    while((line = diStream.readLine()) != null)
       System.out.println(line);

    diStream.close();
 }
}
```

È sufficiente creare un URL, passando al costruttore l'URL sotto forma di stringa, ottenere l'input stream chiamando l'apposito metodo, creare un DataInputStream basandosi su tale stream, e leggere una riga alla volta, stampandola sullo schermo. Il programma può essere eseguito così:

```
java HTTP_URL_Reader http://localhost/mydir/myfile.html
```

Se è installato un web server, si avranno stampate a schermo le varie componenti dell'URL specificato, nonché il contenuto del file richiesto.

La classe URLConnection

Questa classe rappresenta una connessione attiva, specifica di un dato protocollo, a un oggetto rappresentato da un URL. Tale classe è astratta, e quindi, per gestire uno specifico protocollo, si dovrebbe derivare da questa classe.

Descrizione classe

```
public class URLConnection extends Object
```

Costruttori

```
protected URLConnection(URL url)
```

Crea un oggetto di questa classe, dato un URL. Da notare che il costruttore è protetto, quindi può essere chiamato solo da una classe derivata. In effetti, come si è visto nella classe URL, un oggetto di questa classe viene ottenuto tramite la chiamata del metodo openConnection della classe URL.

Metodi

```
public URL getURL()
```

Restituisce semplicemente l'URL su cui è stato costruito l'oggetto URLConnection.

```
public abstract void connect() throws IOException
```

Permette di connettersi all'URL, specificato nel costruttore. La connessione quindi non avviene con la creazione dell'oggetto, ma avviene quando viene richiamato questo metodo, oppure un metodo che necessita che la connessione sia attiva (a quel punto la richiesta della connessione viene stabilita implicitamente).

```
public Object getContent() throws IOException
```

Restituisce il contenuto dell'URL. Viene restituito un oggetto di classe Object, poiché il tipo dell'oggetto dipende dal particolare URL.

```
public InputStream getInputStream() throws IOException
```

Restituisce un input stream con il quale si può leggere dall'URL.

```
public OutputStream getOutputStream() throws IOException
```

In questo caso si tratta di uno stream di output, con il quale è possibile inviare dati a un URL; ciò può risultare utile se si deve compiere un'operazione di post HTTP.

Questa classe contiene inoltre molti metodi che permettono di avere informazioni dettagliate sull'URL, quali il tipo di contenuto, la sua lunghezza, la data dell'ultima modifica, ecc. Per una rassegna completa si rimanda ovviamente alla documentazione on-line ufficiale.

Esistono poi alcuni metodi statici, da utilizzare per implementare gestori di protocollo personalizzati. Il trattamento di tale argomento va però oltre lo scopo di questo manuale.

I messaggi HTTP GET e POST

I web server permettono di ottenere informazioni come risultato di una query (interrogazione). Invece di richiedere un normale documento, si specifica nell'URL il nome di un programma (che segue l'interfaccia CGI), passandogli alcuni parametri che rappresentano la query vera e propria.

Molti sostengono che l'arrivo di Java abbia decretato la morte della programmazione CGI. In effetti tramite Java si ha più flessibilità, e i programmi vengono eseguiti dal lato client. Con la programmazione CGI invece il programma viene eseguito sul server, limitando così l'interazione con l'utente. Del resto il CGI è ancora molto usato, anche perché il web è pieno di programmi CGI già scritti e collaudati. Il presente paragrafo non vuole essere una spiegazione dettagliata della programmazione CGI (di cui non sarà data nessuna descrizione approfondita), ma vuol illustrare come dialogare con programmi CGI tramite Java.

Una query CGI è costituita quindi da un normale URL, con in coda alcuni parametri. La parte dell'URL che specifica i parametri inizia con un punto interrogativo (?). Ogni parametro è separato da una "e commerciale" (&), e i valori che si assegnano ai parametri sono specificati in questo modo: nome = valore (il valore è facoltativo). Un esempio di query è il seguente:

```
http://localhost/cgi-bin/mycgi.exe?nome=lorenzo&cognome=bettini&eta=29
```

In questo modo si richiama il programma CGI mycgi.exe e ad esso si passano i valori lorenzo, bettini, 29, da assegnare rispettivamente ai parametri nome, cognome, eta.

Con la classe URL, presente nel pacchetto java.net, eseguire una tale query è semplicissimo: basta passare tale stringa al costruttore della classe.

Una query spedisce quindi dei dati al web server, inserendoli direttamente nell'URL. In questo modo però si può andare incontro a problemi dovuti alla limitatezza della lunghezza delle query: non si possono spedire grandi quantità di dati in questo modo.

Per far questo si deve utilizzare un altro messaggio HTTP: il messaggio POST. Infatti mentre un messaggio GET spedisce solo un header (intestazione) nel proprio messaggio, un messaggio POST, oltre che di un header, è dotato anche di un contenuto (content). Vale a dire che un messaggio POST è molto simile, strutturalmente, ad una risposta del server web, quando si richiede un

documento (si veda a tal proposito l'esempio per ottenere una pagina web tramite le socket, nella sezione specifica). Infatti in un messaggio POST si deve includere il campo Content-length:.

Nel caso in cui si voglia inviare un messaggio POST si deve prima di tutto creare un oggetto URL, specificando un URL valido, creare un URLConnection con l'apposito metodo di URL, e abilitare la possibilità di utilizzare tale oggetto sia per l'output che per l'input. Inoltre è bene disabilitare la cache, in modo da essere sicuri che la risposta arrivi realmente dal server e non dalla cache.

```
URL destURL = new URL("http://localhost/cgi-bin/test-cgi");

URLConnection urlConn = destURL.openConnection();

urlConn.setDoOutput(true);
urlConn.setDoInput(true);
urlConn.setUseCaches(false);
```

Si deve poi riempire l'header del messaggio con alcune informazioni vitali, come il tipo del contenuto del messaggio e la lunghezza del messaggio, supponendo che il messaggio venga memorizzato in una stringa. Si ricordi che il contenuto deve essere sempre terminato da un \r\n.

```
String request = ...contenuto... + "\r\n";

urlConn.setRequestProperty("Content-type", "application/octet-stream");
urlConn.setRequestProperty("Content-length", "" + request.length());
```

A questo punto si può spedire il messaggio utilizzando lo stream dell'oggetto URLConnection (magari tramite un DataOutputStream). Dopo aver fatto questo ci si può mettere in attesa della risposta del server, sempre tramite lo stream (stavolta di input) di URLConnection.

```
DataOutputStream outStream
= new DataOutputStream(urlConn.getOutputStream());

outStream.writeBytes(request);
outStream.close();

DataInputStream inStream
= new DataInputStream(urlConn.getInputStream());

// lettura risposta dal server...
```

La classe Socket

Per creare una socket con un server in esecuzione su un certo host è sufficiente creare un oggetto di classe Socket, specificando nel costruttore l'indirizzo internet dell'host, e il numero di porta. Dopo che l'oggetto Socket è stato costruito è possibile ottenere (tramite appositi metodi)

due stream (uno di input e uno di output). Tramite questi stream è possibile comunicare con l'host, e ricevere messaggi da esso. Qualsiasi metodo che prenda in ingresso un InputStream (o un OutputStream) può comunicare con l'host in rete.

Quindi, una volta creata la socket, è possibile comunicare in rete tramite l'usuale utilizzo degli stream.

Descrizione classe

```
public class Socket extends Object
```

Costruttori

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
```

Viene creato un oggetto Socket connettendosi con l'host specificato (sotto forma di stringa o di InetAddress) alla porta specificata. Se sull'host e sulla porta specificata non c'è un server in ascolto, verrà generata un'IOException (verrà specificato il messaggio connection refused).

Metodi

```
public InetAddress getInetAddress()
```

Restituisce un oggetto InetAddress corrispondente all'indirizzo dell'host con il quale la socket è connessa.

```
public InetAddress getLocalAddress()
```

Restituisce un oggetto InetAddress corrispondente all'indirizzo locale al quale la socket è collegata.

```
public int getPort()
```

Restituisce il numero di porta dell'host remoto con il quale la socket è collegata.

```
public int getLocalPort()
```

Restituisce il numero di porta locale con la quale la socket è collegata. Quando si crea una socket, come si è già detto, ci si collega con un server su una certa macchina, che è in ascolto su una certa porta. Anche sulla macchina locale, sulla quale viene creata la socket, si userà per tale socket una determinata porta, assegnata dal sistema operativo, scegliendo il primo numero di porta non occupato. Si deve ricordare infatti che ogni connessione TCP consiste sempre di un indirizzo locale e di uno remoto, e di un numero di porta locale e un numero di porta remoto. Questo metodo può essere utile quando un programma, già collegato con un server remoto, crei esso stesso un server.

Per tale nuovo server può non essere specificato un numero di porta (a questo punto si prende il numero di porta assegnato dal sistema operativo). Con questo metodo si riesce a ottenere tale numero di porta, che potrà ad esempio essere comunicato ad altri programmi su altri host.

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Tramite questi metodi si ottengono gli stream, per mezzo dei quali è possibile comunicare attraverso la connessione TCP instaurata con la creazione della socket. Tale comunicazione sarà quindi basata sull'utilizzo degli stream, impiegati di continuo nella programmazione in Java. Come si può notare vengono restituite InputStream e OutputStream, che sono classi astratte. In realtà vengono restituiti dei SocketInputStream e SocketOutputStream, ma tali classi non sono pubbliche. Quando si comunica attraverso connessioni TCP, i dati vengono suddivisi in pacchetti (pacchetti IP appunto), quindi è consigliabile non utilizzare tali stream direttamente, ma sarebbe meglio costruire stream "bufferizzati" evitando così di avere pacchetti contenenti poche informazioni (infatti quando si inizia a scrivere i primi byte su tali stream, verranno spediti dei pacchetti con pochi byte, o forse anche un solo byte!).

```
public synchronized void close() throws IOException
```

Con questo metodo viene chiusa la socket (e quindi la connessione), e tutte le risorse che erano in uso verranno rilasciate. Dati contenuti nel buffer verranno comunque spediti, prima della chiusura del socket. La chiusura di uno dei due stream associati alla socket comporterà automaticamente la chiusura della socket stessa.

Può essere lanciata un'IOException, a significare che ci sono stati dei problemi sulla connessione. Ad esempio quando uno dei due programmi che utilizza la socket chiude la connessione, l'altro programma potrà ricevere una tale eccezione.

```
public synchronized void setSoTimeout(int timeout) throws SocketException
```

Dopo la chiamata di tale metodo, una lettura dall'InputStream della socket bloccherà il processo solo per una quantità di tempo pari a timeout (specificato in millisecondi). Se tale lasso di tempo scade, il processo riceverà un'InterruptedIOException. La socket rimane comunque valida e riutilizzabile. Se come timeout viene specificato 0, l'attesa sarà illimitata (infinita), che è anche la situazione di default.

```
public synchronized int getSoTimeout() throws SocketException
```

Con questo metodo si può ottenere il timeout settato con il precedente metodo. Se viene restituito 0, vuol dire che non è stato settato nessun timeout.

Quindi connettersi e comunicare con un server è molto semplice: basta creare una socket specificando host e porta (queste informazioni devono essere conosciute), ottenere e memorizzare gli stream della socket richiamando gli appositi metodi della socket, e utilizzarli per comunicare (sia per spedire informazioni, che per ricevere informazioni), magari dopo aver "bufferizzato" tali stream.

Utilizzo delle socket (client-server)

Si prenderà adesso in esame un semplice programma client: si tratta di un client HTTP che, dato un URL, richiede un file al server HTTP di quell'host. Si tratta di una variazione di HTTP_URL_Reader visto precedentemente durante la spiegazione della classe URL.

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

public class HTTPClient {
 public HTTPClient(String textURL) throws IOException {
    Socket socket = null;
    dissectURL(textURL);
    socket = connect();
    try {
       getPage();
    } finally {
       socket.close();
    }
 }

 protected String host, file;
 protected int port;

 protected void dissectURL(String textURL) throws MalformedURLException {
    URL url = new URL(textURL);
    host = url.getHost();
    port = url.getPort();
    if(port == -1)
       port = 80;
    file = url.getFile();
 }

 protected DataInputStream in;
 protected DataOutputStream out;

 protected Socket connect() throws IOException {
    System.err.println("Connessione a " + host + ":" + port + "...");
    Socket socket = new Socket(host, port);
    System.err.println("Connessione avvenuta.");

    BufferedOutputStream buffOut = new BufferedOutputStream(socket.getOutputStream());
    out = new DataOutputStream(buffOut);
    in = new DataInputStream(socket.getInputStream());

    return socket;
 }
```

```
  protected void getPage() throws IOException {
    System.err.println("Richiesta del file " + file + " inviata...");
    out.writeBytes("GET " + file + " HTTP/1.0\r\n\r\n");
    out.flush();

    System.err.println("Ricezione dati...");

    String input ;
    while((input = in.readLine()) != null)
      System.out.println(input);
  }

  public static void main(String args[]) throws IOException {
    if(args.length < 1)
      throw new IOException("Sintassi : HTTPClient URL");

    try {
      new HTTPClient(args[0]);
    } catch(IOException ex) {
      ex.printStackTrace();
    }

    System.out.println("exit");
  }
}
```

In effetti è stata ancora utilizzata questa classe per gestire l'URL passato sulla linea di comando, ma poi si effettua una connessione con il server creando esplicitamente una socket.

Nel metodo connect si effettua la connessione vera e propria aprendo una socket con l'host e sulla porta specificati:

```
Socket socket = new Socket(host, port);
```

Effettuata la connessione si possono ottenere gli stream associati con i metodi getOutputStream e getInputStream della classe Socket. Si crea poi un DataOutputStream e un DataInputStream su tali stream ottenuti (effettivamente, per ottimizzare la comunicazione in rete, prima viene creato uno stream "bufferizzato" sullo stream di output, ma questi dettagli, al momento, possono non essere approfonditi).

A questo punto si deve richiedere il file al server web e quindi si spedisce tale richiesta tramite lo stream di output:

```
out.writeBytes("GET " + file + " HTTP/1.0\r\n\r\n");
```

Ora non resta che mettersi in attesa, sullo stream di input, dell'invio dei dati dal server:

```
while((input = in.readLine()) != null)
```

Il contenuto del file viene stampato sullo schermo una riga alla volta.

Questo semplice programma illustra un esempio di client che invia al server una richiesta, e riceve dal server i dati richiesti. Questo è quanto avviene quando, tramite il proprio browser, si visita una pagina web: anche se in modo senz'altro più complesso il client apre una connessione con il server, comunica al server quello che desidera tramite un protocollo di comunicazione (nell'esempio, l'HTTP), e attende la risposta del server (comunicata sempre tramite lo stesso protocollo). Il comando GET infatti fa parte del protocollo HTTP.

Per testare il programma non è necessaria una connessione a Internet, basta avere un web server installato e attivo e lanciare il programma in questo modo:

```
java HTTPClient http://localhost/index.html
```

E sullo schermo verrà stampato il contenuto dell'intero file index.html (se il file viene trovato, ovviamente, altrimenti si otterrà il tipico errore di file non trovato a cui ormai la navigazione web ci ha abituati).

Si vedrà adesso un esempio di programma *server*. Un server rimane in attesa di connessioni su una certa porta e, ogni volta che un client si connette a tale porta, il server ottiene una socket, tramite la quale può comunicare con il client. Il meccanismo messo a disposizione da Java per queste operazioni è la classe ServerSocket, tramite la quale il server può appunto accettare connessioni dai client attraverso la rete.

I passi tipici di un server saranno quindi:

1. creare un oggetto di classe ServerSocket specificando un numero di porta locale;
2. attendere (tramite il metodo accept() di suddetta classe) connessioni dai client;
3. usare la socket ottenuta ad ogni connessione, per comunicare con il client.

Infatti il metodo accept() della classe ServerSocket crea un oggetto Socket per ogni connessione. Il server potrà poi comunicare come fa un client: estraendo gli stream di input ed output dalla socket.

Tali passi possono essere riassunti nel seguente estratto di listato:

```
ServerSocket server = new ServerSocket(port);
Socket client = server.accept();
server.close();

InputStream i = client.getInputStream();
OutputStream o = client.getOutputStream();
```

Il server dell'esempio precedente chiude il ServerSocket appena ha ricevuto una richiesta di connessione, quindi tale server funziona una sola volta! Si ricorda che tale chiusura non chiude la connessione con il client appena creata: semplicemente il server non accetta ulteriori connessioni. Un server che "si rispetti", invece deve essere in grado di accettare più connessioni e, inoltre, dovrebbe essere in grado di soddisfare più richieste contemporaneamente. Per risolvere questo problema si deve ricorrere al *multithreading*, per il quale Java offre diversi strumenti. Il programma sarà modificato nei due punti seguenti: il thread principale rimarrà in ascolto di richieste di connessioni; appena arriva una richiesta di connessione viene creato un thread che si occuperà di tale connessione e il thread principale tornerà ad aspettare nuove connessioni.

In effetti è questo quello che avviene nei server di cui si è già parlato. Se si osserva un programma scritto in C che utilizza le socket, si potrà vedere che appena viene ricevuta una richiesta di connessione, il programma si duplica (esegue una fork()), e il programma figlio lancia un programma che si occuperà di gestire la connessione appena ricevuta. Nel caso in esame basterà creare un nuovo thread e passargli la socket della nuova connessione.

Segue il programma modificato per trattare più connessioni contemporaneamente:

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

public class SimpleServer extends Thread {
 protected Socket client ;

 public SimpleServer(Socket socket) {
    System.out.println("Arrivato un nuovo client da " + socket.getInetAddress()) ;
    client = socket;
 }

 public void run() {
    try {
       InputStream i = client.getInputStream();
       OutputStream o = client.getOutputStream();
       PrintStream p = new PrintStream(o);
       p.println("BENVENUTI.");
       p.println("Questo è il SimpleServer :-)");
       p.println();
       p.println("digitare HELP per la lista di servizi disponibili");

       int x;
       ByteArrayOutputStream command =new ByteArrayOutputStream();
       String HelpCommand = new String("HELP");
       String QuitCommand = new String("QUIT");
       while((x = i.read()) > -1) {
          o.write(x);
          if(x == 13) { /* newline */
             p.println() ;
```

```
            if(HelpCommand.equalsIgnoreCase(command.toString())) {
               p.println("Il solo servizio disponibile è l'help,");
               p.println("e QUIT per uscire.");
               p.println("Altrimenti che SimpleServer sarebbe... ;-)");
            } else if(QuitCommand.equalsIgnoreCase(command.toString())) {
               p.println("Grazie per aver usato SimpleServer ;-)");
               p.println("Alla prossima. BYE");
               try {
                  Thread.sleep(1000);
               } finally {
                  break;

            }
            } else {
               p.println("Comando non disponibile !-( ");
               p.println("Digitare HELP per la lista dei servizi");
            }
            command.reset();
         } else if( x != 10 ) /* carriage return */
            command.write(x);
      }
   } catch(IOException e) {
         e.printStackTrace();
   } finally {
      System.out.println("Connessione chiusa con " + client.getInetAddress());
      try {
         client.close();
      } catch(IOException e) {
         e.printStackTrace();
      }
   }
}

public static void main(String args[]) throws IOException {
   int port = 0;
   Socket client;

   if(args.length == 1)
      port = Integer.parseInt(args[0]) ;

   System.out.println("Server in partenza sulla porta " + port);
   ServerSocket server = new ServerSocket(port);
   System.out.println("Server partito sulla porta " + server.getLocalPort() );

   while(true) {
      System.out.println("In attesa di connessioni...");
      client = server.accept();
```

```
            System.out.println("Richiesta di connessione da " + client.getInetAddress());
            (new SimpleServer(client)).start();
        }
    }
}
```

Questo programma accetta da linea di comando un parametro che specifica la porta su cui mettersi in ascolto di richieste di connessioni. Se non viene passato alcun argomento si userà la porta scelta dal sistema operativo. Dopo la creazione dell'oggetto ServerSocket ci si mette in ascolto di connessioni e, appena se ne riceve una, si fa partire un Thread per gestire tale connessione. In effetti tale classe deriva dalla classe Thread, e quindi, quando si crea un oggetto di questa classe, si crea in effetti un nuovo thread di esecuzione. In pratica si può riassumere:

- nel main si entra in un ciclo infinito (il ciclo finirà quando viene sollevata un'eccezione, oppure quando interrompiamo il programma), in cui viene eseguito accept();
- appena viene ricevuta una richiesta di connessione si crea un nuovo oggetto della classe (e quindi un nuovo thread), passando ad esso la socket relativa a tale connessione, e viene lanciato così un nuovo thread di esecuzione;
- si torna ad eseguire l'accept();
- il codice che si occupa della comunicazione con il client è nel metodo run che viene chiamato automaticamente quando un thread viene mandato in esecuzione (cioè quando si richiama il metodo start()).

Tramite l'oggetto Socket restituito dal metodo accept si ottengono i due stream per comunicare con il client. Si attende poi che il client invii dei comandi: ogni volta che viene letto un carattere, questo viene rispedito al client, in modo che quest'ultimo possa vedere quello che sta inviando. Appena viene digitato un newline (cioè invio o enter) si controlla se il servizio richiesto (memorizzato via via in un buffer) è disponibile, e si risponde in modo opportuno. Si noti come tutte le comunicazioni fra il server e il client siano racchiuse in un blocco try–finally: se nel frattempo avviene un'eccezione, si è comunque sicuri che la connessione verrà chiusa. L'eccezione in questione è tipicamente una IOException dovuta alla disconnessione da parte del client.

La classe deriva dalla classe Thread. È da notare come — poiché il metodo run della classe Thread, che viene ridefinito dalla nostra classe, non lancia nessuna eccezione — dobbiamo intercettare tutte le eccezioni all'interno del metodo: in questo caso l'eccezione in questione è IOException che può essere lanciata anche quando si cerca di chiudere la comunicazione. A proposito di client: in questo esempio, dov'è il client? Come nell'altro esempio avevamo usato un server già esistente (web server) per testare il nostro client, questa volta per testare il nostro server utilizzeremo un client classico: il *Telnet*.

Quindi se si è lanciato il server con la seguente riga di comando

```
java SimpleServer 9999
```

basterà utilizzare da un'altro terminale (ad esempio un'altra shell del DOS in Windows, o un altro xterm sotto Linux) il seguente comando:

```
telnet localhost 9999
```

Adesso è possibile inviare richieste al server semplicemente inserendo una stringa e premendo INVIO (provate ad esempio con "HELP").

User Datagram Protocol (UDP)

Finora si è sempre parlato del TCP (*Transfer Control Protocol*), un protocollo sviluppato sopra l'IP (*Internet Protocol*). Un altro protocollo basato sempre sull'IP, è l'UDP (*User Datagram Protocol*). In questo protocollo vengono spediti pacchetti di informazioni. Si tratta di un protocollo non basato sulla connessione (*connectionless*) e che non garantisce né l'arrivo né l'ordine dei pacchetti. Comunque, se i pacchetti arrivano, è garantito che siano integri e non corrotti. In un protocollo basato sulla connessione, come il TCP, si deve prima di tutto stabilire la connessione, dopo di che tale connessione può essere utilizzata sia per spedire che per ricevere. Quando la comunicazione è terminata, la connessione dovrà essere chiusa. Nell'UDP, invece, ogni messaggio sarà spedito come un pacchetto indipendente, che seguirà un percorso indipendente. Oltre a non garantire l'arrivo di tali pacchetti il protocollo non garantisce nemmeno che i pacchetti arrivino nell'ordine in cui sono stati spediti, e che non ci siano duplicati. Entrambi i protocolli utilizzano pacchetti, ma l'UDP, da questo punto di vista, è molto più vicino all'IP.

Ma perché utilizzare un protocollo così poco "affidabile"? Si tenga presente, che rispetto al TCP, l'UDP ha molto poco overhead (dovendo fare molti meno controlli), quindi può essere utilizzato quando la latenza è di fondamentale importanza. La perdita di pacchetti UDP è dovuta sostanzialmente alla congestione della rete. Utilizzando Internet questo è molto comune, ma se si utilizza una rete locale, questo non dovrebbe succedere.

Si può, comunque, aggiungere manualmente un po' di controllo sulla spedizione dei pacchetti. Si può supporre che, se non si riceve una risposta entro un certo tempo, il pacchetto sia andato perso, e quindi può essere rispedito. Va notato che, se il server ha ricevuto il pacchetto ma è la sua risposta che ha trovato traffico nella rete, il server riceverà nuovamente un pacchetto identico al precedente!

Si potrebbe allora pensare di implementare ulteriori controlli, ma questo porterebbe sempre più vicini al TCP. Nel caso in cui si voglia avere sicurezza sulla qualità di pacchetti, conviene passare direttamente al protocollo TCP appositamente pensato per questo scopo.

La classe DatagramPacket

Si devono creare oggetti di questa classe sia per spedire, sia per ricevere pacchetti. Un pacchetto sarà costituito dal messaggio vero e proprio e dall'indirizzo di destinazione. Per ricevere un pacchetto UDP si dovrà costruire un oggetto di questa classe e accettare un pacchetto UDP dalla rete. Non si possono filtrare tali pacchetti: si ricevono tutti i pacchetti UDP con il proprio indirizzo.

Descrizione classe

```
public final class DatagramPacket extends Object
```

Costruttori

```
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)
```

Si costruisce un datagram packet specificando il contenuto del messaggio (i primi ilength bytes dell'array ibuf) e l'indirizzo IP del destinatario (sempre nella forma "host + numero porta").

> Importante: poiché si tratta di protocolli differenti, un server UDP ed uno TCP possono essere in ascolto sulla stessa porta.

```
public DatagramPacket(byte ibuf[], int ilength)
```

In questo modo si costruisce un oggetto DatagramPacket da utilizzare per ricevere pacchetti UDP dalla rete. Il pacchetto ricevuto sarà memorizzato nell'array ibuf che dovrà essere in grado di contenere il pacchetto. ilenght specifica la dimensione massima di bytes che potranno essere ricevuti.

Metodi

Vi sono alcuni metodi che permettono di leggere le informazioni e il contenuto di un pacchetto UDP.

```
public InetAddress getAddress()
```

Se il pacchetto è stato ricevuto, tale metodo restituirà l'indirizzo dello host mittente; se l'oggetto DatagramSocket è invece stato creato per essere trasmesso, conterrà l'indirizzo IP del destinatario.

```
public int getPort()
```

Restituisce il numero del mittente o destinatario (vedi sopra), che può essere utilizzato per rispondere.

```
public byte[] getData()
```

Estrae dal pacchetto il contenuto del messaggio. L'array avrà la stesa grandezza specificata nel costruttore, e non l'effettiva dimensione del messaggio. Per ottenere tale dimensione si deve utilizzare il seguente metodo:

```
public int getLength()
```

La classe DatagramSocket

Questa classe permette di spedire e ricevere pacchetti UDP (sempre utilizzando le socket). Quando si spedisce un pacchetto UDP, come nel TCP, ci deve essere un DatagramSocket in ascolto sulla porta specificata.

Trattandosi di un protocollo connectionless, lo stesso oggetto DatagramSocket può essere utilizzato per spedire pacchetti a host differenti e ricevere pacchetti da host diversi.

Descrizione classe

```
public class DatagramSocket extends Object
```

Costruttori

Si può specificare il numero di porta, oppure lasciare che sia il sistema operativo ad assegnarla. Tipicamente se si deve spedire un pacchetto (client) si utilizzerà la porta assegnata dal sistema operativo, e se si deve ricevere (server) si specificherà un numero di porta preciso. Ovviamente tale numero dovrà essere noto anche ai client. Ci sono quindi due costruttori:

```
public DatagramSocket() throws SocketException

public DatagramSocket(int port) throws SocketException
```

Si può inoltre specificare anche l'indirizzo al quale tale socket sarà legata:

```
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

Metodi

```
public void send(DatagramPacket p) throws IOException
```

Spedisce un pacchetto all'indirizzo di destinazione.

```
public synchronized void receive(DatagramPacket p) throws IOException
```

Riceve un singolo pacchetto UDP memorizzandolo in p. A questo punto si potranno ottenere tutte le informazioni su tale pacchetto, con i metodi della classe DatagramPacket.

```
public InetAddress getLocalAddress()
public int getLocalPort()
public synchronized void setSoTimeout(int timeout) throws SocketException
public synchronized int getSoTimeout() throws SocketException
public void close()
```

Questi metodi hanno lo stesso significato degli omonimi metodi della classe Socket; si rimanda quindi alla trattazione di tale classe.

Un esempio

Ecco un semplice esempio di utilizzo del protocollo UDP, tramite le due classi appena illustrate. Si tratta di due classi UDPSender e UDPReceiver, il cui nome dovrebbe essere esplicativo circa il loro funzionamento.

Ecco UDPSender:

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

public class UDPSender {
 static protected DatagramPacket buildPacket(String host, int port, String message) throws IOException {
    ByteArrayOutputStream boStream = new ByteArrayOutputStream();
    DataOutputStream doStream = new DataOutputStream(boStream);
    doStream.writeUTF(message);
    byte[] data = boStream.toByteArray();
    return new DatagramPacket(data, data.length,
                                    netAddress.getByName(host), port);
 }

 public static void main(String args[]) throws IOException {
    if(args.length < 3)
       throw new IOException("Uso: UDPSender <host> <port> <messaggio> {messaggi}");

    DatagramSocket dsocket = new DatagramSocket();
    DatagramPacket dpacket ;

    for(int i = 2; i < args.length; i++) {
       dpacket = buildPacket(args[0], Integer.parseInt(args[1]), args[i]);
       dsocket.send(dpacket);
       System.out.println("Messaggio spedito");
    }
 }
}
```

e UDPReceiver:

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

public class UDPReceiver {
```

```
 static protected void showPacket(DatagramPacket p) throws IOException {
    System.out.println("Mittente : " + p.getAddress());
    System.out.println("porta  : " + p.getPort()) ;
    System.out.println("Lunghezza messaggio : " + p.getLength());
    ByteArrayInputStream biStream
    = new ByteArrayInputStream(p.getData(), 0, p.getLength());
    DataInputStream diStream = new DataInputStream(biStream);
    String content = diStream.readUTF();
    System.out.println("Messaggio : " + content);
 }

 public static void main(String args[]) throws IOException {
    if(args.length != 1)
       throw new IOException("Uso: UDPReceiver <port>");

    byte buffer[] = new byte[65536];
    DatagramSocket dsocket
= new DatagramSocket(Integer.parseInt(args[0]));
    DatagramPacket dpacket;

    while(true) {
       System.out.println("In attesa di messaggi...");
       dpacket = new DatagramPacket(buffer, buffer.length);
       dsocket.receive(dpacket);
       System.out.println("Ricevuto messaggio");
       showPacket(dpacket);
    }
 }
}
```

Si dovrà lanciare prima l'UDPReceiver specificando semplicemente il numero di porta su cui rimarrà in ascolto:

```
java UDPReceiver 9999
```

E su un altro terminale si potrà lanciare il sender specificando l'indirizzo e la porta, e poi una serie di stringhe, che verranno inviate al receiver:

```
java UDPSender localhost 9999 ciao a tutti
```

A questo punto il receiver mostrerà le informazioni riguardanti ogni messaggio ricevuto.

Su altri terminali si possono lanciare altri sender, sempre diretti allo stesso receiver, e si potrà notare che il receiver potrà ricevere messaggi da più sender, tramite lo stesso DatagramSocket. Non trattandosi di un protocollo con connessione, il socket rimarrà attivo anche quando i sender termineranno, cosa che non accade quando si crea una connessione diretta tramite una socket nel TCP.

Nuove estensioni e classi di utility presenti nella piattaforma Java 2

A partire dalla versione 2 del linguaggio sono state aggiunte al package java.net alcune classi di utilità che offrono un maggiore livello di astrazione o mettono a disposizione alcune *feature* ormai comuni nell'ambito del networking. Queste classi sono dedicate principalmente allo sviluppo di applicazioni che si appoggiano sul protocollo HTTP. Si vedranno qui di seguito, sinteticamente, alcune di queste classi con la descrizione dei principali metodi.

La classe HttpURLConnection

Estensione della classe URLConnection, questa classe mette a disposizione alcuni metodi specifici per il protocollo HTTP, metodi che permettono di tralasciare alcuni dettagli implementativi del protocollo stesso.

Il costruttore della classe ha la seguente firma

```
protected HttpURLConnection(URL myUrl)
```

Anche in questo caso, analogamente alla URLConnection, il costruttore è protected; per ottenere un oggetto di questa classe sarà sufficiente ricorrere allo stesso sistema con cui si ottiene un URLConnection, preoccupandosi di eseguire il cast opportuno.

Ad esempio si può scrivere

```
URL url = new URL(name);
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
```

Metodi

```
public InputStream getErrorStream()
```

In caso di fallimento della connessione, permette di utilizzare lo stream restituito per ottenere le informazioni eventualmente inviate dal server sulle cause del fallimento.

```
public boolean getFollowRedirect()
```

Restituisce true se questa connessione è abilitata a seguire le redirezioni indicate dal server, false altrimenti (vedi oltre: setFollowRedirect()).

```
public Permission getPermission()
```

Restituisce un oggetto di tipo Permission, contenente i permessi necessari a eseguire questa connessione.

```
public String getRequestMethod()
```

Restituisce il metodo richiesto per questa connessione (POST, GET, ecc.) (vedi oltre: setRequestMethod()).

```
public int getResponseCode()
```

Restituisce il codice di stato della richiesta, inviato dal server.

```
public String getResponseMessage()
```

Restituisce il messaggio di risposta del server, collegato al codice.

```
public static void setFollowRedirect(boolean set)
```

Permette di configurare il comportamento di questa connessione a fronte di una richiesta di redirezione da parte del server.

```
public void setRequestMethod(String method) throws ProtocolException
```

Utilizzato per settare il metodo voluto per questa connessione. Il parametro è tipicamente una stringa indicante una delle operazioni previste dal protocollo HTTP, ad esempio GET, POST, ecc.; il metodo di default è GET.

La classe JarURLConnection

Questa classe astrae la connessione (HTTP) verso file archivio .jar remoti: il suo utilizzo si dimostra utile ad esempio nelle Applet, per accedere a file di immagini già presenti nella cache del browser.

Il meccanismo per ottenere un oggetto di tipo JarURLConnection è analogo a quello per HttpURLConnection; da notare in questo caso che, nella creazione della URL, è necessario specificare nel protocollo che si richiede una connessione ad un file .jar.

```
URL url = new URL("jar:http://www.my.address/jarfile.jar");
```

Per ottenere tutto il file.

```
URL url = new URL("jar:http://www.my.address/jarfile.jar/adirectory/afilename");
```

Per ottenere un file contenuto all'interno dell'archivio.

```
URL url = new URL("jar:http://www.my.address/jarfile.jar/adirectory");
```

Per ottenere una directory contenuta nell'archivio.

Anche in questo caso per il costruttore

```
protected JarURLConnection(URL url)
```

vale quanto illustrato per la classe HttpURLConnection.

Metodi

```
public String getEntryName()
```

Restituisce la entry name se la connessione punta a un file contenuto in un archivio, null altrimenti.

```
public JarEntry getJarEntry()
```

Con questo metodo è possibile ottenere la JarEntry riferita all'oggetto della connessione; attraverso la JarEntry è possibile ottenere informazioni sull'oggetto della connessione, quali la dimensione in bytes, il metodo di compressione, ecc.

```
public JarFile getJarFile()
```

Restituisce il file .jar a cui fa riferimento questa connessione. Da notare che il file in questione non è modificabile.

```
public Manifest getManifest()
```

Restituisce, se esiste, il file Manifest contenuto nell'archivio.

Capitolo 18

JavaBeans

Andrea Gini

La programmazione a componenti

Uno degli obiettivi più ambiziosi dell'ingegneria del software è organizzare lo sviluppo di sistemi in maniera simile a quanto è stato fatto in altre branche dell'ingegneria, dove la presenza di un mercato di parti standard altamente riutilizzabili permette di aumentare la produttività riducendo nel contempo i costi. Nella meccanica, ad esempio, esiste da tempo un importante mercato di componenti riutilizzabili, come viti, dadi, bulloni e ruote dentate; ciascuno di questi componenti trova facilmente posto in centinaia di prodotti diversi.

L'industria del software, sempre più orientata alla filosofia dei componenti, sta dando vita a due nuove figure di programmatore: il progettista di componenti e l'assemblatore di applicazioni.

Il primo ha il compito di scoprire e progettare oggetti software di uso comune, che possano essere utilizzati con successo in contesti differenti. Produttori in concorrenza tra di loro possono realizzare componenti compatibili, ma con caratteristiche prestazionali differenti. L'acquirente può orientarsi su un mercato che offre una pluralità di scelte e decidere in base al budget o a particolari esigenze di prestazione.

L'assemblatore di applicazioni, d'altra parte, è un professionista specializzato in un particolare dominio applicativo, capace di creare programmi complessi acquistando sul mercato componenti standard e combinandoli con strumenti grafici o linguaggi di scripting.

Questo capitolo offre un'analisi approfondita delle problematiche che si incontrano nella creazione di componenti in Java; attraverso gli esempi verrà comunque offerta una panoramica su come sia possibile assemblare applicazioni complesse a partire da componenti concepiti per il riuso.

La specifica JavaBeans

JavaBeans è una specifica, ossia un insieme di regole seguendo le quali è possibile realizzare in Java componenti software riutilizzabili, che abbiano la capacità di interagire con altri componenti, realizzati da altri produttori, attraverso un protocollo di comunicazione comune.

Ogni Bean è caratterizzato dai servizi che è in grado di offrire e può essere utilizzato in un ambiente di sviluppo differente rispetto a quello in cui è stato realizzato. Quest'ultimo punto è cruciale nella filosofia dei componenti: sebbene i Java Beans siano a tutti gli effetti classi Java, e possano essere manipolati completamente per via programmatica, essi vengono spesso utilizzati in ambienti di sviluppo diversi, come tool grafici o linguaggi di scripting.

I tool grafici, tipo JBuilder, permettono di manipolare i componenti in maniera visuale. Un assemblatore di componenti può selezionare i Beans da una palette, inserirli in un apposito contenitore, impostarne le proprietà, collegare gli eventi di un Bean ai metodi di un altro, generando in tal modo applicazioni, Applet, Servlet e persino nuovi componenti senza scrivere una sola riga di codice.

I linguaggi di scripting, di contro, offrono una maggiore flessibilità rispetto ai tool grafici, senza presentare le complicazioni di un linguaggio generico. La programmazione di pagine web dinamiche, uno dei domini applicativi di maggior attualità, deve il suo rapido sviluppo a un'intelligente politica di stratificazione, che vede le funzionalità di più basso livello, come la gestione dei database, la Businnes Logic o l'interfacciamento con le risorse di sistema, incapsulate all'interno di JavaBeans, mentre tutto l'aspetto della presentazione viene sviluppato con un semplice linguaggio di scripting, tipo Java Server Pages o PHP.

Il modello a componenti JavaBeans

Un modello a componenti è caratterizzato da almeno sette fattori: proprietà, metodi, introspezione, personalizzazione, persistenza, eventi e modalità di deployment. Nei prossimi paragrafi si analizzerà il ruolo di ciascuno di questi aspetti all'interno della specifica Java Beans; quindi si procederà a descriverne l'implementazione in Java.

Proprietà

Le proprietà sono attributi privati, accessibili solamente attraverso appositi metodi `get` e `set`. Tali metodi costituiscono l'unica via di accesso pubblica alle proprietà, cosa che permette al progettista di componenti di stabilire per ogni parametro precise regole di accesso. Se si utilizzano i Bean all'interno di un programma di sviluppo visuale, le proprietà di un componente vengono visualizzate in un apposito pannello, che permette di modificarne il valore con un opportuno strumento grafico.

Metodi

I metodi di un Bean sono metodi pubblici Java, con l'unica differenza che essi risultano accessibili anche attraverso linguaggi di scripting e Builder Tools. I metodi sono la prima e più importante via d'accesso ai servizi di un Bean.

Introspezione

I Builder Tools scoprono i servizi di un Bean (proprietà, metodi ed eventi) attraverso un processo noto come introspezione, che consiste principalmente nell'interrogare il componente per conoscerne i metodi, e dedurre da questi le caratteristiche. Il progettista di componenti può attivare l'introspezione in due maniere: seguendo precise convenzioni nella formulazione delle firme dei metodi, o creando una speciale classe `BeanInfo`, che fornisce un elenco esplicito dei servizi di un particolare Bean.

La prima via è senza dubbio la più semplice: se si definiscono i metodi di accesso a un determinato servizio seguendo le regole di naming descritte dalla specifica JavaBeans, i tool grafici saranno in grado, grazie alla reflection, di individuare i servizi di un Bean semplicemente osservandone l'interfaccia di programmazione. Il ricorso ai BeanInfo, d'altro canto, torna utile in tutti quei casi in cui sia necessario mascherare alcuni metodi, in modo da esporre solamente un sottoinsieme dei servizi effettivi del Bean.

Personalizzazione

Durante il lavoro di composizione di Java Beans all'interno di un tool grafico, un apposito Property Sheet, generato al volo dal programma di composizione, mostra lo stato delle proprietà e permette di modificarle con un opportuno strumento grafico, tipo un campo di testo per valori `String` o una palette per proprietà `Color`. Simili strumenti grafici vengono detti editor di proprietà.

I tool grafici dispongono di editor di proprietà in grado di supportare i tipi Java più comuni, come i tipi numerici, le stringhe e i colori; nel caso si desideri rendere editabile una proprietà di un tipo diverso, è necessario realizzare un'opportuna classe di supporto, conforme all'interfaccia `PropertyEditor`. Quando invece si desideri fornire un controllo totale sulla configurazione di un Bean, è possibile definire un Bean Customizer, una speciale applicazione grafica specializzata nella configurazione di un particolare tipo di componenti.

Persistenza

La persistenza permette ad un Bean di salvare il proprio stato e di ripristinarlo in un secondo tempo. JavaBeans supporta la persistenza grazie all'Object Serialization, che permette di risolvere questo problema in modo molto rapido.

Eventi

Nella programmazione a oggetti tradizionale non esiste nessuna convenzione su come modellare lo scambio di messaggi tra oggetti. Ogni programmatore adotta un proprio sistema, creando una fitta rete di dipendenze che rende molto difficile il riutilizzo di oggetti in contesti differenti da quello di partenza. Gli oggetti Java progettati secondo la specifica Java Beans adottano un meccanismo di comunicazione basato sugli eventi, simile a quello utilizzato nei componenti grafici Swing e AWT. L'esistenza di un unico protocollo di comunicazione standard garantisce l'intercomunicabilità tra componenti, indipendentemente da chi li abbia prodotti.

Deployment

I JavaBeans possono essere consegnati, in gruppo o singolarmente, attraverso file JAR, speciali archivi compressi in grado di trasportare tutto quello di cui un Bean ha bisogno, come classi, immagini o altri file di supporto. Grazie ai file .jar è possibile consegnare i Beans con una modalità del tipo "chiavi in mano": l'acquirente deve solamente caricare un file JAR nel proprio ambiente di sviluppo e i Beans in esso contenuti verranno subito messi a disposizione. L'impacchettamento di classi Java all'interno di file JAR segue poche semplici regole, che verranno descritte negli esempi del capitolo.

Guida all'implementazione dei JavaBeans

Realizzare un componente Java Bean è un compito alla portata di qualunque programmatore Java che disponga di buone conoscenze di sviluppo Object Oriented. Nei paragrafi seguenti verranno descritte dettagliatamente le convenzioni di naming dettate dalla specifica, e verranno fornite le istruzioni su come scrivere le poche righe di codice necessarie a implementare i meccanismi che caratterizzano i servizi Bean. Infine verranno presentati degli esempi, che permetteranno di impratichirsi con il processo di implementazione delle specifiche.

Le proprietà

Le proprietà sono attributi che descrivono l'aspetto e il comportamento di un Bean, e che possono essere modificate durante tutto il ciclo di vita del componente. Di base, le proprietà sono attributi privati, ai quali si accede attraverso una coppia di metodi della forma:

```
public <PropertyType> get<PropertyName>()
public void set<PropertyName>(<PropertyType> property)
```

La convenzione di aggiungere il prefisso get e set ai metodi che forniscono l'accesso a una proprietà, permette ad esempio ai tool grafici di rilevare le proprietà Bean, determinarne le regole di accesso (Read Only o Read/Write), dedurne il tipo, visualizzare le proprietà su un apposito Property Sheet e individuare l'editor di proprietà più adatto al caso.

Se ad esempio un tool grafico scopre, grazie all'introspezione, la coppia di metodi

```
public Color getForegroundColor() { ... }
public void setForegroundColor(Color c) { ... }
```

da questi conclude che esiste una proprietà chiamata foregroundColor (notare la prima lettera minuscola), accessibile sia in lettura che in scrittura, di tipo Color. A questo punto, il tool può cercare un editor di proprietà per parametri di tipo Color, e mostrare la proprietà su un property sheet in modo che possa essere vista e manipolata dal programmatore.

Proprietà indicizzate (Indexed Property)

Le proprietà indicizzate permettono di gestire collezioni di valori accessibili attraverso indice, in maniera simile a come si fa con un vettore. Lo schema di composizione dei metodi di accesso di una proprietà indicizzata è il seguente:

```
public <PropertyType>[] get<PropertyName>();
public void set<PropertyName>(<PropertyType>[] value);
```

per i metodi che permettono di manipolare l'intera collection, mentre per accedere ai singoli elementi, si deve predisporre una coppia di metodi del tipo:

```
public <PropertyType> get<PropertyName>(int index);
public void set<PropertyName>(int index, <PropertyType> value);
```

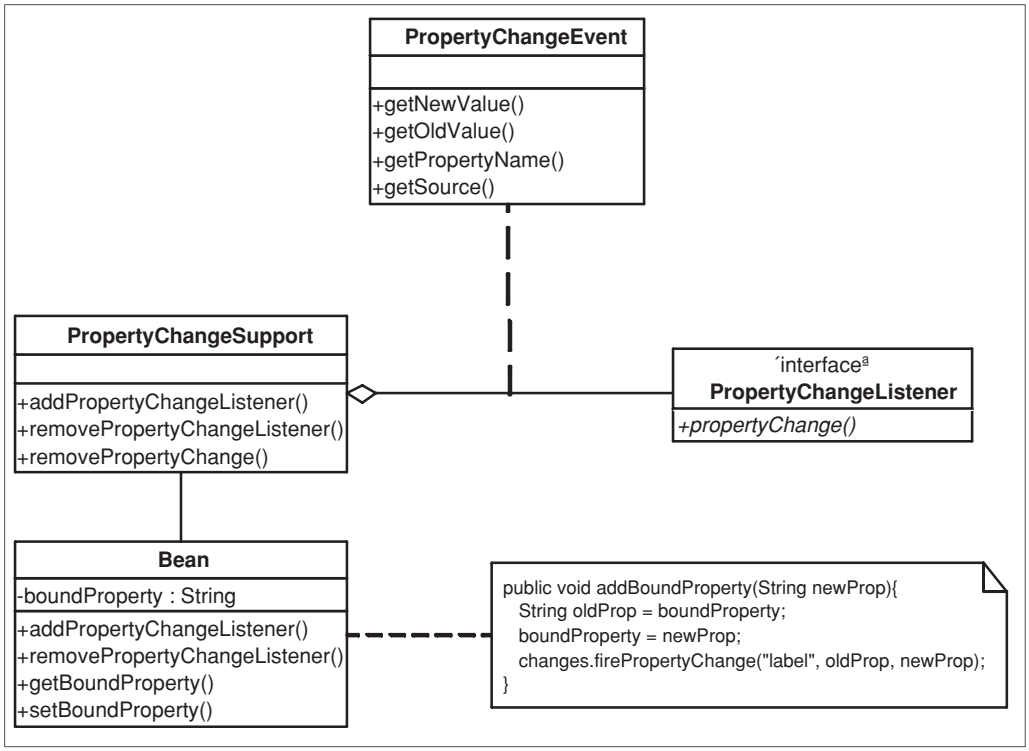
Proprietà bound

Le proprietà semplici, così come sono state descritte nei paragrafi precedenti, seguono una convenzione radicata da tempo nella normale programmazione a oggetti. Le proprietà bound, al contrario, sono caratteristiche dell'universo dei componenti, dove si verifica molto spesso la necessità di collegare il valore delle proprietà di un componente a quelli di un'altro, in modo tale che si mantengano aggiornati. I metodi set delle proprietà bound, inviano una notifica a tutti gli ascoltatori registrati ogni qualvolta viene alterato il valore della proprietà. Il meccanismo di ascolto–notifica, simile a quello degli eventi Swing e AWT, segue il pattern Observer.

Le proprietà bound, a differenza degli eventi Swing, utilizzano un unico tipo di evento, ChangeEvent, cosa che semplifica il processo di implementazione. La classe PropertyChangeSupport, presente all'interno del package java.bean, fornisce i metodi che gestiscono la lista degli ascoltatori e quelli che producono l'invio degli eventi.

Un oggetto che voglia mettersi in ascolto di una proprietà, deve implementare l'interfaccia PropertyChangeListener e deve registrarsi presso la sorgente di eventi. L'oggetto PropertyChangeEvent incapsula le informazioni riguardo alla proprietà modificata, alla sorgente e al valore della proprietà.

Figura 18.1 – *Il meccanismo di notifica di eventi bound segue il pattern Observer*



Come implementare il supporto alle proprietà bound

Per aggiungere a un Bean il supporto alle proprietà bound, bisogna anzitutto importare il package java.beans.*, in modo da garantire l'accesso alle classi PropertyChange-Support e PropertyChangeEvent. Quindi bisogna creare un oggetto PropertyChangeSupport, che ha il compito di mantenere la lista degli ascoltatori e di fornire i metodi che gestiscono l'invio degli eventi.

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

A questo punto bisogna realizzare, nella propria classe, i metodi che permettono di gestire la lista degli ascoltatori. Tali metodi sono dei semplici metodi Wrapper che fanno riferimento a metodi con la stessa firma, presenti nel PropertyChangeSupport:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
 changes.addPropertyChangeListener(l);
}
```

```
public void removePropertyChangeListener(PropertyChangeListener l) {
 changes.removePropertyChangeListener(l);
}
```

La presenza dei metodi addPropertyChangeListener() e removeProperty-ChangeListener() permette ai tool grafici di riconoscere un oggetto in grado di inviare proprietà bound e di mettere a disposizione un'opportuna voce nel menù di gestione degli eventi.

L'ultimo passaggio consiste nel modificare i metodi set relativi alle proprietà che si vuole rendere bound, per fare in modo che venga generato un PropertyChangeEvent ogni volta che la proprietà viene reimpostata

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 changes.firePropertyChange("color", oldColor , newColor);
}
```

Nel caso di proprietà read only, prive di metodo set, l'invio dell'evento dovrà avvenire all'interno del metodo che attua la modifica della proprietà. Un aspetto interessante del meccanismo di invio di PropertyChangeEvent, è che essi trasportano sia il nuovo valore che quello vecchio. Questa scelta dispensa chi implementa un ascoltatore dal compito di mantenere una copia del valore, qualora questo fosse necessario, dal momento che l'evento viene propagato *dopo* la modifica della relativa proprietà. Il metodo fireChangeEvent() della classe PropertyChangeListener fornisce il servizio di Event Dispatching:

```
firePropertyChange(String propertyName, Object oldValue, Object newValue)
```

In pratica esso impacchetta i parametri in un oggetto PropertyChangeEvent, e chiama il metodo propertyChange(PropertyChangeEvent p) su tutti gli ascoltatori registrati. I parametri vengono trattati come Object, e nel caso si debbano inviare proprietà espresse in termini di tipi primitivi, occorre incapsularle nell'opportuno Wrapper (Integer per valori int, Double per valori double e così via). Per facilitare questo compito, la classe propertyChangeSupport prevede delle varianti di firePropertyChange per valori int e boolean.

Come implementare il supporto alle proprietà bound su sottoclassi di JComponent

La classe JComponent, superclasse di tutti i componenti Swing, dispone del supporto nativo alla gestione di proprietà bound. Di base essa fornisce i metodi addProperty-ChangeListener e removePropertyChangeListener, oltre a una collezione di metodi firePropertyChange adatta ad ogni tipo primitivo. In questo caso l'implementazione di una proprietà bound richiederà solo una modifica al metodo set preposto, similmente a come descritto nell'ultimo passaggio del precedente paragrafo, con la differenza che non è necessario ricorrere a un oggetto propertyChangeSupport per inviare la proprietà:

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 firePropertyChange("color", oldColor , newColor);
}
```

Ascoltatori di proprietà

Se si desidera mettersi in ascolto di una proprietà, occorre definire un opportuno oggetto PropertyChangeListener e registrarlo presso il Bean. Un Property-ChangeListener deve definire il metodo propertyChange(PropertyChangeEvent e), che viene chiamato quando avviene la modifica di una proprietà bound.

Un PropertyChangeListener viene notificato quando avviene la modifica di *una qualunque* proprietà bound: per questa ragione esso deve, come prima cosa, verificare, che la proprietà appena modificata sia quella alla quale si è interessati. Una simile verifica richiede una chiamata al metodo getPropertyName di PropertyChangeEvent, che restituisce il nome della proprietà. Per convenzione, i nomi di proprietà vengono estratti dai nomi dichiarati nei metodi get e set, con la prima lettera minuscola. Il seguente frammento di codice presenta un tipico PropertyChange-Listener, che ascolta la proprietà foregroundColor:

```
public class Listener implements PropertyChangeListener() {
 public void propertyChange(PropertyChangeEvent e) {
    if(e.getPropertyName().equals("foregroundColor"))
       System.out.println(e.getNewValue());
 }
}
```

Un esempio di Bean con proprietà bound

Un Java Bean rappresenta un mattone di un programma. Ogni componente è un'unità di utilizzo abbastanza grossa da incorporare una funzionalità evoluta, ma piccola rispetto ad un programma fatto e finito. Il concetto del riuso può essere presente a diversi livelli del progetto: il seguente Bean fornisce un esempio di elevata versatilità

Il Bean PhotoAlbum è un pannello grafico al cui interno vengono caricate delle immagini. Il metodo showNext() permette di passare da un'immagine all'altra, in modo ciclico. Il numero ed il tipo di immagini viene determinato al momento dell'avvio: durante la fase di costruzione viene letto il file comment.txt, presente nella directory images, che contiene una riga di commento per ogni immagine presente nella cartella. Le immagini devono essere nominate in modo progressivo (img0.jpg, img1.jpg, img2.jpg…) e devono essere presenti in numero uguale alle righe del file comment.txt. Questa scelta progettuale consente di introdurre il riuso a un livello abbastanza alto: qualunque utente, anche con scarse conoscenze del linguaggio, può personalizzare il componente, inserendo le sue foto preferite, senza la necessità di alterare il codice sorgente.

Il Bean PhotoAlbum ha tre proprietà:

- imageNumber, che restituisce il numero di immagini contenute nell'album. Essendo una quantità immutabile, tale proprietà è stata implementata come proprietà semplice.

- imageIndex: restituisce l'indice dell'immagine attualmente visualizzata. Al cambio di immagine viene inviato un PropertyChangeEvent.

- imageComment: restituisce una stringa di commento all'immagine. Anche in questo caso, al cambio di immagine viene generato un PropertyChangeEvent.

Il Bean viene definito come sottoclasse di JPanel: per questo motivo non vengono dichiarati i metodi addPropertyChangeListener e removeProperty-ChangeListener, già presenti nella superclasse. L'invio delle proprietà verrà messo in atto grazie al metodo firePropertyChange di JComponent.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class PhotoAlbum extends JPanel {

 private Vector comments = new Vector();
 private int imageIndex;

 public PhotoAlbum() {
    super();
    setLayout(new BorderLayout());
    setupComments();
    imageIndex = 0;
    showNext();
 }
 private void setupComments() {
    try {
       URL indexUrl = getClass().getResource("images/" + "comments.txt");
       InputStream in = indexUrl.openStream();
       BufferedReader lineReader = new BufferedReader(new InputStreamReader(in));
       String line;
       while((line = lineReader.readLine())!=null)
          comments.add(line);
    }
    catch(Exception e) {
       e.printStackTrace();
```

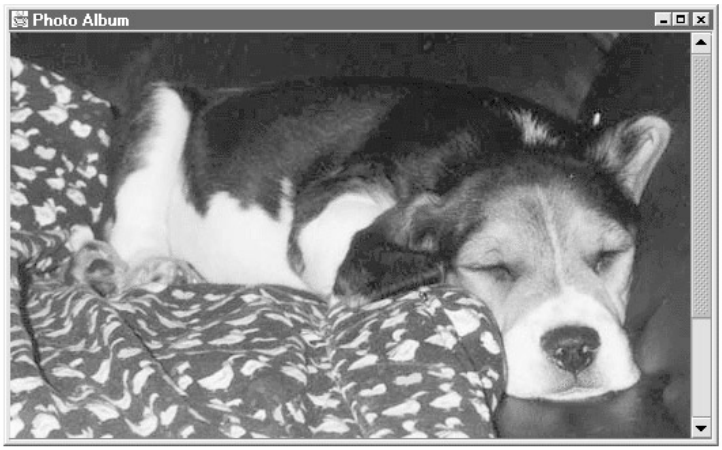
```
        }
    }
    public int getImageNumber() {
        return comments.size();
    }
    public int getImageIndex() {
        return imageIndex;
    }
    public String getImageComment() {
        return (String)comments.elementAt(imageIndex);
    }
    public void showNext() {
        int oldImageIndex = imageIndex;
        imageIndex = ((imageIndex +1 ) % comments.size());
        String imageName = «img» + Integer.toString(imageIndex) + «.jpg»;
        showImage(getClass().getResource(„images/“ + imageName));
        String oldImageComment = (String)comments.elementAt(oldImageIndex);
        String currentImageComment = (String)comments.elementAt(imageIndex);
        firePropertyChange(“imageComment”, oldImageComment, currentImageComment);
        firePropertyChange(“imageIndex”, oldImageIndex, imageIndex);
    }
    private void showImage(URL imageUrl) {
        ImageIcon img = new ImageIcon(imageUrl);
        JLabel picture = new JLabel(img);
        JScrollPane pictureScrollPane = new JScrollPane(picture);
        removeAll();
        add(BorderLayout.CENTER,pictureScrollPane);
        validate();
    }
}
```

È possibile testare il Bean come fosse una normale classe Java, utilizzando queste semplici righe di codice:

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import com.mokabyte.mokabook.javaBeans.photoAlbum.*;
import java.beans.*;
import javax.swing.*;

public class PhotoAlbumTest {
public static void main(String argv[]) {
    JFrame f = new JFrame(“Photo Album”);
    PhotoAlbum p = new PhotoAlbum();
    f.getContentPane().add(p);
    p.addPropertyChangeListener(new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent e) {
```

```
            System.out.println(e.getPropertyName() + ": " + e.getNewValue());
        }
    });
f.setSize(500,400);
f.setVisible(true);

while(true)
    for(int i=0;i<7;i++) {
        p.showNext();
        try {Thread.sleep(1000);}catch(Exception e) {}
    }
}
}
```

Figura 18.2 – *Un programma di prova per il Bean PhotoAlbum*



Creazione di un file JAR

Prima di procedere alla consegna del Bean entro un file JAR, bisogna anzitutto compilare le classi PhotoAlbum.java e PhotoAlbumTest.java, che devono trovarsi nella cartella com\mokabyte\mokabook\javaBeans\

```
javac com\mokabyte\mokabook\javaBeans\photoAlbum\*.java
```

A questo punto bisogna creare, ricorrendo a un semplice editor di testo tipo Notepad, un file photoAlbumManifest.tmp con il seguente contenuto

```
Main-Class: com.mokabyte.mokabook.javaBeans.photoAlbum.PhotoAlbumTest
```

```
Name: com/mokabyte/mokabook/javaBeans/photoAlbum/PhotoAlbum.class
Java-Bean: True
```

Le prime due righe, opzionali, segnalano la presenza di una classe dotata di metodo main.

Le ultime due righe del file manifest specificano che la classe PhotoAlbum.class è un Java Bean. Se l'archivio contiene più di un Bean, è necessario elencarli tutti.

Per generare l'archivio photoAlbum.jar, bisogna digitare la riga di comando:

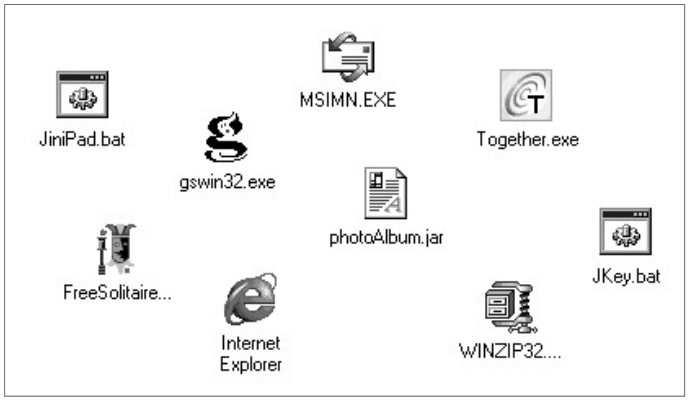
```
jar cfm photoAlbum.jar photoAlbumManifest.tmp
 com\mokabyte\mokabook\javaBeans\photoAlbum\*.class
 com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

Il file così generato contiene tutte le classi e le immagini necessarie a dar vita al Bean PhotoAlbum. Tale file potrà essere utilizzato facilmente all'interno di tool grafici o di pagine web, racchiuso dentro una Applet.

Il file .jar potrà essere avviato digitando

```
java PhotoAlbum.jar
```

Figura 18.3 – *Un file JAR opportunamente confezionato può essere aperto con un opportuno tool come Jar o WinZip*



Le istruzioni fornite sono valide per la piattaforma Windows. Su piattaforma Unix, le eventuali occorrenze del simbolo "\", che funge da path separator su piattaforme Windows, andranno sostituite col simbolo "/". Le convenzioni adottate all'interno del file manifest valgono invece su entrambe le piattaforme.

Integrazione con altri Bean

Nonostante il Bean PhotoAlbum fornisca un servizio abbastanza evoluto, non è ancora classificabile come applicazione. Esso, opportunamente integrato con altri Beans, può comunque dar vita a numerosi programmi; di seguito, ecco qualche esempio: collegato a un CalendarBean, PhotoAlbum può dar vita a un simpatico calendario elettronico; collegando un bottone Bean al metodo showNext() è possibile creare un album interattivo, impacchettarlo su un'Applet e pubblicarlo su Internet; impacchettando il Bean PhotoAlbum con foto natalizie, e collegandolo con un Bean Carillon, si può ottenere un biglietto di auguri elettronico.

Figura 18.4 – *Combinando, all'interno del Bean Box, il Bean PhotoAlbum con un pulsante Bean, si ottiene una piccola applicazione*

A questi esempi se ne possono facilmente aggiungere altri; altri ancora diventano possibili aggiungendo al Bean nuovi metodi, come previousImage() e setImageAt(int i); un compito ormai alla portata del lettore che fornisce un ottimo pretesto per esercitarsi.

Eventi Bean

La notifica del cambiamento di valore delle proprietà bound è un meccanismo di comunicazione tra Beans. Se si vuole che un Bean sia in grado di propagare eventi di tipo più generico, o comunque eventi che non è comodo rappresentare come un cambiamento di stato, è possibile utilizzare un meccanismo di eventi generico, del tutto simile a quello pre-

sente nei componenti grafici Swing e AWT. I prossimi paragrafi servono a illustrare le tre fasi dell'implementazione: creazione dell'evento, definizione dell'ascoltatore e infine creazione della sorgente di eventi.

Creazione di un evento

Per implementare un meccanismo di comunicazione basato su eventi, occorre anzitutto definire un'opportuna sottoclasse di EventObject, che racchiuda tutte le informazioni relative all'evento da propagare.

```
public class <EventType> extends EventObject {
 private <ParamType> param
 public <EventType>(Object source,<ParamType> param) {
    super(source);
    this.param = param;
 }
 public <ParamType> getParameter() {
    return param;
 }
}
```

La principale variazione sul tema si ha sul numero e sul tipo di parametri: tanto più complesso è l'evento da descrivere, maggiori saranno i parametri in gioco. L'unico parametro che è obbligatorio fornire è un reference all'oggetto che ha generato l'evento: tale reference, richiamabile con il metodo getSource() della classe EventObject, permetterà all'ascoltatore di interrogare la sorgente degli eventi qualora ce ne fosse bisogno.

Destinatari di eventi

Il secondo passaggio è quello di definire l'interfaccia di programmazione degli ascoltatori di eventi. Tale interfaccia deve essere definita come sottoclasse di EventListener, per essere riconoscibile come ascoltatore dall'Introspector. Lo schema di sviluppo degli ascoltatori segue lo schema

```
import java.awt.event.*;

public Interface <EventListener> extends EventListener {
 public void <EventType>Performed(<EventType> e);
}
```

Le convenzioni di naming dei metodi dell'interfaccia non seguono uno schema standard: la convenzione descritta nell'esempio, <EventType>performed, può essere seguita o meno. L'importante è che il nome dei metodi dell'interfaccia Listener suggeriscano il tipo di azione sottostante, e che accettino come parametro un evento del tipo giusto.

Sorgenti di eventi

Se si desidera aggiungere a un Bean la capacità di generare eventi, occorre implementare una coppia di metodi

```
add<EventListenerType>(<EventListenerType> l)
remove<EventListenerType>(<EventListenerType> l)
```

La gestione della lista degli ascoltatori e l'invio degli eventi segue una formula standard, descritta nelle righe seguenti:

```
private Vector listeners = new Vector();

public void add<EventListenerType>(<EventListenerType> l) {
 listeners.add(l);
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listeners.remove(l);
}
protected void fire<Eventype>(<EvenType> e) {
 Enumeration listenersEnumeration = listeners.elements();
 while(listenersEnumeration.hasMoreElements()) {
 <EventListenerType> listener = (<EventListenerType>)listenersEnumeration.nextElement();
    listener.<EventType>Performed(e);
 }
}
```

Sorgenti unicast

In alcuni casi occorre definire sorgenti di eventi capaci di servire un unico ascoltatore. Per implementare tali classi, che fungono da sorgenti unicast, si può seguire il seguente modello

```
private <EventListenerType> listener;

public void add<EventListenerType>(
<EventListenerType> l) throws TooManyListenersException {
 if(listener == null)
    listener = l;
 else
    throw new TooManyListenerException();
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listener = null;
}
protected void fire<Eventype>(<EvenType> e) {
```

```
if(listener! = null)
    listener.<EventType>Performed(e);
}
```

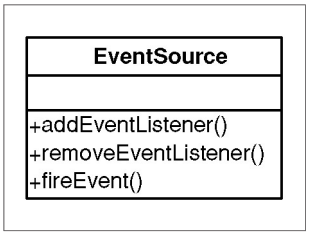
Ascoltatori di eventi: Event Adapter

Se si vuole che un evento generato da un Bean scateni un'azione su un altro Bean, è necessario creare un oggetto che realizzi un collegamento tra i due. Tale classe, detta Adapter, viene registrata come ascoltatore presso la sorgente dell'evento, e formula una chiamata al metodo destinazione ogni volta che riceve una notifica dal Bean sorgente.

Gli strumenti grafici tipo JBuilder generano questo tipo di classi in maniera automatica: tutto quello che l'utente deve fare è collegare, con pochi click di mouse, l'evento di un Bean sorgente a un metodo di un Bean target. Qui di seguito viene riportato il codice di un Adapter, generato automaticamente dal Bean Box, che collega la pressione di un pulsante al metodo startJuggling(ActionEvent e) del Bean Juggler.

```
// Automatically generated event hookup file.

public class ___Hookup_172935aa26 implements     java.awt.event.ActionListener, java.io.Serializable {

 public void setTarget(sunw.demo.juggler.Juggler t) {
    target = t;
 }

 public void actionPerformed(java.awt.event.ActionEvent arg0) {
    target.startJuggling(arg0);
 }

 private sunw.demo.juggler.Juggler target;
}
```

Figura 18.5 – *Un Adapter funge da ponte di collegamento tra gli eventi di un Bean e i metodi di un altro*



Un esempio di Bean con eventi

Il prossimo esempio è un Bean Timer, che ha il compito di generare battiti di orologio a intervalli regolari. Questo componente è un tipico esempio di Bean non grafico.

La prima classe che si definisce è quella che implementa il tipo di evento

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerEvent extends EventObject implements Serializable {

 public TimerEvent(Object source) {
    super(source);
 }
}
```

Come si può vedere, l'implementazione di un nuovo tipo di evento è questione di poche righe di codice. L'unico particolare degno di nota è che il costruttore del nuovo tipo di evento deve invocare il costruttore della superclasse, passando un reference alla sorgente dell'evento.

L'interfaccia che rappresenta l'ascoltatore deve estendere l'interfaccia EventListener; a parte questo, al suo interno si può definire un numero arbitrario di metodi, la cui unica costante è quella di avere come parametro un reference all'evento da propagare.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public interface TimerListener extends java.util.EventListener {
 public void clockTicked(TimerEvent e);
}
```

Per finire, ecco il Bean vero e proprio. Come si può notare, esso implementa l'interfaccia Serializable che rende possibile la serializzazione.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerBean implements Serializable {
```

```
private int time = 1000;
private transient TimerThread timerThread;
private Vector timerListeners = new Vector();

public void addTimerListener(TimerListener t) {
 timerListeners.add(t);
}
public void removeTimerListener(TimerListener t) {
 timerListeners.remove(t);
}
protected void fireTimerEvent(TimerEvent e) {
 Enumeration listeners = timerListeners.elements();
 while(listeners.hasMoreElements())
    ((TimerListener)listeners.nextElement()).clockTicked(e);
}
public synchronized void setMillis(int millis) {
 time = millis;
}
public synchronized int getMillis() {
 return time;
}
public synchronized void startTimer() {
 if(timerThread!=null)
    forceTick();
 timerThread = new TimerThread();
 timerThread.start();
}
public synchronized void stopTimer() {
 if(timerThread == null)
    return;

 timerThread.killTimer();
 timerThread = null;
}
public synchronized void forceTick() {
 if(timerThread!=null) {
    stopTimer();
    startTimer();
 }
 else
    fireTimerEvent(new TimerEvent(this));
}

 class TimerThread extends Thread {
    private boolean running = true;

    public synchronized void killTimer() {
```

```
        running = false;
    }
    private synchronized boolean isRunning() {
        return running;
    }
    public void run() {
        while(true)
            try {
                if(isRunning()) {
                    fireTimerEvent(new TimerEvent(TimerBean.this));
                    Thread.sleep(getMillis());
                }
                else
                    break;
        catch(InterruptedException e) {}
        }
    }
 }
}
```

I primi tre metodi servono a gestire la lista degli ascoltatori. Il terzo e il quarto gestiscono la proprietà millis, ossia il tempo, in millisecondi, tra un tick e l'altro. I due metodi successivi, startTimer, stopTimer, servono ad avviare e fermare il timer, mentre forceTick lancia un tick e riavvia il timer, se questo è attivo. Il timer vero e proprio viene implementato grazie a una classe interna TimerThread, sottoclasse di Thread. Si noti il metodo killTimer, che permette di terminare in modo pulito la vita del thread: questa soluzione è da preferire al metodo stop (deprecato a partire dal JDK 1.1), che in certi casi può provocare la terminazione del thread in uno stato inconsistente.

Per compilare le classi del Bean, bisogna usare la seguente riga di comando

```
javac com\mokabyte\mokabook\javaBeans\timer\*.java
```

Per impacchettare il Bean in un file .jar, è necessario per prima cosa creare con un editor di testo il file timerManifest.tmp, con le seguenti righe

```
Name: com/mokabyte/mokabook/javaBeans/timer/TimerBean.class
Java-Bean: True
```

Per creare l'archivio si deve quindi digitare il seguente comando

```
jar cfm timer.jar timerManifest.tmp com\mokabyte\mokabook\javaBeans\timer\*.class
```

Per testare la classe TimerBean, si può usare il seguente programma, che crea un oggetto TimerBean e registra un TimerListener il quale stampa a video una scritta ad ogni tick del timer.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public class TimerTest {
 public static void main(String argv[]) {


TimerBean t = new TimerBean();
    t.addTimerListener(new TimerListener() {
      public void clockTicked(TimerEvent e) {
        System.out.println("Tick");
      }
    });
    t.startTimer();
 }
}
```

Introspezione: l'interfaccia BeanInfo

Le convenzioni di naming descritte nei paragrafi precedenti permettono ai tool grafici abilitati ai Beans di scoprire i servizi di un componente grazie alla reflection. Questo processo automatico è certamente comodo, ma ha il difetto di non offrire nessun tipo di controllo sul numero e sul tipo di servizi da mostrare. In alcune occasioni può essere necessario mascherare un certo numero di servizi, specie quelli ereditati da una superclasse.

I Beans creati a partire dalla classe JComponent, ad esempio, ereditano automaticamente più di dieci attributi (dimensioni, colore, allineamento…) e ben dodici tipi diversi di evento (ComponentEvent, MouseEvent, HierarchyEvent…). Un simile eccesso provoca di solito disorientamento nell'utente; in questi casi è preferibile fornire un elenco esplicito dei servizi da associare al nostro Bean, in modo da "ripulire" gli eccessi.

Per raggiungere questo obiettivo, bisogna associare al Bean una classe di supporto, che implementi l'interfaccia BeanInfo. Una classe BeanInfo permette di fare un certo numero di cose: esporre solamente i servizi che si desidera rendere visibili, aggirare le convenzioni di naming imposte dalle specifiche Java Beans, associare al Bean un'icona e attribuire ai servizi nomi più descrittivi di quelli rilevabili con il processo di analisi delle firme dei metodi.

Creare una classe BeanInfo

Per creare una classe BeanInfo bisogna anzitutto definire una classe con lo stesso nome del Bean, a cui si deve aggiungere il suffisso BeanInfo. Per semplificare il lavoro si può estendere SimpleBeanInfo, una classe che fornisce un'implementazione nulla di tutti i metodi dell'interfaccia. In questo modo ci si limiterà a sovrascrivere solamente i metodi che interessano, lasciando tutti gli altri con l'impostazione di default.

Per ridefinire il numero ed il tipo dei servizi Bean, occorre agire in modo appropriato a restituire le proprietà, i metodi o gli eventi che si desidera esporre. Opzionalmente, si può associare

un'icona al Bean, definendo il metodo public java.awt.Image getIcon(int iconKind). Per finire, si può specificare la classe del Bean e il suo Customizer, qualora ne esista uno, con il metodo public BeanDescriptor getBeanDescriptor().

La classe BeanInfo così prodotta deve essere messa nello stesso package che contiene il Bean. In assenza di una classe BeanInfo, i servizi di un Bean vengono trovati con la reflection.

Feature Descriptors

Una classe di tipo BeanInfo restituisce, tramite i seguenti metodi, vettori di *descriptors* che contengono informazioni relative ad ogni proprietà, metodo o evento che il progettista di un Bean desidera esporre.

```
PropertyDescriptor[] getPropertyDescriptors();
MethodDescriptor[] getMethodDescriptors();
EventSetDescriptor[] getEventSetDescriptors();
```

Ogni Descriptor fornisce una precisa rappresentazione di una classe di servizi Bean. Il package java.bean implementa le seguenti classi:

- FeatureDescriptor: è la classe base per tutte le altre classi Descriptor, e definisce gli aspetti comuni a tutta la famiglia.
- BeanDescriptor: descrive il tipo e il nome della classe Bean associati, oltre a fornire il Customizer, se ne esiste uno.
- PropertyDescriptor: descrive le proprietà del Bean.
- IndexedPropertyDescriptor: è una sottoclasse di PropertyDescriptor, e descrive le proprietà indicizzate.
- EventSetDescriptor: descrive gli eventi che il Bean è in grado di inviare.
- MethodDescriptor: descrive i metodi del Bean.
- ParameterDescriptor: descrive i parametri dei metodi.

Esempio

In questo esempio si analizzerà un BeanInfo per il Bean PhotoAlbum, che permette di nascondere una grossa quantità di servizi Bean che per default vengono ereditati dalla superclasse JPanel.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.beans.*;
```

```
import com.mokabyte.mokabook.javaBeans.photoAlbum.*;

public class PhotoAlbumBeanInfo extends SimpleBeanInfo {

private static final Class beanClass = PhotoAlbum.class;

 public PropertyDescriptor[] getPropertyDescriptors() {
    try {
       PropertyDescriptor imageNumber
       = new PropertyDescriptor("imageNumber", beanClass, "getImageNumber", null);
       PropertyDescriptor imageIndex = new PropertyDescriptor("imageIndex", beanClass, "getImageIndex", null);
       PropertyDescriptor imageComment
       = new PropertyDescriptor("imageComment", beanClass, "getImageComment", null);

       imageIndex.setBound(true);
       imageComment.setBound(true);

       PropertyDescriptor properties[]
       = {imageNumber, imageIndex, imageComment};
       return properties;
    } catch (IntrospectionException e) {
        throw new Error(e.toString());
    }
 }
 public EventSetDescriptor[] getEventSetDescriptors() {
    try {
       EventSetDescriptor changed
       = new EventSetDescriptor(beanClass, "propertyChange", PropertyChangeListener.class, "propertyChange");
       changed.setDisplayName("Property Change");
       EventSetDescriptor events[] = {changed};
       return events;
    } catch (IntrospectionException e) {
       throw new Error(e.toString());
    }
 }
 public MethodDescriptor[] getMethodDescriptors() {
    try {
       MethodDescriptor showNext
       = new MethodDescriptor(beanClass.getMethod("showNext", null));

          MethodDescriptor methods[] = {showNext};
          return methods;
    } catch (Exception e) {
           throw new Error(e.toString());
    }
 }
 public java.awt.Image getIcon(int iconKind){
```

```
    if(iconKind == SimpleBeanInfo.ICON_COLOR_16x16)
      return loadImage("photoAlbumIcon16.gif");
    else
      return loadImage("photoAlbumIcon32.gif");
  }
}
```

La classe viene definita come sottoclasse di SimpleBeanInfo, in modo da rendere il processo di sviluppo più rapido.

Il primo metodo, getPropertyDescriptors, restituisce un array con un tre PropertyDescriptor, uno per ciascuna delle proprietà che si vogliono rendere visibili. Il costruttore di PropertyDescriptor richiede quattro argomenti: il nome della proprietà, la classe del Bean, il nome del metodo getter e quello del metodo setter: quest'ultimo è posto a null, a significare che le proprietà sono di tipo Read Only. Si noti, in questo metodo e nei successivi, che la creazione dei Descriptors deve essere definita all'interno di un blocco try–catch, dal momento che può generare IntrospectionException.

Il secondo metodo, getEventSetDescriptors(), restituisce un vettore con un unico EventSetDescriptor. Quest'ultimo viene inizializzato con quattro parametri: la classe del Bean, il nome della proprietà, la classe dell'ascoltatore e la firma del metodo che riceve l'evento. Si noti la chiamata al metodo setDisplayName(), che permette di impostare un nome più leggibile di quello che viene normalmente ottenuto dalle firme dei metodi.

Il terzo metodo, getMethodDescriptors, restituisce un vettore contenente un unico MethodDescriptor, che descrive il metodo showNext(). Il costruttore di MethodDescriptor richiede come unico parametro un oggetto di classe Method, che in questo esempio viene richiesto alla classe PhotoAlbum ricorrendo alla reflection.

Infine il metodo getIcon() restituisce un'icona, che normalmente viene associata al Bean all'interno di strumenti visuali.

Per impacchettare il Bean PhotoAlbum con le icone e il BeanInfo, si può seguire la procedura già descritta, modificando la riga di comando dell'utility jar in modo da includere le icone nell'archivio.

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp com\mokabyte\mokabook\
javaBeans\photoAlbum\*.class com\mokabyte\mokabook\javaBeans\
photoAlbum\*.gif com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

Personalizzazione dei Bean

L'aspetto e il comportamento di un Bean possono essere personalizzati in fase di composizione all'interno di un tool grafico abilitato ai Beans. Esistono due strumenti per personalizzare un Bean: gli Editor di proprietà e i Customizer. Gli Editor di proprietà sono componenti grafici specializzati nell'editing di un particolare *tipo* di proprietà: interi, stringhe, files… Ogni Editor di proprietà viene associato a un particolare tipo Java, e il tool grafico compone automaticamente un Property Sheet analizzando le proprietà di un Bean, e ricorrendo agli Editor più adatti alla circostanza. In fig. 18.6 si può vedere un esempio di Property Sheet, realizzato dal Bean Box: ogni riga presenta, accanto al nome della proprietà, il relativo Editor.

Figura 18.6 – *Un Property Sheet generato in modo automatico a partire dalle proprietà di un pulsante Bean*

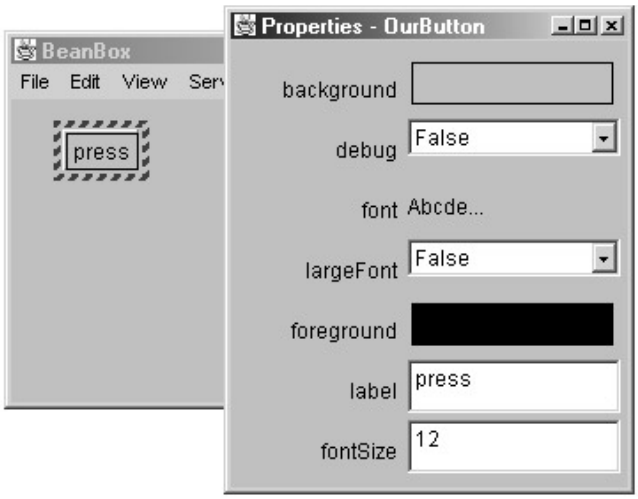
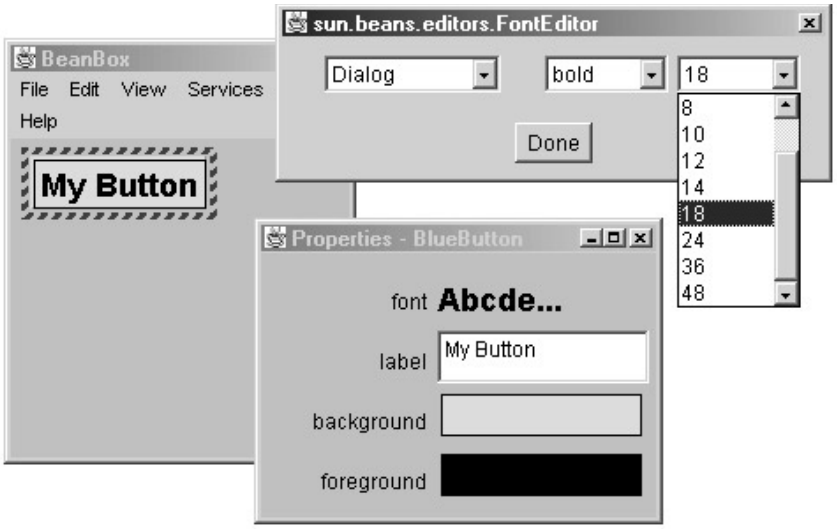


Figura 18.7 – *Il Property Sheet relativo a un pulsante Bean. Si noti il pannello ausiliario FontEditor*



Un Customizer, d'altra parte, è un pannello di controllo specializzato per un particolare Bean: in questo caso è il programmatore a decidere cosa mostrare nel pannello e in quale maniera. Per questa ragione un Customizer viene associato, grazie al BeanInfo, a un particolare Bean e non può, in linea di massima, essere usato su Bean differenti.

Come creare un Editor di proprietà

Un Editor di proprietà deve implementare l'interfaccia PropertyEditor, o in alternativa, estendere la classe PropertyEditorSupport che fornisce un'implementazione standard dei metodi dell'interfaccia. L'interfaccia PropertyEditor dispone di metodi che premettono di specificare come una proprietà debba essere rappresentata in un property sheet. Alcuni Editor consistono in uno strumento direttamente editabile, altri presentano uno strumento a scelta multipla, come un ComboBox; altri ancora, per permettere la modifica, aprono un pannello separato, come nella proprietà font dell'esempio, che viene modificata grazie al pannello ausiliario FontEditor.

Per fornire il supporto a queste modalità di editing, bisogna implementare alcuni metodi di PropertyEditor, in modo che ritornino valori non nulli.

I valori numerici o String possono implementare il metodo setAsText(String s), che estrae il valore della proprietà dalla stringa che costituisce il parametro. Questo sistema permette di inserire una proprietà con un normale campo di testo.

Gli Editor standard per le proprietà Color e Font usano un pannello separato, e ricorrono al Property Sheet solamente per mostrare l'impostazione corrente. Facendo click sul valore, viene aperto l'Editor vero e proprio. Per mostrare il valore corrente della proprietà, è necessario sovrascrivere il metodo isPaintable() in modo che restituisca true, e sovrascrivere paintValue in modo che dipinga la proprietà attuale in un rettangolo all'interno del Property Sheet.

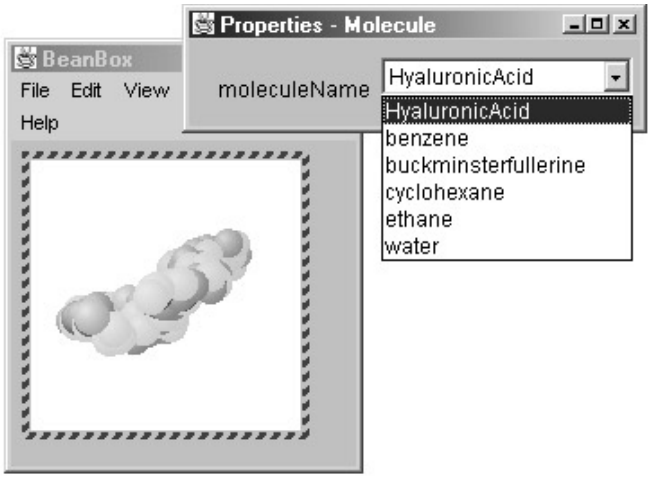
Per supportare l'Editor di Proprietà personalizzato occorre sovrascrivere altri due metodi della classe PropertyEditorSupport: supportsCustomEditor, che in questo caso deve restituire true, e getCustomEditor, in modo che restituisca un'istanza dell'Editor.

Registrare gli Editor

I Property Editor vengono associati alle proprietà attraverso un'associazione esplicita, all'interno del metodo getPropertyDescriptors() del BeanInfo, con una chiamata al metodo setPropertyEditorClass(Class propertyEditorClass) del PropertyDescriptor corrispondente, come avviene nel Bean Molecule

```
PropertyDescriptor pd = new PropertyDescriptor("moleculeName", Molecule.class);
pd.setPropertyEditorClass(MoleculeNameEditor.class);
```

Figura 18.8 – *Il Bean Molecule associa alla proprietà moleculeName di un Editor di proprietà personalizzato*



In alternativa si può registrare l'Editor con il seguente metodo statico

```
PropertyEditorManager.registerEditor(Class targetType, Class editorType)
```

che richiede come parametri la classe che specifica il tipo e quella che specifica l'Editor.

Customizers

Con un Bean Customizer è possibile fornire un controllo completo sul modo in cui configurare ed editare un Bean. Un Customizer è in pratica una piccola applicazione specializzata nell'editing di un particolare Bean, ogni volta che la configurazione di un Bean richiede modalità troppo sofisticate per il normale processo di creazione automatica del Property Sheet.

Le uniche regole a cui ci si deve attenere per realizzare un Customizer sono:

- deve estendere la classe `Component`, o una delle sue sottoclassi;
- deve implementare l'interfaccia `java.bean.Customizer`;
- deve implementare un costruttore privo di parametri.

Per associare il Customizer al proprio Bean, bisogna sovrascrivere il metodo `getBeanDescriptor` nella classe `BeanInfo`, in modo che restituisca un opportuno `BeanDescriptor`, il quale a sua volta dovrà restituire la classe del Customizer alla chiamata del metodo `getCustomizerClass`.

Serializzazione

Per rendere serializzabile una classe Bean è di norma sufficiente implementare l'interfaccia Serializable, sfruttando così l'Object Serialization di Java. L'interfaccia Serializable non contiene metodi: essa viene usata dal compilatore per marcare le classi che possono essere serializzate. Esistono solo poche regole per implementare classi Serializable: anzitutto è necessario dichiarare un costruttore privo di argomenti, che verrà chiamato quando l'oggetto verrà ricostruito a partire da un file .ser; in secondo luogo una classe serializzabile deve definire al suo interno solamente attributi serializzabili.

Se si desidera fare in modo che un particolare attributo non venga salvato al momento della serializzazione, si può ricorrere al modificatore transient. La serializzazione standard, inoltre, non salva lo stato delle variabili static.

Per tutti i casi in cui la serializzazione standard non risultasse applicabile, occorre procedere all'implementazione dell'interfaccia Externalizable, fornendo, attraverso i metodi readExternal(ObjectInput in) e writeExternal(ObjectOutput out), delle istruzioni esplicite su come salvare lo stato di un oggetto su uno stream e come ripristinarlo in un secondo tempo.

Appendice A

Installazione dell'SDK

GIOVANNI PULITI

Scelta del giusto SDK

Per poter lavorare con applicazioni Java o crearne di nuove, il programmatore deve poter disporre di un ambiente di sviluppo e di esecuzione compatibile con lo standard 100% Pure Java. Sun da sempre rilascia un kit di sviluppo che contiene tutti gli strumenti necessari per la compilazione ed esecuzione di applicazioni Java. Tale kit è comunemente noto come Java Development Kit (JDK): nel corso del tempo sono state rilasciate le versioni 1.0, 1.1, 1.2, 1.3 e 1.4. Attualmente l'ultima versione rilasciata è la 1.4, mentre si annuncia un prossimo JDK 1.5. Il JDK comprende una Java Virtual Machine (JVM), invocabile con il comando `java`, un compilatore (comando `javac`), un debugger (`jdbg`), un interprete per le applet (`appletviewer`) e altro ancora.

A partire dalla versione 1.2, Sun ha introdotto una nomenclatura differente per le varie versioni del kit di sviluppo. In quel momento nasceva infatti Java 2, a indicare la raggiunta maturità del linguaggio e della piattaforma. Pur mantenendo la completa compatibilità con il passato, Java 2 ha introdotto importanti miglioramenti, quali una maggiore stabilità e sicurezza, migliori performance e l'ottimizzazione dell'uso della memoria.

Con Java 2 nasce il concetto di SDK: non più un Java Development Kit ma un Software Development Kit. Il linguaggio Java può essere finalmente considerato un potente strumento general purpose.

La notazione di JDK non è stata eliminata: il JDK è formalmente una release dell'SDK Java 2.

Con Java 2, per organizzare e raccogliere al meglio le diverse tecnologie che costituiscono ormai la piattaforma, Sun ha suddiviso l'SDK in tre grandi categorie:

- Java 2 Standard Edition (J2SE): questa versione contiene la JVM standard più tutte le librerie necessarie per lo sviluppo della maggior parte delle applicazioni Java.

- Java 2 Enterprise Edition (J2EE): contiene in genere le API enterprise come EJB, JDBC 2.0, Servlet ecc. La JVM normalmente è la stessa, quindi lavorare direttamente con l'SDK in boundle spesso non è molto utile: è molto meglio partire dalla versione J2SE e aggiungere l'ultima versione delle API EE, a seconda delle proprie esigenze.

- Java 2 Micro Edition (J2ME): Java è nato come linguaggio portabile in grado di essere eseguito con ogni tipo di dispositivo. La J2ME include una JVM e un set di API e librerie appositamente limitate, per poter essere eseguite su piccoli dispositivi embedded, telefoni cellulari ed altro ancora. Questa configurazione deve essere scelta solo se si vogliono scrivere applicazioni per questo genere di dispositivi.

La procedura di installazione è in genere molto semplice, anche se sono necessarie alcuni piccoli accorgimenti per permettere un corretto funzionamento della JVM e dei programmi Java. Si limiterà l'attenzione alla distribuzione J2SE. Per chi fosse interessato a scrivere programmi J2EE non vi sono particolari operazioni aggiuntive da svolgere. Per la J2ME, invece, il processo è del tutto analogo, anche se si deve seguire un procedimento particolare a seconda del dispositivo scelto e della versione utilizzata.

I file per l'installazione possono essere trovati direttamente sul sito di Sun, come indicato in [SDK]. Altri produttori rilasciano JVM per piattaforme particolari (molto note e apprezzate sono quelle di IBM). Per la scelta di JVM diverse da quelle prodotte da Sun si possono seguire le indicazioni della casa produttrice o del particolare sistema operativo utilizzato.

Chi non fosse interessato a sviluppare applicazioni Java, ma solo a eseguire applicazioni già finite, potrà scaricare al posto dell'SDK il Java Runtime Environment (JRE) , che in genere segue le stesse edizioni e release dell'SDK. Non sempre il JRE è sufficiente: per esempio, se si volessero eseguire applicazioni JSP già pronte, potrebbe essere necessario mettere ugualmente a disposizione di Tomcat (o di un altro servlet-JSP engine) un compilatore Java, indispensabile per la compilazione delle pagine JSP.

Installazione su Windows

In ambiente Windows, in genere, il file di installazione è un eseguibile autoinstallante, che guida l'utente nelle varie fasi della procedura.

Non ci sono particolari aspetti da tenere in considerazione, a parte la directory di installazione e le variabili d'ambiente da configurare. Per la prima questione, a volte l'installazione nella directory "program files" può causare problemi di esecuzione ad alcuni applicativi Java che utilizzano la JVM di sistema (per esempio Tomcat o JBoss). Per questo, si consiglia di installare in una directory con un nome unico e senza spazi o altri caratteri speciali ("c:\programs", "c:\programmi" o semplicemente "c:\java").

Per poter funzionare, una qualsiasi applicazione Java deve sapere dove è installato il JDK, e quindi conoscere il path dell'eseguibile java (l'interprete), di javac (il compilatore usato da Tomcat per compilare le pagine JSP) e di altri programmi inclusi nel JDK.

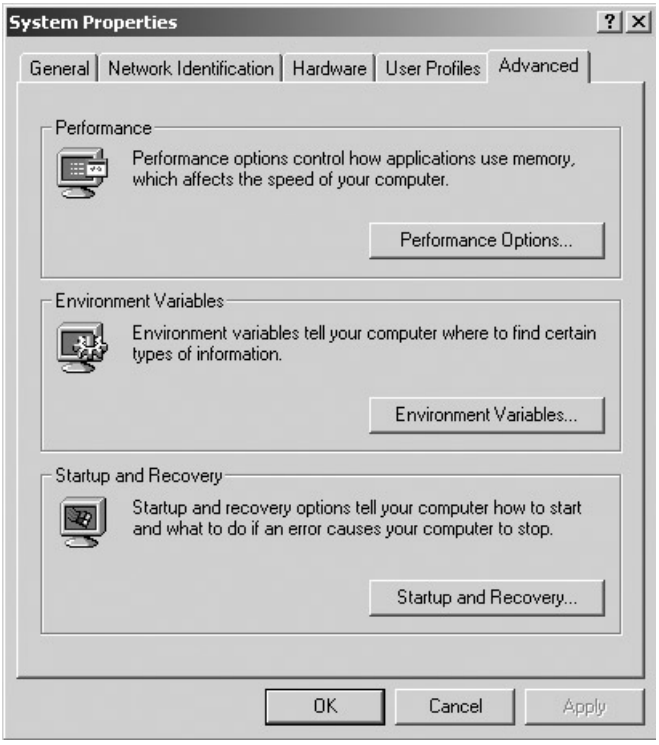
Inoltre, un programma Java deve anche poter ricavare la variabile d'ambiente CLASSPATH, all'interno della quale dovranno essere inseriti i riferimenti ai vari package utilizzati (directory scompattate, file .jar o .zip). Al momento dell'installazione, il classpath viene automaticamente impostato in modo da contenere le librerie di base del Java SDK (in genere, nella sottodirectory jre/lib, o semplicemente lib).

A partire dal JDK 1.1 è invalsa l'abitudine di utilizzare la variabile JAVA_HOME, che deve puntare alla directory di installazione del JDK. Di conseguenza, il path di sistema dovrà essere impostato in modo che punti alla directory %JAVA_HOME%\bin.

Di norma, queste impostazioni sono effettuate in modo automatico dal programma di installazione, ma possono essere facilmente modificate o impostate ex-novo tramite il pannello di controllo di Windows.

Per esempio, aprendo la finestra per l'impostazione delle variabili d'ambiente dal pannello di controllo...

Figura A.1 – *In Windows il pannello di controllo permette di impostare le diverse variabili d'ambiente.*



...si può procedere a inserire sia la variabile JAVA_HOME (in questo caso c:\programs\jdk1.4) sia la directory con gli eseguibili nel path (%JAVA_HOME%\bin).

Figura A.2 – *Come impostare la variabile JAVA_HOME in modo che punti alla directory di installazione del JDK.*

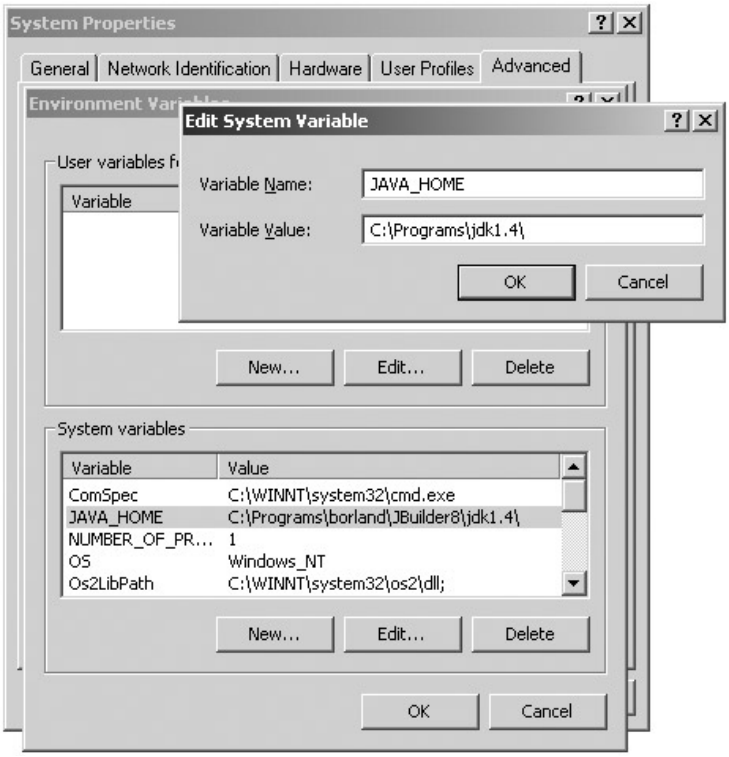
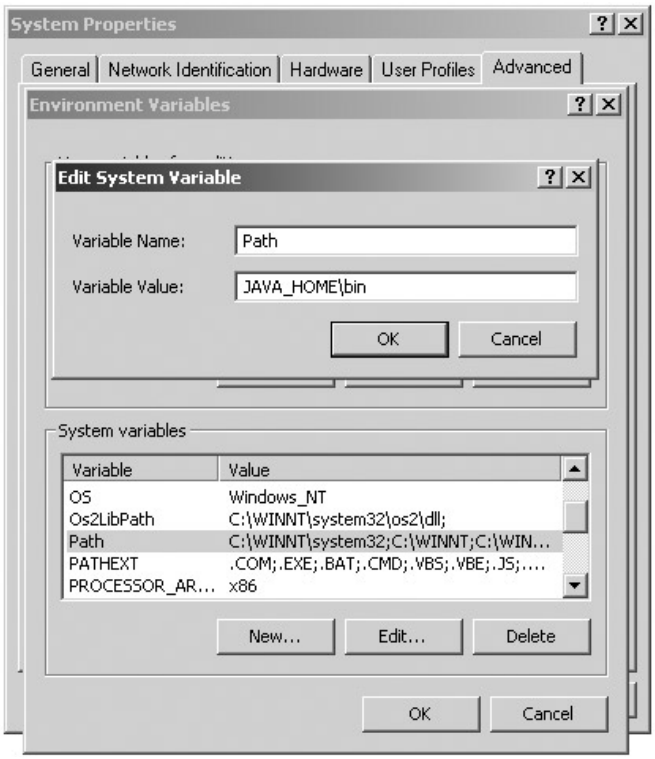


Figura A.3 – *Come impostare il path in modo che includa la dir* JAVA_HOME\bin.



Se tutto è stato fatto come si deve, aprendo una console DOS si può verificare la correttezza delle impostazioni inserite.

Per esempio, per conoscere il contenuto della variabile JAVA_HOME si potrà scrivere:

```
C:\>echo %JAVA_HOME%
C:\programs\jdk1.4
```

Il comando path, invece, mostrerà Tra le altre cose:

```
C:\> path
PATH=…….C:\programs\jdk1.4\bin
```

A questo punto, si può provare a eseguire la JVM con il comando:

```
C:\>java -version
```

L'opzione –version permette di conoscere la versione della JVM installata. In questo caso, il comando restituisce il seguente output:

```
java version "1.4.1"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1-b21)
Java HotSpot(TM) Client VM (build 1.4.1-b21, mixed mode)
```

se si usa un sistema basato su Windows 95-98 o ME, l'impostazione delle variabili d'ambiente può essere fatta tramite il file autoexec.bat. In questo caso, con un'istruzione del tipo:

```
set JAVA_HOME="…."
```

Si potrà definire la variabile in modo che punti alla directory indicata. Si tenga presente che tali sistemi operativi offrono un supporto ridotto per lo sviluppo e l'esecuzione di applicazioni che fanno uso dei protocolli di rete (socket TCP, database ecc...) o server side (servlet, web application, EJB, per esempio). Si consiglia pertanto di utilizzare le versioni più evolute (NT, 2000, XP), che supportano in modo più corretto e completo lo strato di network TCP/IP e offrono maggiori servizi.

Installazione su Linux

Per l'installazione su Linux, la procedura è molto simile a quella per Windows: si deve scaricare un file di installazione e installarlo. Per quest'ultimo aspetto si può utilizzare un rpm autoinstallante o un file eseguibile con estensione .bin.

Per esempio, se j2sdk-1.4.2-nb-3.5-bin-linux.bin è il file installante scaricato dal sito Sun, per prima cosa lo si renda eseguibile con il comando:

```
chmod o+x j2sdk-1.4.2-nb-3.5-bin-linux-i586.bin
```

quindi lo si mandi in esecuzione tramite:

```
./j2sdk-1.4.2-nb-3.5-bin-linux-i586.bin
```

per eseguire l'installazione. Di norma, questo porterà all'installazione dell'SDK in una directory il cui nome segue lo schema usr/java/jdk-<version-number>.

Questo significa che dovranno essere modificate di conseguenza le variabili JAVA_HOME e PATH, intervenendo sui file di profilo .bashrc o .bash_properties (a seconda del tipo di shell usata) dell'utente che dovrà usare Java:

```
JAVA_HOME=/usr/java/jdk1.4.1/
export JAVA_HOME
PATH=$JAVA_HOME/bin:$PATH
export PATH
```

Nel caso in cui un'applicazione debba far uso di altri package oltre a quelli di base del Java SDK, come un parser XML Xerces (contenuto in genere in un file xerces.jar), il package Java Mail, la Servlet API o altro ancora, si dovrà aggiungere manualmente al classpath il contenuto di tali librerie. Questo può essere fatto in due modi.

Il primo sistema consiste nell'aggiungere tali librerie al classpath di sistema, tramite il pannello di controllo di Windows o mediante l'impostazione ed esportazione di una variabile globale su Linux. In questo caso si potrà essere sicuri che tutte le applicazioni che dovranno utilizzare un parser Xerces o JavaMail potranno funzionare correttamente senza ulteriori impostazioni.

Attualmente, però, lo scenario Java è molto complesso, quindi un'impostazione globale difficilmente si adatta a tutte le applicazioni: in un caso potrebbe essere necessaria la versione 1.0 di Xerces, mentre un'altra applicazione potrebbe funzionare solo con la 1.2. Per questo motivo, in genere, si preferisce impostare un classpath personalizzato per ogni applicazione, passando tale configurazione alla JVM con il flag –classpath o –cp. Per esempio, in Windows si potrebbe scrivere:

```
set MY_CP=c:\programs\mokabyte\mypackages.jar
java –cp %MY_CP% com.mokabyte.mokacode.TestClasspathApp
```

Dove TestClasspathApp potrebbe essere un'applicazione che abbia bisogno di una serie di classi e interfacce contenute in mypackages.jar.

In questo modo si potranno costruire tutti i classpath personalizzati, concatenando file e directory di vario tipo.

In ambito J2EE le cose si complicano: entrano infatti in gioco il tipo di applicazione e le regole di caricamento del classloader utilizzato. Per questi aspetti, che comunque riguardano il programmatore esperto, si rimanda alla documentazione del prodotto utilizzato, e si consiglia l'adeguamento alle varie convenzioni imposte dalla specifica Java.

Bibliografia

[SUN] – Sito web ufficiale di Sun dedicato a Java: http://java.sun.com

[SDK] – Sito web di Sun per il download dell'SDK nelle versioni per le varie piattaforme: http://java.sun.com/downloads

[JIBM] – Sito web IBM dedicato a Java: http://www.ibm.com/java

[xIBM] – "IBM Developer Kit for Linux: Overview": http://www-106.ibm.com/developerworks/java/jdk/linux140/?dwzone=java

Appendice B

Ginipad, un ambiente di sviluppo per principianti

Andrea Gini

Ginipad è un ambiente di sviluppo per Java realizzato da Andrea Gini, uno degli autori di questo manuale. Ginipad è stato pensato come strumento per principianti, che non hanno tempo o voglia di barcamenarsi tra editor testuali e tool a riga di comando. La sua interfaccia grafica semplice ed essenziale ne ha decretato il successo anche presso utenti più esperti, che spesso necessitano di uno strumento rapido e leggero da alternare agli ambienti di sviluppo più complessi.

Ginipad è stato progettato per offrire il massimo grado di funzionalità nel modo più semplice e intuitivo possibile. Bastano cinque minuti per prendere confidenza con l'ambiente e le sue funzioni. Questa appendice fornisce una tabella riassuntiva dei principali comandi e una guida all'installazione.

Si consiglia di visitare la home page del progetto per trovare tutte le informazioni necessarie:

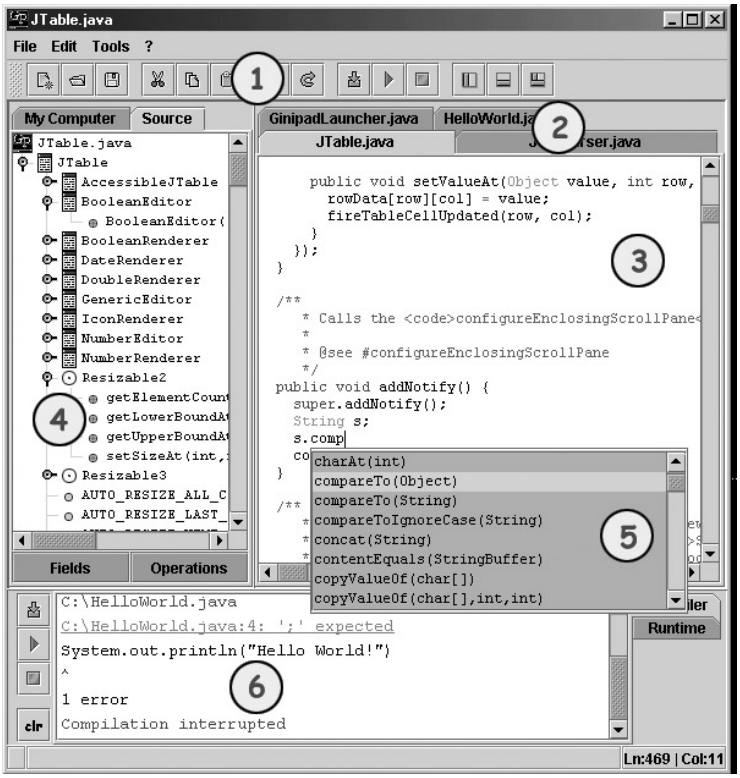
http://www.mokabyte.it/ginipad

Il tempo di apprendimento può essere ridotto ad appena cinque minuti grazie a uno slideshow in PowerPoint, disponibile all'indirizzo:

http://www.mokabyte.it/ginipad/download/GinipadVisualTutorial.ppt

Caratteristiche principali

Figura B.1 – *Caratteristiche principali di Ginipad.*



1. Pochi pulsanti facili da identificare.
2. Possibilità di lavorare su più di un documento.
3. Editor con Syntax Highlight.
4. Indice navigabile di metodi, campi e classi interne.
5. Autocompletamento delle dichiarazioni.
6. Hyperlink verso gli errori.

Tabella riassuntiva dei comandi

File			
	New	(Ctrl-N)	Crea un nuovo sorgente Java.
	Open	(Ctrl-O)	Carica un sorgente da disco.
	Save As	(Ctrl-S)	Salva il documento corrente.
	Close	(Ctrl-W)	Chiude il documento corrente.
	Open All		Apre in una volta sola gli ultimi otto sorgenti.
	Exit		Chiude il programma.
Edit			
	Cut	(Ctrl-X)	Taglia il testo selezionato.
	Copy	(Ctrl-C)	Copia il testo selezionato.
	Paste	(Ctrl-V)	Incolla il testo contenuto nella clipboard.
	Select All	(Ctrl-A)	Seleziona tutto il contenuto dell'editor.
	Undo	(Ctrl-Z)	Annulla l'ultima modifica.
	Redo	(Ctrl-Y)	Ripristina l'ultima modifica.
	Find	(Ctrl-F)	Apre la finestra di dialogo Find.
	Replace	(Ctrl-R)	Apre la finestra di dialogo Replace.
Tools			
	Compile	(Ctrl-Shift-C)	Compila il documento corrente.
	Run	(Ctrl-Shift-R)	Esegue il documento corrente.
	Stop		Interrompe l'esecuzione del processo corrente.
	Format source code	(Ctrl-Shift-F)	Esegue una formattazione del codice.
Console			
	Hide Tree		Nasconde il componente ad albero.
	Show Tree		Mostra il componente ad albero.
	Hide Console		Nasconde la console.
	Show Console		Mostra la console.
	Show Panels		Mostra tutti i pannelli.

	Full Screen		Espande l'editor a pieno schermo.
clr	Clear Console		Ripulisce la console.
Dialog			
	Preferences		Apre la finestra delle preferenze.
	Help		Apre la finestra di Help.
	About		Apre la finestra di About.

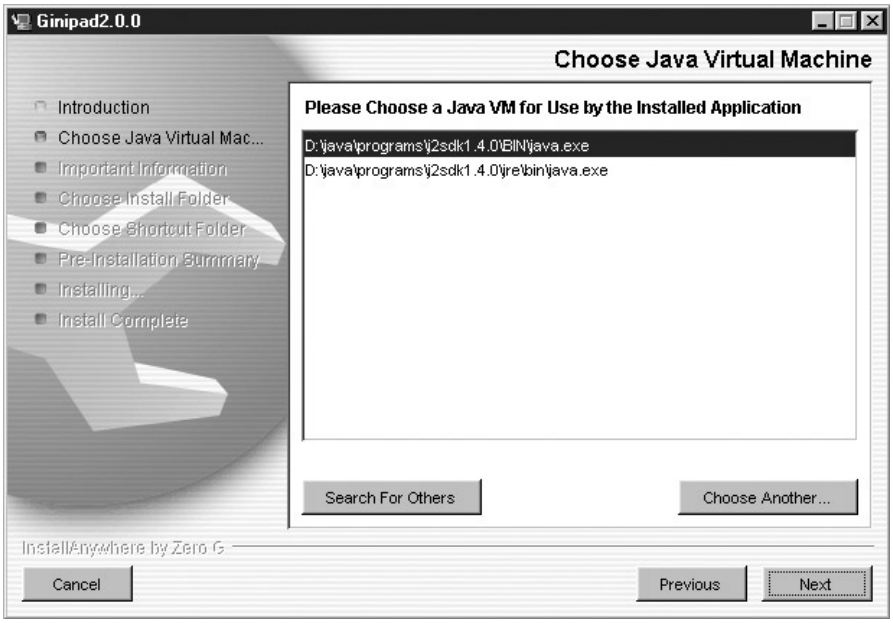
Installazione

1. Caricare e Installare il JDK 1.4.

2. Lanciare il file di setup di Ginipad.

3. Al secondo passaggio della fase di installazione verrà richiesto di scegliere la Virtual Machine. Si consiglia di scegliere quella presente nella cartella \bin del JDK, come si vede in Fig B.2.

Figura B.2 – *Scelta della Virtual Machine.*

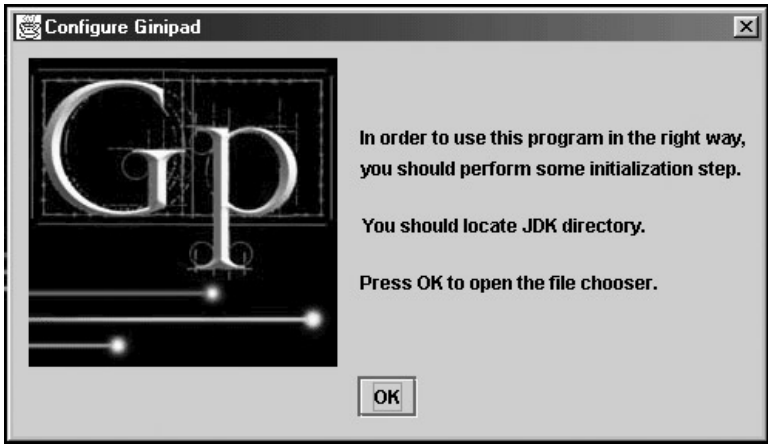


4. Al termine dell'installazione si può avviare il programma. Al primo avvio, Ginipad effettua una ricerca per localizzare la posizione del JDK. Tale processo è automatico e trasparente all'utente.

Cosa fare se Ginipad non trova il JDK

Ginipad è in grado di identificare da solo la posizione del JDK su disco, durante la fase di installazione. Tuttavia, se tale posizione dovesse cambiare, per esempio in seguito a un aggiornamento del JDK, all'avvio successivo verrà richiesto di indicare la nuova posizione dell'ambiente di sviluppo.

Figura B.3 - *La finestra per aprire il File Chooser.*



Dopo aver dato l'OK, verrà visualizzata una finestra File Chooser, tramite la quale si dovrà localizzare la directory del JDK sul disco. Una volta trovata la cartella, non resta che premere il pulsante Locate JDK Directory.

Appendice C

Parole chiave

Andrea Gini

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	final	native	this
case	float	package	throw
char	for	private	throws
class	(goto)	protected	transient
(const)	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

La maggior delle parole riservate di Java deriva dal C, il linguaggio dal quale Java ha ereditato la sintassi delle strutture di controllo. Le somiglianze con il C++, al contrario, sono minori, dal momento che Java adotta una sintassi differente per quanto riguarda i costrutti caratteristici della programmazione a oggetti. Le parole chiave goto e const, presenti nel C, fanno parte dell'insieme delle keyword, ma di fatto non compaiono nel linguaggio Java: in questo modo, il compilatore può segnalare uno speciale messaggio di errore se il programmatore le dovesse utilizzare inavvertitamente.

Appendice D

Diagrammi di classe e sistemi orientati agli oggetti

Andrea Gini

Un effetto della strategia di incapsulamento è quello di spingere il programmatore a esprimere il comportamento di un sistema a oggetti unicamente attraverso l'interfaccia di programmazione delle classi. In questo senso, quando un programmatore si trova a dover utilizzare una libreria di classi realizzate da qualcun altro, non è interessato a come essa sia stata effettivamente implementata: di norma, è sufficiente conoscere le firme dei metodi, le relazioni di parentela tra le classi, le associazioni e le dipendenze, informazioni che non dipendono dall'implementazione dei singoli metodi.

Il diagramma di classe è un formalismo che permette di rappresentare per via grafica tutte queste informazioni, nascondendo nel contempo i dettagli di livello inferiore. L'uso dei diagrammi di classe permette di vedere un insieme di classi Java da una prospettiva più alta rispetto a quella fornita dal codice sorgente, simile a quella che si ha quando si guarda una piantina per vedere com'è fatta una città. La piantina non contiene tutti i dettagli della zona rappresentata, come la posizione delle singole abitazioni o dei negozi, ma riporta informazioni sufficienti per orientarsi con precisione.

I diagrammi di classe fanno parte di UML (Unified Modeling Language), un insieme di notazioni grafiche che permette di fornire una rappresentazione dei diversi aspetti di un sistema software orientato agli oggetti, indipendentemente dal linguaggio di programmazione effettivamente utilizzato. L'UML comprende sette tipi diversi di diagrammi, che permettono di modellare i vari aspetti dell'architettura e del comportamento di un sistema software prima di iniziarne lo sviluppo. I diagrammi UML costituiscono una parte fondamentale della documentazione di un sistema informativo, e forniscono una guida essenziale in fase di studio o di manutenzione del sistema stesso.

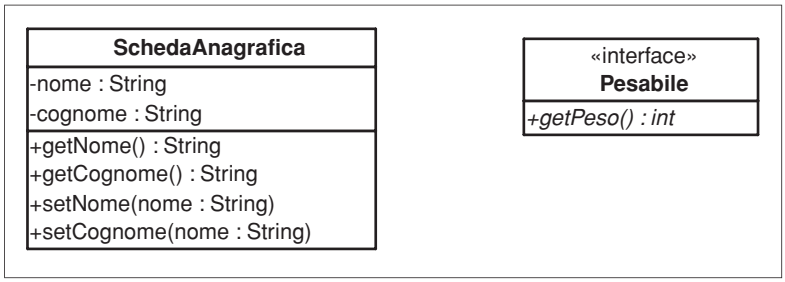
L'UML non è un linguaggio di programmazione, anche se negli ultimi anni gli ambienti di sviluppo hanno iniziato a includere strumenti che permettono di produrre codice a partire dai

diagrammi e viceversa. I seguenti paragrafi vogliono fornire una guida essenziale ai diagrammi di classe, l'unico formalismo UML presente in questo libro.

Classi e interfacce UML

In UML le classi e le interfacce sono rappresentate come rettangoli, suddivisi in tre aree: l'area superiore contiene il nome della classe o dell'interfaccia, quella intermedia l'elenco degli attributi e quella inferiore l'elenco dei metodi:

Figura D.1 – *Un esempio di classe e di interfaccia in UML.*



Entrambi i diagrammi non contengono alcun dettaglio sul contenuto dei metodi: il comportamento di una classe o di un'interfaccia UML è espresso unicamente tramite il nome dei suoi metodi. Le firme di metodi e attributi seguono una convenzione differente rispetto a quella adottata in Java: in questo caso, il nome precede il tipo, e tra i due compare un simbolo di due punti (:) come separatore. I parametri dei metodi, quando presenti, seguono la stessa convenzione. Il simbolo più (+), presente all'inizio, denota un modificatore public, mentre il trattino (-) indica private e il cancelletto (#) significa protected.

Il diagramma di interfaccia presenta alcune differenze rispetto a quello di classe:

- Al di sopra del nome compare un'etichetta "interface".
- Gli attributi (normalmente assenti) sono sottolineati, a indicare che si tratta di attributi statici immutabili.
- I metodi sono scritti in corsivo, per indicare che sono privi di implementazione.

Si osservi una tipica implementazione Java del diagramma di classe presente in figura 1:

```
public class SchedaAnagrafica {
```

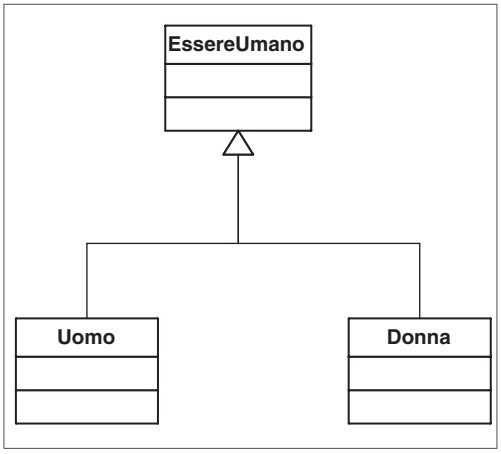
```
    private String nome;
    private String cognome;

    public String getNome() {
      return nome;
    }
    public void setNome(String nome) {
            this.nome = nome;
    }
    public String getCognome() {
      return cognome;
    }
    public void setCognome(String cognome) {
      this.cognome = cognome;
    }
}
```

Spesso il diagramma di classe presenta un livello di dettaglio inferiore rispetto al codice sottostante: tipicamente, si usa un diagramma per descrivere un particolare aspetto di una classe, e si omettono i metodi e gli attributi che non concorrono a definire tale comportamento. In questo libro, i diagrammi di classe sono stati disegnati secondo questa convenzione.

Ereditarietà e realizzazione

L'ereditarietà è rappresentata in UML con una freccia dalla punta triangolare, che parte dalla classe figlia e punta alla classe padre:

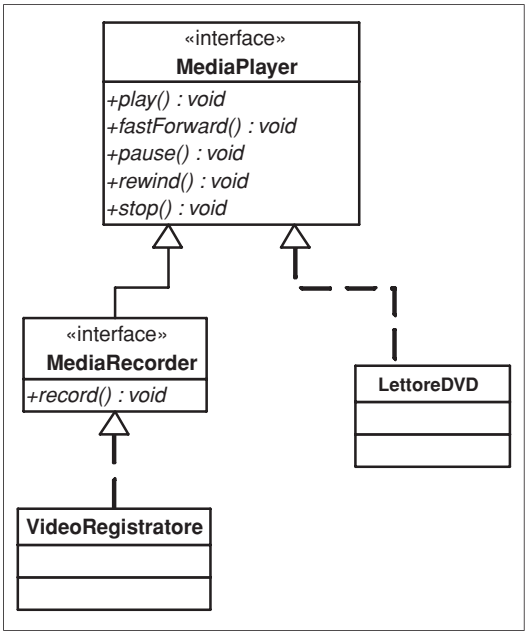
Figura D.2 – *Ereditarietà tra le classi.*



La realizzazione, equivalente all'implementazione di un'interfaccia in Java, viene rappresentata con una freccia simile a quella usata per l'ereditarietà, ma tratteggiata. Si noti che si ricorre alla realizzazione solo quando una classe implementa un'interfaccia, mentre se un'interfaccia ne estende un'altra si utilizza la normale ereditarietà.

In figura D.3 è possibile vedere un diagramma di classe contenente una relazione di ereditarietà tra interfacce (l'interfaccia MediaRecorder è figlia dell'interfaccia MediaPlayer) e due casi di realizzazione (la classe LettoreDVD realizza l'interfaccia MediaPlayer, mentre la classe VideoRegistratore realizza MediaRecorder).

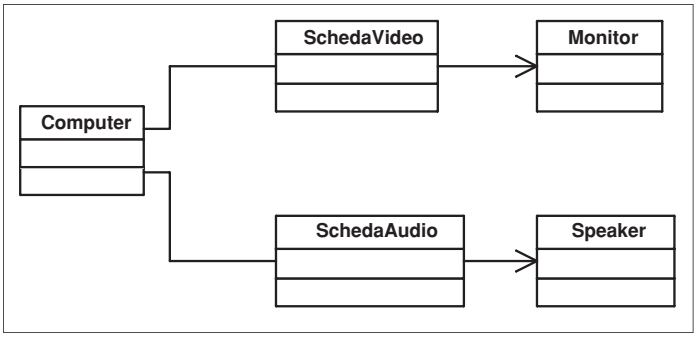
Figura D.3 – *Un diagramma che contiene sia l'ereditarietà sia la realizzazione.*



Associazione

L'associazione, rappresentata da una linea che congiunge due classi, denota una relazione di possesso. Un'associazione può essere bidirezionale o unidirezionale. Nel secondo caso, al posto di una linea semplice si utilizza una freccia. La freccia indica la direzione del flusso della comunicazione: in pratica, la classe da cui parte la freccia può chiamare i metodi di quella indicata dalla punta, ma non viceversa. L'equivalente Java dell'associazione è la presenza di un attributo in una classe, che di fatto denota il possesso di un particolare oggetto e la possibilità di invocare metodi su di esso.

Figura D.4 – *Classi unite da associazioni.*

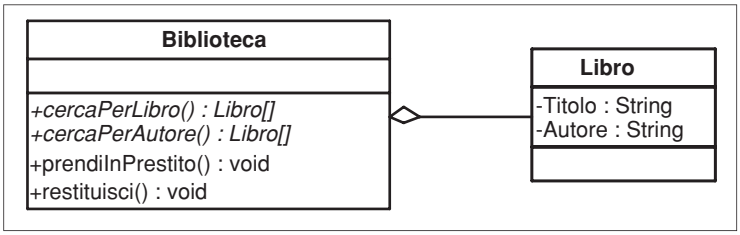


In figura D.4 è possibile osservare un insieme di classi caratterizzate da associazioni sia uni- sia bidirezionali: un computer è collegato alle schede audio e video da associazioni bi direzionali, a indicare che la comunicazione avviene in entrambe le direzioni; le due schede, invece, presentano un'associazione unidirezionale rispettivamente con gli speaker e il monitor, poiché non è permessa la comunicazione in senso inverso.

Aggregazione

Un tipo speciale di associazione è l'aggregazione, rappresentata da una linea tra due classi con un'estremità a diamante, che denota un'associazione uno a molti. In figura D.5 si può osservare una relazione uno a molti tra una biblioteca e i libri in essa contenuti.

Figura D.5 – *Un esempio di aggregazione.*

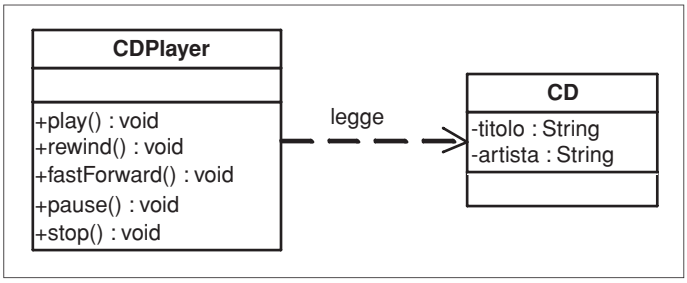


A parte la cardinalità, l'aggregazione equivale a un'associazione: nell'esempio di figura D.5 la classe Biblioteca possiede una collezione di libri e può invocare metodi su ognuno di essi. In Java, l'aggregazione corrisponde solitamente a un attributo di tipo Vector o HashTable, o più semplicemente a un array.

Dipendenza

La dipendenza è rappresentata da una freccia tratteggiata. Letteralmente, la dipendenza suggerisce che la classe puntata dalla freccia esista indipendentemente dalla classe da cui parte la freccia: in Java, questo significa che la prima può essere compilata e utilizzata anche in assenza della seconda.

Figura D.6 – *Relazione di dipendenza in UML.*



La figura D.6 presenta un esempio di relazione di dipendenza: il CD, inteso come supporto per musica e dati, esiste indipendentemente dal particolare lettore con cui lo si legge: le sue caratteristiche sono definite da un documento denominato Red Book, al quale si devono attenere i produttori di lettori CD. Si noti l'etichetta "play" che compare sopra la freccia: le etichette permettono di fornire maggiori informazioni sul tipo di relazione che sussiste tra due classi.