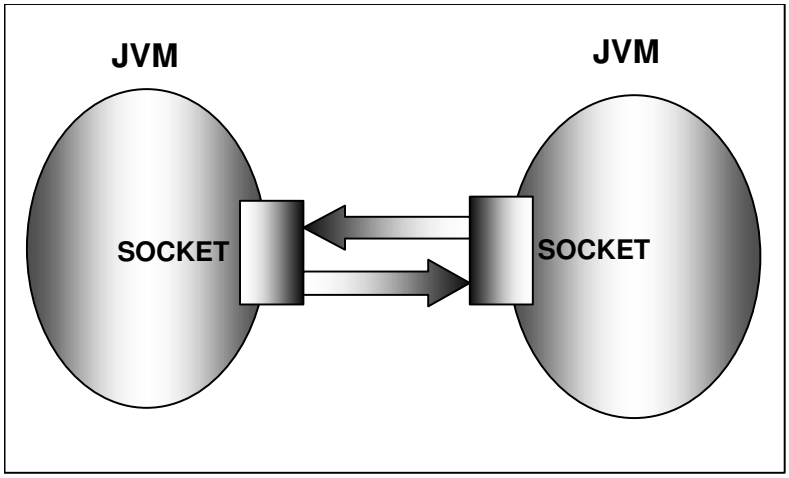
Graziano Frosini
Alessio Vecchio

PROGRAMMARE IN JAVA
VOLUME II

PROGRAMMAZIONE DI RETE
INTERFACCE GRAFICHE
STRUTTURE DATI



PROGRAMMARE IN JAVA
Volume II

POGRAMMAZIONE DI RETE
INTERFACCE GRAFICHE
STRUTTURE DATI

INDICE

Prefazione

In questo secondo volume di *Programmare in Java* vengono illustrate le tecniche di programmazione di rete, vengono descritte le interfacce grafiche e vengono trattate le più importanti strutture dati.

Più precisamente, il Capitolo 1 presenta le tecniche di comunicazione fra i processi di una applicazione distribuita, con riferimento ai protocolli di trasporto TCP e UDP comunemente utilizzati in Internet. Il Capitolo 2 affronta la stessa problematica della comunicazione ricorrendo all'invocazione di metodi remoti (RMI: *Remote Method Invocation*). Il Capitolo 3 descrive lo sviluppo di interfacce grafiche, facendo uso degli strumenti messi a disposizione dalle librerie AWT e Swing. Il Capitolo 4 illustra le tecniche di realizzazione di Applet e Servlet, che estendono le funzionalità dei Web browser (lato cliente) e dei Web server (lato servitore). Infine, il Capitolo 5 tratta le più importanti strutture dati (liste, pile, code, tabelle, alberi), e descrive le realizzazioni messe a disposizione dal cosiddetto *Java collection framework*.

Ulteriori tematiche meriterebbero di essere sviluppate, nella consapevolezza che il linguaggio Java costituisce uno strumento di riferimento nella soluzione di molte problematiche legate alla programmazione distribuita. Occorre però portare avanti obiettivi raggiungibili nei tempi prestabiliti, col proposito di apportare perfezionamenti e aggiunte in un futuro che speriamo non remoto.

Come nel Volume I, gli argomenti di questo testo sono stati sviluppati con la scopo primario di descrivere tecniche per realizzare programmi, e non di riportare tutte le possibilità previste dal linguaggio Java.

È in preparazione un terzo volume, che conterrà una serie di esercizi relativi agli argomenti sviluppati nel Volume I e nel Volume II. Gli esempi

illustrati derivano da una lunga esperienza, maturata nell'insegnamento della programmazione in corsi universitari che passa per il Pascal, per il C e per il C++.

I numerosi esempi presentati in questo volume sono disponibili, suddivisi per Capitoli, al seguente indirizzo Internet:

http://www2.ing.unipi.it/LibroJava

1. Programmazione di rete

1.1. Applicazioni e processi

Uno dei motivi che hanno prodotto il successo del linguaggio Java risiede nella semplicità con cui è possibile realizzare applicazioni distribuite, basate su protocolli comunemente utilizzati in Internet. In particolare, il linguaggio supporta il protocollo di rete IP (*Internet Protocol*), i protocolli di trasporto TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*), e il protocollo di livello applicazione HTTP (*Hypertext Transfer Protocol*) (Fig. 1.1).

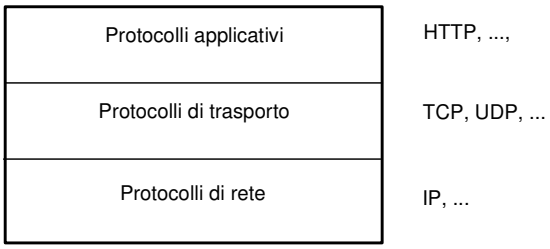


Figura 1.1. Livelli dei protocolli Internet

Una applicazione distribuita è costituita da più *processi*, dove un processo è ottenuto facendo eseguire il metodo *main()* di una classe da una JVM (tale classe, insieme ad altre eventualmente utilizzate, costituisce il

corpo del processo). I processi non sono soggetti a nessun meccanismo di schedulazione esplicitamente previsto dal linguaggio: il comando *java* (con la specifica di una classe che contiene il metodo *main()*) attiva una JVM che inizia ad eseguire un processo, la JVM porta avanti con continuità l'esecuzione del processo stesso, e si disattiva nel momento in cui il metodo *main()* termina. Ogni processo può a sua volta essere costituito da uno o più thread.

Su un computer della rete (*host*) possono essere in esecuzione una o più JVM: l'host, attraverso il suo sistema operativo, può eseguire autonomamente una schedulazione delle varie applicazioni, fra cui le JVM che sono in quel momento attive.

In Java, come in molti altri contesti, i processi si classificano in *processi servitori* (quelli che forniscono servizi) e *processi clienti* (quelli che richiedono servizi) (un processo può anche essere contemporaneamente sia cliente che servitore). La comunicazione fra processi avviene utilizzando classi appartenenti al package *java.net,* che va pertanto incluso.

1.2. La classe *InetAddress*

In Internet, ogni host è individuato da un indirizzo (*indirizzo IP*) a 32 bit, comunemente rappresentato attraverso 4 numeri naturali, da 0 a 255, seperati da "." (esempio: 131.114.9.226). L'indirizzo può anche essere costituito da un nome simbolico (esempio: *pc-frosini.iet.unipi.it*), che viene fatto corrispondere a un indirizzo IP da un apposito server di risoluzione dei nomi (*DNS: Domain Name Server*) (il nome simbolico può anche essere parziale, e mancare il nome del dominio a cui appartiene (esempio: *pc-frosini*)).

In Java, gli indirizzi di rete sono oggetti della classe *InetAddress*, la quale consente di mettere in corrispondenza nomi simbolici con indirizzi IP, utilizzando lo stesso servizio di risoluzione dei nomi adoperato dalla macchina locale.

La classe *InetAddress* non dispone di costruttori, e oggetti della classe possono essere ottenuti solo usando opportuni metodi statici della classe

stessa, alcuni dei quali sono riportati di seguito (la classe *UnknownHostException* è una sottoclasse di *IOException*):

public static InetAddress getByName (String** host **)
throws UnknownHostException
restituisce l'oggetto *InetAddress* relativo alla stringa specificata, che rappresenta o l'indirizzo IP dell'host (per esempio "*131.114.28.20*"), o il suo nome simbolico (per esempio "*docenti.ing.unipi.it*");

public static InetAddress getLocalHost ()
throws UnknownHostException
restituisce l'oggetto *InetAddress* associato all'host locale.

Una volta ottenuta un'istanza di *InetAddress*, è possibile recuperare l'indirizzo IP associato all'host oppure il suo nome simbolico, con i seguenti metodi:

public byte [] getAddress ()
resituisce l'indirizzo IP dell'host sotto forma di array di 4 byte;

public String getHostAddress ()
restituisce l'indirizzo IP dell'host sotto forma di stringa;

public String getHostName ()
restituisce il nome simbolico dell'host sotto forma di stringa (il nome del dominio può anche mancare, nel caso in cui sia lo stesso per l'oggetto a cui si applica il metodo e per l'host che esegue il processo).

Come esempio, riportiamo il seguente programma che, dopo aver creato alcuni oggetti *InetAddress* relativi all'host locale e a host individuati mediante indirizzi IP e indirizzi simbolici, ricava il nome degli host e il loro indirizzo:

```
import java.net.*;
public class IndIP
{ public static void main(String[] args)
  { try
    { InetAddress ia1 =
        InetAddress.getByName("131.114.9.226");
      InetAddress ia2 =
```

```
          InetAddress.getByName("docenti.ing.unipi.it");
        InetAddress ia3 = InetAddress.getLocalHost();
        Console.scriviStringa
               (ia1.getHostName());
        // stampa pc-frosini.iet.unipi.it,
        // oppure pc-frosini
        Console.scriviStringa(ia2.getHostAddress());
        // stampa 131.114.28.20
        Console.scriviStringa(ia3.getHostName());
        // stampa il nome della macchina locale
      }
      catch (UnknownHostException e)
      { Console.scriviStringa("Host sconosciuto"); }
   }
}
```

1.3. Risorse e accessi

Un URI (*Uniform Resource Identifier*) è una stringa di caratteri che identifica una risorsa (come per esempio un file) all'interno del World Wide Web. Un URL (*Uniform Resource Locator*) è un URI che identifica una risorsa evidenziando il meccanismo di accesso principale alla risorsa stessa, (per esempio la sua locazione) piuttosto che specificando il suo nome o le sue caratteristiche.

Un esempio di URL è il seguente:

http://docenti.ing.unipi.it/linguaggi/index.html

In forma semplificata, in un URL è possibile identificare:

- il protocollo da usare (*http*);
- l'host su cui risiede la risorsa *(docenti.ing.unipi.it*);
- il nome della risorsa (*/linguaggi/index.html*) (il significato del nome della risorsa dipende sia dal protocollo che dall'host).

Opzionalmente, in dipendenza dal protocollo, può essere specificata esplicitamente anche una *porta* (numero naturale a 16 bit), che individua il passaggio da usare nell'host remoto per comunicare con quella risorsa: se non specificata viene usata la porta di default del protocollo (per il protocollo *http*, la porta 80). Sempre in dipendenza del protocollo, può anche essere indicato un *riferimento di partenza* all'interno della risorsa (noi non useremo mai questa possibilità).

In Java un URL è rappresentato da una istanza della classe *URL.* Tale classe è dotata di costruttori che consentono di specificare una risorsa in modo assoluto o relativo. Tutti i costruttori possono lanciare eccezioni appartenenti alla classe *MalformedURLException,* sottoclasse di *IOException*.

Costruttori assoluti:

```
public URL ( String  specificatore )
public URL ( String protocollo , String  host , String file )
public URL ( String protocollo , String  host , int port , String file )
```

Per esempio si può scrivere:

```
URL url1 = new URL
   ("http://www.ing.unipi.it/linguaggi/index.html")
URL url2 = new URL
   ("http://www.ing.unipi.it:80/linguaggi/index.html)
URL url3 = new URL
   ("http", "www.ing.unipi.it", 80,
          "/linguaggi/index.html")
```

Costruttore relativo:

```
public URL ( URL contesto , String specificatore )
```

Per esempio si può avere:

```
URL url4 = new URL("http://www.ing.unipi.it");
URL url5 =  new URL(url4,"/linguaggi/index.html");
```

1.3.1 Interazioni con una risorsa

Una volta identificata una risorsa, è possibile interagire con essa attraverso il seguente metodo della classe *URL*:

public URLConnection openConnection () throws IOException

URLConnection è una classe astratta, superclasse di tutte le classi che rappresentano una connessione tra un'applicazione e un URL. Il metodo precedente restituisce un oggetto appartenente a quella sottoclasse di *URLConnection* che supporta il protocollo specificato. Tale oggetto viene quindi usato per inviare e ricevere dati. Due sottoclassi significative di *URLConnection* sono *HttpURLConnection*, che supporta il protocollo HTTP (vedi Appendice I), e *JarURLConnection*, che supporta una connessione con un file archivio (*jar*) o con un'entità in esso contenuta.

La classe *URLConnection* prevede metodi per accedere alla risorsa, per specificare le caratteristiche della richiesta e per conoscere le caratteristiche della risposta.

Metodi per accedere alla risorsa:

public void setDoInput (boolean b)
public void setDoOutput (boolean b)
abilitano o disabilitano l'oggetto *URLConnection* a effettuare operazioni di lettura o di scrittura, a seconda del valore dell'argomento (per default la connessione è abilitata in lettura e disabilitata in scrittura);

public abstract void connect () throws IOException
effettua la connessione con la risorsa (devono essere già specificate le opzioni della connessione); il metodo è implementato dalle sottoclassi di *URLConnection*;

public InputStream getInputStream () throws IOException
public OutputStream geOutputStream () throws IOException
restituiscono, rispettivamente, un oggetto stream tramite il quale si possono effettuare letture da un URL o scritture dati verso un URL.

Metodi per specificare le caratteristiche della richiesta:

public void setRequestProperty (String** prop **, String** val **)
imposta la proprietà *prop* con il valore *val*: nel caso del protocollo HTTP viene aggiunto un header del tipo "*prop: val*" alla richiesta.

Metodi per conoscere le caratteristiche della risposta:

public String getHeaderField (String** n **)
restituisce il valore dell'header che ha il nome specificato come argomento;

public String getContentType ()
restituisce il tipo del messaggio di risposta, come tipo MIME (*null* se il tipo è sconosciuto);

public int getContentLength ()
restituisce la lunghezza del messaggio di risposta (se questa non è nota restituisce *–1*).

La classe *HttpURLConnection* dispone di metodi specifici per il protocollo HTTP, tra cui:

public void setRequestMethod (String** m **)
imposta il tipo di richiesta HTTP da eseguire (il valore predefinito è "GET").

public int getResponseCode ()
restituisce il codice della risposta;

public String getResponseMessage ()
restituisce il messaggio di risposta.

Inoltre, tale classe possiede delle costanti di tipo intero corrispondenti ai possibili codici di risposta. Per esempio *HttpURLConnection.HTTP_OK* (valore 200), *HttpURLConnection.HTTP_NOT_FOUND* (valore 404), e così via.

Il procedimento che consente di accedere a una risorsa si compone delle seguenti fasi:

1. mediante il metodo *openConnection()* della classe *URL* viene creato un oggetto di tipo *URLConnection*;
2. attraverso metodi della classe *URLConnection* vengono impostate le caratteristiche della richiesta (per esempio, nel caso del protocollo HTTP, viene specificato il tipo della richiesta (*GET*, *HEAD*, eccetera);
3. attraverso il metodo *connect()* della classe *URLConnection* viene inviata la richiesta;
4. attraverso metodi della classe *URLConnection* è possibile i) conoscere i dettagli della risposta e ii) leggere dalla risorsa o scrivere nella risorsa, creando e utilizzando stream ottenuti dall'oggetto *URLConnection*.

La classe *URLConnection* esegue implicitamente una connessione all'URL specificato, anche senza che venga chiamato il metodo *connect()*, quando viene invocato un metodo che richiede la conoscenza del messaggio di risposta o quando si tenta di accedere alla risorsa.

Come esempio riportiamo un programma che ha lo scopo di connettersi all'URL *http://www.ing.unipi.it/~d3671/esami/index.html*, e di stampare su video (per righe) il contenuto del file che costituisce la risorsa:

```
import java.net.*;
import java.io.*;
class LeggiRisorsa
{ private URL u;
  private HttpURLConnection c;
  LeggiRisorsa(URL ur)
  { u = ur;  }
  void leggi()
  { try
    { c = (HttpURLConnection) u.openConnection();
      // la richiesta e` implicitamente GET
      c.connect();
      BufferedReader in = new BufferedReader
        (new InputStreamReader(c.getInputStream()));
      String linea;
      while ((linea = in.readLine()) != null)
        Console.scriviStringa(linea);
      in.close();
    }
```

```
    catch(IOException e)
    {  Console.scriviStringa("Problemi di accesso " +
                e.getMessage());
    }
  }
}

public class ProvaURL
{  public static void main(String[] args)
  {  try
    {  URL ss = new URL
          ("http://www.ing.unipi.it");
       URL uu = new URL
          (ss, "/~d3671/esami/index.html");
       LeggiRisorsa lr = new LeggiRisorsa(uu);
       lr.leggi();
    }
    catch(MalformedURLException e)
    {  Console.scriviStringa("URL non corretto");
    }
  }
}
```

Un altro esempio che mostra come usare la classe *HttpURLConnection* per inviare dati ad un URL è riportato nel paragrafo 4.3.

1.4. Comunicazione mediante socket

La comunicazione fra due processi, uno servitore e uno cliente, può essere gestita direttamente attraverso i protocolli di livello trasporto, senza cioè ricorrere a protocolli applicativi. In Internet i protocolli di trasporto comunemente utilizzati sono due: *TCP* e *UDP*. Il primo fornisce l'astrazione di una connessione bidirezionale e affidabile tramite la quale le parti coinvolte nella comunicazione sono in grado di scambiare *un flusso di byte*. Il secondo implementa un servizio in cui le parti comunicanti sono in grado di scambiare singoli pacchetti di dati (noti come *datagrammi*), senza garanzie di consegna e di ordinamento (non esiste in questo caso il

concetto di connessione). Per entrambi i protocolli, un punto terminale di comunicazione è identificato da un indirizzo IP e da un numero di porta. Un punto terminale di comunicazione è comunemente noto con il nome di *socket*. In questo paragrafo affrontiamo il problema di come far comunicare due processi di un applicazione Java attraverso il protocollo TCP, mentre l'impiego del protocollo UDP è oggetto del paragrafo successivo.

1.4.1. La classe *Socket*

In Java l'astrazione *socket TCP* è realizzata dalla classe *Socket*. Tale classe viene utilizzata da un processo cliente attraverso i costruttori:

```
public Socket ( InetAddress server , int portaServer )
                    throws IOException
public Socket ( String server , int portaServer )
                    throws UnknownHostException , IOException
```

creano un oggetto *Socket* già connesso al punto terminale di comunicazione remoto specificato dagli argomenti *server* e *portaServer*.

Una volta creato un oggetto *Socket*, è possibile comunicare con il processo servitore remoto mediante i due stream di byte, uno di ingresso e uno di uscita, ottenuti mediante i due metodi:

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

Quando un socket non viene più utilizzato è opportuno chiuderlo mediante il seguente metodo:

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

1.4.2. La classe *ServerSocket*

Per realizzare un processo servitore, si utilizza la classe *ServerSocket*, che definisce non tanto un socket, quanto un "*fabbricatore di socket*". Un oggetto *ServerSocket* viene creato attraverso il seguente costruttore:

public ServerSocket (int** porta **) throws IOException
crea un *ServerSocket* che si pone in ascolto sulla porta specificata.

A questo punto è possibile chiamare sull'oggetto *ServerSocket* il metodo:

public Socket accept () throws IOException
aspetta una richiesta di connessione da parte di un processo cliente.

Tale metodo è bloccante, ovvero non termina fin quando non arriva una richiesta. Quando questa arriva, esso genera un socket ausiliario connesso con il processo cliente. Il socket restituito dal metodo *accept()* è quello che viene poi effettivamente usato dal processo servitore nella comunicazione con il processo cliente, lasciando libero il *ServerSocket* di ricevere altre eventuali richieste di connessione.

La chiusura di un server socket avviene mediante il metodo:

public void void close () throws IOException

1.4.3. Connessione cliente-servitore

In sintesi, per effettuare una connessione tra un processo cliente e un processo servitore si deve operare come segue:

Processo servitore:

- crea un oggetto *ServerSocket* e si mette in ascolto su una determinata porta;
- si pone in attesa di una connessione da parte di un processo cliente mediante il metodo bloccante *accept()*;
- quando riceve una richiesta di servizio, il metodo *accept()* restituisce un oggetto *Socket*, che sarà poi adoperato dal processo servitore per comunicare con il processo cliente.

Processo cliente:

- crea un proprio oggetto *Socket* connesso con l'oggetto *ServerSocket* del processo servitore (le operazioni di creazione e di

connessione avvengono in maniera contestuale usando i costruttori visti in nel sottoparagrafo 1.4.1).

A questo punto il canale di comunicazione cliente-servitore è pronto per essere utilizzato: per scambiare dati è sufficiente che i due processi creino gli stream associati ai due socket, quindi leggere/scrivere su questi. Gli stream associati ai socket appartengono alle classi *InputStream* ed *OutputStream*: tipicamente, questi si racchiudono in stream di tipo *DataInputStream* e *DataOutputStream* se i dati da scambiare sono valori di un tipo primitivo, in stream di tipo *ObjectInputStream* e *ObjectOutputStream* se i dati da scambiare sono oggetti appartenenti a classi serializzabili, in stream di tipo *BufferedReader* e *PrintWriter* se i dati da scambiare sono testi. Per evitare problemi di blocco legati ai costruttori degli stream (letture iniziali di apposite marche), conviene che il processo servitore ottenga dal socket prima l'input stream e dopo l'output stream, e che un processo cliente ottenga dal socket detti stream in ordine invertito.

A titolo di esempio, riportiamo una semplice applicazione cliente-servitore. Il processo servitore, che risiede su un dato host (per esempio quello avente come indirizzo IP 131.114.9.226), si pone in attesa di una richiesta di servizio utilizzando un server socket in ascolto sulla porta 1234. Quando un processo cliente si connette, il processo servitore crea un socket ed entra in un ciclo in cui legge dallo stream di ingresso associato al socket un intero e, se diverso da zero, lo eleva al quadrato e scrive il risultato sullo stream di uscita associato al socket. La lettura del valore zero comporta l'uscita dal ciclo e quindi la terminazione del processo servitore.

```
import java.net.*;
import java.io.*;
class Servizio
{  void servi()
   { ServerSocket ss; Socket aus;
     DataInputStream ingresso; DataOutputStream uscita;
     try
     {  ss = new ServerSocket(1234);
        aus = ss.accept();
        ingresso = new DataInputStream(
                                  aus.getInputStream());
```

```
        uscita = new DataOutputStream(
                                  aus.getOutputStream());
        while(true)
        { int i = ingresso.readInt();
          if (i == 0) break;
          int u = i*i;
          uscita.writeInt(u);
        }
        ss.close(); aus.close();
      }
      catch(IOException e)
      { Console.scriviStringa(e.getMessage()); }
   }
}

public class Servitore
{ public static void main(String[] args)
   { Servizio se = new Servizio();
     se.servi();
   }
}
```

Il processo cliente utilizza un socket collegato con l'host su cui risiede il processo servitore, attraverso la porta 1234. Una volta connesso, il processo cliente entra in un ciclo in cui attende l'immissione di un intero da parte dell'utente, lo invia al processo servitore scrivendo sullo stream di uscita associato al socket e, se diverso da zero, legge il risultato dallo stream di ingresso associato al socket e lo stampa su video. L'immissione da parte dell'utente del valore zero, oltre al normale invio al processo servitore, causa anche la terminazione del ciclo e del processo cliente.

```
import java.net.*;
import java.io.*;
class Richiesta
{ private String IPServitore;
  Richiesta(String ip)
  { IPServitore = ip; }
  void interagisci()
  { int dato, risultato;
    Socket cli;
    DataInputStream ingresso;
    DataOutputStream uscita;
```

```
    try
    { cli = new Socket(IPServitore, 1234);
      ingresso = new DataInputStream(
                                cli.getInputStream());
      uscita = new DataOutputStream(
                                cli.getOutputStream());
      while (true)
      { dato = Console.leggiIntero();
        uscita.writeInt(dato);
        if (dato == 0) break;
        risultato = ingresso.readInt();
        Console.scriviIntero(risultato);
      }
      cli.close();
    }
    catch (UnknownHostException e)
    { Console.scriviStringa("Server sconosciuto"); }
    catch (IOException e)
    { Console.scriviStringa
             ("Connessione non stabilita");  }
  }
}
public class Cliente
{ public static void main(String[] args)
  { Richiesta ri = new Richiesta("131.114.9.226");
    ri.interagisci();
  }
}
```

L'esempio precedente può essere leggermente modificato, prevedendo che lo scambio di dati tra cliente e servitore riguardi oggetti appartenenti a una classe (serializzabile) *Uno*, avente solo un campo dati di tipo intero che il servitore eleva al quadrato:

```
// servitore
import java.net.*; import java.io.*;
class Uno implements Serializable
{ public int i; }
class Servizio
{ void servi()
  { ServerSocket ss; Socket aus;
    ObjectInputStream ingresso;
```

```
    ObjectOutputStream uscita;
    try
    { ss = new ServerSocket(1234); aus = ss.accept();
      ingresso =
        new ObjectInputStream(aus.getInputStream());
    uscita =
        new ObjectOutputStream(aus.getOutputStream());
      while(true)
      { Uno uu = (Uno)ingresso.readObject();
        if (uu.i == 0) break;
        uu.i = uu.i*uu.i;  uscita.writeObject(uu);
      }
      ss.close(); aus.close();
    }
    catch(ClassNotFoundException e)
    { Console.scriviStringa("Classe non trovata");   }
    catch(IOException e)
    { Console.scriviStringa(e.getMessage()); }
  }
}

public class Servitore
{ public static void main(String[] args)
  { Servizio se = new Servizio(); se.servi();
  }
}

// cliente
import java.net.*; import java.io.*;
class Uno implements Serializable
{ public int i; }
class Richiesta
{ private String IPServitore;
  Richiesta(String ip)
  { IPServitore = ip; }
  void interagisci()
  { int dato; Socket cli; Uno ou, ouu;
    ObjectInputStream ingresso;
    ObjectOutputStream uscita;
    try
    { cli = new Socket(IPServitore, 1234);
      uscita =
        new ObjectOutputStream(cli.getOutputStream());
```

```
        ingresso =
          new ObjectInputStream(cli.getInputStream());
        while (true)
        { dato = Console.leggiIntero(); ou = new Uno();
          ou.i = dato;
          uscita.writeObject(ou);
          if (dato == 0) break;
          ouu = (Uno)ingresso.readObject();
          Console.scriviIntero(ouu.i);
        }
        cli.close();
      }
      catch(ClassNotFoundException e)
        { Console.scriviStringa(e.getMessage());}
      catch (UnknownHostException e)
      { Console.scriviStringa("Server sconosciuto");   }
      catch (IOException e)
      { Console.scriviStringa("Connessione fallita"); }
   }
}

public class Cliente
{ public static void main(String[] args)
   { Richiesta ri = new Richiesta("131.114.9.226");
     ri.interagisci();
   }
}
```

1.4.4. Servitore organizzato a thread

Il processo servitore illustrato nel sottoparagrafo precedente è in grado di servire un solo processo cliente alla volta. Ciò può costituire una limitazione, in quanto il numero dei clienti non è generalmente noto a priori e varia dinamicamente.

Una soluzione al problema consiste nello strutturare il processo servitore in modo che abbia più flussi di esecuzione (thread). Un primo thread rimane costantemente in attesa dell'arrivo di richieste di servizio da parte dei clienti. Ogni volta che arriva una richiesta, viene generato un thread ausiliario avente lo scopo di servire il processo cliente che si è

appena connesso. I thread ausiliari terminano la loro esecuzione una volta concluso il servizio richiesto dal processo cliente.

Il processo servitore può quindi essere organizzato secondo il seguente schema di funzionamento:

```
while (true)
{ /*   attende, sul metodo accept() relativo al
       ServerSocket, l'arrivo di una richiesta di
       servizio da parte di un cliente; */
  /*   quando un cliente si connette, viene creato
       un nuovo Socket, sia aus; */
  /*   crea un nuovo thread ausiliario e gli passa come
       argomento attuale aus; */
}
```

Il thread ausiliario:

- comunica con il processo cliente attraverso il socket *aus,* ricopiando il suo riferimento in quello di un suo socket interno sia *s*;
- dopo aver compiuto l'elaborazione, termina chiudendo *s* (e quindi *aus*).

È importante notare come un'organizzazione di questo tipo non abbia nessuna ripercussione su un processo cliente, il quale non si accorge del fatto che il processo servitore è organizzato a thread.

A titolo di esempio, riportiamo una applicazione con un processo servitore e più processi clienti che hanno lo stesso corpo (più JVM eseguono il metodo *main()* della stessa classe). Il processo servitore è composto da due classi, *Servizio* e *ServitoreThread*. La classe *Servizio* implementa le funzioni da eseguire all'interno dei thread ausiliari. A tale scopo, essa è definita come sottoclasse di *Thread* ed è dotata di un metodo costruttore che consente di inizializzare i thread (di tipo *Servizio*) con il riferimento al *Socket* da usare per comunicare con il processo cliente. Una volta attivato, un thread ausiliario esegue un ciclo in cui riceve dal processo cliente a cui è connesso una stringa e la rimanda al processo cliente stesso. Il thread ausiliario termina la sua esecuzione quando il processo cliente invia la stringa "*fine*".

La classe *ServitoreThread* crea un *ServerSocket* sulla porta 1234, quindi entra in un ciclo in cui i) attende l'arrivo di una richiesta di servizio,

ii) crea una nuova istanza di *Servizio* (tramite il costruttore le passa il riferimento al *Socket* da usare).

```
import java.net.*; import java.io.*;
class ServizioT extends Thread
{ private Socket so;
  private BufferedReader ingresso;
  private PrintWriter uscita;
  ServizioT(Socket s) throws IOException
  { so = s;
    ingresso = new BufferedReader(new
      InputStreamReader(so.getInputStream()));
    uscita =
      new PrintWriter(so.getOutputStream(), true);
    start();
  }
  void run()
  { String st;
    try
    { while(true)
      { st = ingresso.readLine();
        if (st.equals("fine")) break;
      }
      so.close();   // quindi anche s.close
      }
    catch(IOException e)
    { Console.scriviStringa(e.getMessage()); }
  }
}

public class ServitoreThread
{ public static void main(String[] args)
  { ServerSocket ss;
    try
    { ss = new ServerSocket(1234);
      while(true)
      { Socket aus = ss.accept();
        // aspetta un cliente
        new ServizioT(aus);
      }
    }
    catch(IOException e)
    { Console.scriviStringa(e.getMessage()); }
```

```
   }
}
```

Un processo cliente si connette al processo servitore supponendo che quest'ultimo sia in esecuzione sull'host avente indirizzo IP 131.114.9.226. Il cliente, una volta connesso, entra in un ciclo in cui i) legge da tastiera una stringa *s*, ii) invia *s* al processo servitore scrivendo sullo stream di uscita associato al socket utilizzato, iii) se *s* è diversa dalla stringa "*fine*" legge dallo stream di ingresso del socket utilizzato una stringa che costituisce la risposta del processo servitore e ricomincia il ciclo, altrimenti termina l'esecuzione del ciclo.

```
import java.io.*; import java.net.*;
class RichiestaT
{ String IPServitore;
  RichiestaT(String s)
  { IPServitore = s;  }
  void fai()
  { Socket so;
    PrintWriter uscita ; BufferedReader ingresso;
    try
    { so = new Socket(IPServitore, 1234);
      uscita =
        new PrintWriter(so.getOutputStream(), true);
      ingresso = new BufferedReader(new
        InputStreamReader(so.getInputStream()));
      String s;
      while (true)
      { s = Console.leggiStringa(); uscita.println(s);
        if (s.equals("fine")) break;
        s = ingresso.readLine();
        Console.scriviStringa("risposta: " + s);
      }
      so.close();
    }
    catch (IOException e)
    { Console.scriviStringa(e.getMessage()); }
  }
}

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

```
  { RichiestaT re = new RichiestaT ("131.114.9.226");
    re.fai();
  }
}
```

1.5. Comunicazione mediante datagrammi

Il protocollo UDP consente ai processi di una applicazione di comunicare mediante lo scambio di singoli pacchetti, detti *datagrammi*. Non vi è nessuna garanzia sulla effettiva consegna dei datagrammi, né che l'ordine di consegna sia legato all'ordine in cui vengono inviati. Tale protocollo è comunemente usato in alcune tipologie di applicazioni, quali la trasmissione di dati audio/video e nel multicast.

Java offre supporto alla comunicazione basata su UDP attraverso le classi *DatagramSocket* e *DatagramPacket.* La prima classe è utilizzata dal processo servitore e dal processo cliente per inviare e ricevere datagrammi (a differenza del protocollo TCP, non esiste una classe specifica da impiegare nella realizzazione del processo servitore). La seconda classe è utilizzata dal processo servitore e dal processo cliente per creare oggetti che rappresentano i datagrammi contenenti i dati da inviare (devono essere anche specificati l'indirizzo e la porta del destinatario) o da ricevere.

I costruttori della classe *DatagramSocket* più comunemente adoperati sono i seguenti:

public DatagramSocket () throws SocketException
public DatagramSocket (int** port **) throws SocketException
il primo crea un socket in grado di inviare e ricevere datagrammi e lo associa ad una porta casuale tra quelle libere sull'host locale, mentre il secondo crea un socket legato alla porta specificata come argomento (il secondo costruttore è generalmente impiegato nello sviluppo di un processo servitore, in modo tale che possa mettersi in ascolto su una porta di propria scelta).

I due metodi più importanti della classe *DatagramSocket* sono i seguenti:

public void receive (DatagramPacket** p1 **) throws IOException
blocca l'esecuzione fino a quando il *DatagramSocket* non riceve un datagramma (i dati ricevuti vengono inseriti nel datagramma *p1*); opportuni campi di *p1* consentono di conoscere la lunghezza dei dati ricevuti, l'indirizzo IP e la porta del mittente;

public void send (DatagramPacket** p2 **) throws IOException
invia il datagramma *p2* usando il *DatagramSocket* su cui il metodo viene invocato (*p2* include, oltre ai dati veri e propri, l'indirizzo IP e la porta del destinatario).

La chiusura di un *DatagramSocket* avviene mediante il metodo:

public void close ()

Prima di inviare o ricevere dati è necessario creare datagrammi, ossia oggetti della classe *DatagramPacket*. Il costruttore da usare per inviare datagrammi è il seguente:

public DatagramPacket (byte[]** buf **, int** len **, InetAddress** add **,
int** port **)
l'array *buf* contiene i dati da inviare, *len* il numero di byte da trasferire, *add* e *port* individuano indirizzo e porta a cui il datagramma deve essere inviato.

Quando invece si devono ricevere datagrammi si usa il seguente costruttore:

public DatagramPacket (byte[]** buf **, int** len **)
l'array *buf* è destinato a contenere i dati ricevuti, con un massimo di *len* byte.

Riportiamo adesso alcuni metodi della classe *DatagramPacket*:

public InetAddress getAddress ()
restituisce l'indirizzo IP della macchina a cui il datagramma verrà inviato o da cui è stato ricevuto;

public int getPort ()
restituisce il numero di porta dell'host remoto a cui il datagramma verrà inviato o da cui è stato ricevuto;

public byte[] getData ()
restituisce i dati contenuti nel datagramma;

public int getLength ()
restituisce la lunghezza del campo dati del datagramma da inviare o ricevuto.

Esistono inoltre degli analoghi metodi "*set*" (invece che "*get*") che consentono di impostare l'indirizzo IP, il campo dati, eccetera.

Come esempio, riportiamo una applicazione con un processo servitore e un processo cliente, dove il processo servitore (in esecuzione sull'host di indirizzo IP 131.114.9.226) crea un *DatagramSocket* legato alla porta 1234, crea un'istanza di *DatagramPacket* in cui memorizzare i dati ricevuti dal processo cliente, e quindi entra in un ciclo infinito. All'interno del ciclo il processo servitore i) si blocca in attesa della ricezione di un nuovo datagramma (privo di dati), ii) crea una stringa che rappresenta la data attuale e la memorizza in un array di byte, iii) recupera l'indirizzo e la porta relativi al processo cliente, iv) crea un nuovo *DatagramPacket* contenente l'array di byte che rappresenta la data e lo invia al processo cliente.

```
import java.net.*;
import java.io.*;
import java.util.*;
class ServizioD
{ void servi()
  { DatagramSocket ds = null;
    try
    { ds = new DatagramSocket(1234);
      byte[] bufferInvia;
      byte[] bufferRicevi = new byte[0];
      // il server non riceve dati
      DatagramPacket p = new DatagramPacket
        (bufferRicevi, bufferRicevi.length );
      while(true)
      { ds.receive(p);
```

```
          String st = new Date().toString();
          bufferInvia = st.getBytes();
          InetAddress ad = p.getAddress();
          int po = p.getPort();
          DatagramPacket pp = new DatagramPacket
             (bufferInvia, bufferInvia.length, ad, po);
          ds.send(pp);
        }
      }
      catch(IOException e)
      { Console.scriviStringa(e.getMessage()); }
    }
}

public class ServitoreDatagram
{ public static void main(String[] args)
    { ServizioD se = new ServizioD();
      se.servi();
    }
}
```

Il processo cliente, all'interno del metodo *interagisci()*, i) crea un *DatagramSocket* da usare per scambiare messaggi con il processo servitore, ii) crea un *DatagramPacket* con una parte dati vuota (l'array *bufferInvia* è vuoto) e lo invia al punto terminale remoto, iii) crea un *DatagramPacket* per la ricezione dei dati, quindi si blocca in attesa della risposta del processo servitore, iv) quando la risposta arriva, recupera i dati, li trasforma in stringa che stampa poi su video.

```
import java.net.*;
import java.io.*;
import java.util.*;
class RichiestaD
{ private String IPServer;
  RichiestaD(String s)
  { IPServer = s; }
  void interagisci()
  { try
    { DatagramSocket ds = new DatagramSocket();
      byte[] bufferInvia = new byte[0];
      byte[] bufferRicevi = new byte[32];
      InetAddress in =
```

```
                InetAddress.getByName(IPServer);
          DatagramPacket p = new DatagramPacket
             (bufferInvia, bufferInvia.length,
              in, 1234);
        // non invia dati
        ds.send(p);
        p = new DatagramPacket
           (bufferRicevi, bufferRicevi.length);
        ds.receive(p);
        String rx = new String(p.getData() );
        Console.scriviStringa("Data: " + rx);
        ds.close();
      }
      catch (UnknownHostException u)
      { Console.scriviStringa("Host sconosciuto"); }
      catch (IOException e)
      { Console.scriviStringa(e.getMessage()); }
   }
}

public class ClienteDatagram
{ public static void main(String[] args)
   { RichiestaD ri = new RichiestaD("131.114.9.226");
     ri.interagisci();
   }
}
```

1.5.1. Multicast

Alcuni tipi di applicazioni hanno la necessità di far fluire gli stessi dati da una sorgente ad un certo numero, anche elevato, di destinazioni. Esempi tipici sono le applicazioni di videoconferenza o, più in generale, di streaming audio/video. In questi casi è opportuno sfruttare un meccanismo, il *multicast*, che consente a un processo servitore di far recapitare lo stesso pacchetto a tutti i clienti appartenenti ad un determinato gruppo. In particolare, il processo servitore invia ogni singolo pacchetto una sola volta (e non tante volte quanti sono i clienti), e il meccanismo si preoccupa di far giungere il pacchetto a tutti i processi clienti che lo desiderano.

Un gruppo multicast è specificato da uno degli indirizzo IP di classe D, che vanno da 224.0.0.0 a 239.255.255.255, escludendo però quelli che

vanno da 224.0.0.0 a 224.0.0.255 in quanto riservati per scopi specifici. Per ulteriori informazioni vedi il sito:

http://www.iana.org/assignments/multicast-addresses

Quando viene inviato un messaggio ad un gruppo multicast, tutti gli iscritti al gruppo all'interno di un certo raggio ricevono il messaggio (la consegna non è garantita). Per quanto riguarda l'operazione di invio, non è necessario che il mittente faccia parte del gruppo multicast a cui si vogliono inviare i dati, ma è sufficiente che specifichi l'indirizzo del gruppo come destinatario.

La classe *MulticastSocket*, che estende la classe *DatagramSocket*, consente a un'applicazione di prendere parte a un gruppo multicast. La classe dispone del seguente costruttore:

public MulticastSocket (int** porta **) throws IOException

Esso crea un socket legato alla porta specificata. Per poter ricevere traffico multicast, tale socket deve essere aggiunto al gruppo di suo interesse, e quando invece non si vuole più ricevere traffico multicast esso deve essere rimosso dal gruppo. I seguenti metodi della classe *MulticastSocket* consentono rispettivamente di iscriversi ad un gruppo o di essere rimossi:

public void joinGroup (InetAddress** gruppo **) throws IOException
public void leaveGroup (InetAddress** gruppo **) throws IOException

L'invio e la ricezione di datagrammi avvengono con i metodi *send()* e *receive()* ereditati dalla superclasse. In aggiunta, l'invio di pacchetti destinati ad un gruppo multicast può essere limitato ai ricevitori all'interno di un certo raggio usando il metodo della classe *MulticastSocket*:

public void setTimeToLive (int** ttl **) throws IOException
i pacchetti inviati mediante tale socket possono attraversare al più *ttl* router prima di essere scartati.

Riportiamo una applicazione con un processo servitore che implementa una sorgente di traffico multicast, e tanti processi clienti che possono ricevere i pacchetti inviati.

La classe *ServizioM*, relativa al processo servitore, produce ogni cinque secondi un pacchetto UDP contenente la data e l'ora attuale, quindi lo

invia al gruppo multicast individuato dall'indirizzo 224.5.9.12, ai clienti in in ascolto sulla porta 1235. L'invio dei pacchetti interessa la sottorete locale, in quanto il *MulticastSocket* stabilisce che non possono attraversare alcun router.

```
import java.net.*;
import java.io.*;
import java.util.*;
class ServizioM
{  final int CINQUE_SEC = 5000;
   MulticastSocket ds;
   byte[] bufferInvia;
   InetAddress iaM;
   void servi()
   {  try
      {  ds = new MulticastSocket();
         ds.setTimeToLive(0);
         iaM = InetAddress.getByName("224.5.9.12");
         while(true)
         {  String st = new Date().toString();
            bufferInvia = st.getBytes();
            DatagramPacket pp =
              new DatagramPacket(bufferInvia,
                   bufferInvia.length, iaM, 1235);
            ds.send(pp);
            try
            {  Thread.sleep(CINQUE_SEC);  }
            catch (InterruptedException e) { }
         }
      }
      catch(IOException e)
      {  Console.scriviStringa(e.getMessage()); }
   }
}

public class Servitore
{  public static void main(String[] args) {
      ServizioM sm = new ServizioM();
      sm.servi();
   }
}
```

La classe *RiceviM*, relativa a un processo cliente, si iscrive al gruppo multicast creando un *MulticastSocket* legato alla porta 1235 e invocando il metodo *joinGroup()*. Quindi, entra in un ciclo in cui attende la ricezione di un nuovo pacchetto e ne stampa a video il contenuto. Dopo aver ricevuto tre messaggi, esce dal ciclo e abbandona il gruppo multicast.

```
import java.net.*;
import java.io.*;
import java.util.*;
class RiceviM
{  InetAddress in;
   MulticastSocket ms;
   int quanti = 3;
   byte[] bufferRicevi = new byte[32];
   void interagisci()
   {  try
      {  in = InetAddress.getByName("224.5.9.12");
         ms = new MulticastSocket(1235);
         ms.joinGroup(in);
         DatagramPacket p =new DatagramPacket
                (bufferRicevi, bufferRicevi.length);
         while (quanti > 0)
         {  quanti--; ms.receive(p);
            String rx = new String(p.getData());
            Console.scriviStringa("Data: " + rx);
         }
      }
      catch (IOException e)
      {  Console.scriviStringa(e.getMessage());}
      finally
      {  try
         {  ms.leaveGroup(in);   }
         catch (IOException ee) {}
         ms.close();
      }
   }
}
```

```
public class ClienteMulticast
{ public static void main(String[] args)
  { RiceviM rm = new RiceviM();
    rm.interagisci();
  }
}
```

2. Invocazione di metodi remoti

2.1. Oggetti remoti e interfacce

Il meccanismo di invocazione di metodi remoti (*RMI: Remote Method Invocation*) permette a un processo cliente di invocare i metodi di un oggetto gestito da un processo servitore (*metodi remoti* di un *oggetto remoto*). Questa proprietà consente di strutturare un'applicazione distribuita come una collezione di oggetti dislocati su host diversi ed interagenti attraverso l'invocazione di metodi remoti, con vantaggi di chiarezza rispetto al caso in cui la comunicazione avvenga con un protocollo di livello trasporto, quale TCP o UDP.

Un oggetto remoto è una istanza di una classe (*classe remota*) che implementa una *interfaccia remota*: quest'ultima specifica l'insieme dei metodi dell'oggetto che è possibile invocare, e deve essere dichiarata sia nel corpo del processo servitore che nel corpo del processo cliente, mentre la classe remota deve essere definita solo nel corpo del processo servitore (non confondere interfaccia o classe con oggetto). Una interfaccia remota deve estendere l'interfaccia di sistema *Remote* (package *java.rmi*), la quale non prevede alcun metodo e ha solo funzioni di marcatura. Tutti i metodi di una interfaccia remota possono lanciare eccezioni di tipo *RemoteException* (sottoclasse di *Exception*, package *java.rmi*), in quanto esiste una elevata probabilità di fallimento di un'invocazione di metodo remoto (per esempio a causa di un malfunzionamento dell'host che ospita l'oggetto remoto o della rete). Tale classe di eccezioni è di tipo *checked*: obbligando il programmatore a gestirla, si ottiene un codice più robusto.

Un esempio di interfaccia remota è il seguente:

```
import java.rmi.*
public interface InterfRemota extends Remote
{ public int somma(int a, int b)
                throws RemoteException;
  public void stampa(String s)
                throws RemoteException;
  public MiaClasse elabora(MiaClasse m)
                throws RemoteException, MiaEcc;
}
```

Come si deduce dall'esempio, i metodi remoti possono avere argomenti e risultato sia di un tipo primitivo che di un tipo classe. In quest'ultimo caso, la classe può essere sia di sistema che definita dal programmatore, come la classe *MiaClasse*. La definizione delle classi introdotte dal programmatore deve comparire sia dal lato del processo servitore che dal lato del processo cliente. Oltre alla classe di eccezione *RemoteException*, ai metodi remoti si possono associare anche altre classi di eccezione. Se durante l'esecuzione di un metodo remoto si verifica una situazione anomala, la corrispondente eccezione viene notificata al processo cliente.

2.1.1. Architettura di Java RMI

Nella realtà, i processi clienti invocano localmente i metodi di un oggetto surrogato, detto oggetto *stub* (appartenente a una classe *stub*) che implementa anch'esso l'interfaccia remota e che inoltra l'invocazione al processo che gestisce l'oggetto remoto. Sul lato dell'oggetto remoto esiste un componente analogo detto oggetto *skeleton*, che a partire dalla versione 1.2 del linguaggio non è più strettamente necessario (anche se può essere presente per motivi di compatibilità con le vecchie versioni).

L'architettura RMI può essere schematizzata come mostrato in Figura 2.1. Il livello *stub-skeleton* intercetta le chiamate effettuate da un processo cliente e le inoltra al processo che gestisce l'oggetto remoto. Il livello *riferimento remoto* interpreta la semantica del tipo di riferimento. Esistono due tipi di riferimento remoto: *unicast* (i riferimenti ad un oggetto remoto sono validi fintanto che è vivo il processo servitore relativo all'oggetto remoto) e *activatable* (i riferimenti rimangono validi aldilà della singola esecuzione del processo servitore che contiene l'oggetto remoto, e possono

quindi essere salvati su disco e ripristinati). Il livello trasporto è basato sul protocollo TCP/IP.

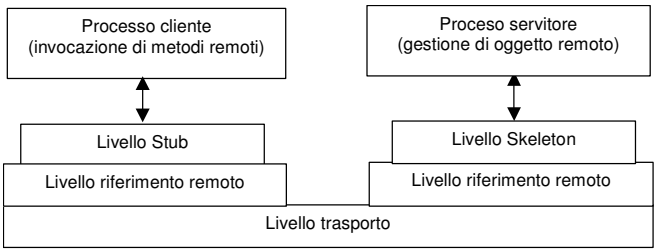


Figura 2.1. Architettura di Java RMI.

2.2. Il registro di RMI

Come detto nel sottoparagrafo 2.1.1, un processo cliente, per poter essere in grado di invocare i metodi di un oggetto remoto, deve possedere un oggetto stub (che contiene il *riferimento remoto*). Il meccanismo RMI fa uso di un registro (*rmiregistry*) che consente di eseguire le seguenti operazioni:

- da parte di un processo servitore, associare a un riferimento di un oggetto remoto (contenuto in un oggetto stub) un nome simbolico;
- da parte di un processo cliente, ottenere il riferimento dell'oggetto remoto (contenuto in un oggetto stub) associato a un dato nome simbolico.

Il registro viene attivato con il comando:

rmiregistry *porta*

L'argomento *porta* specifica la porta sulla quale il registro si mette in attesa: tale argomento è opzionale e, se non specificato, vale 1099. Il

registro deve essere attivato necessariamente sullo stesso host che ospita il processo servitore, ed è in grado di registrare uno o più oggetti remoti. Il registro è utile nella fase di avvio della applicazione distribuita, in quanto attraverso di esso i clienti possono procurarsi un riferimento degli oggetti remoti che intendono utilizzare.

Il processo servitore invia al registro RMI il riferimento di un oggetto remoto e specifica il nome simbolico a cui associarlo, utilizzando il seguente metodo statico della classe *Naming* (package *java.rmi*):

```
public static void rebind ( String nome , Remote obj )
        throws RemoteException , MalformedURLException
```

nome è un URL del tipo *//nomehost:porta/nomesimbolico* (*nomehost* è il nome dell'host locale, *porta* indica il numero di porta su cui è in ascolto il registro (va specificato se diverso da 1099), e *nomesimbolico* è la stringa con cui si vuole registrare il riferimento dell'oggetto), *obj* è il riferimento dell'oggetto stesso (la classe *MalformedURLException* fa parte del package *java.net*).

Un processo cliente chiede al registro RMI il riferimento di un oggetto remoto attraverso il nome simbolico ad esso associato, utilizzando il seguente metodo statico della classe *Naming* (la classe di eccezioni *NotBoundException,* sottoclasse di *Exception*, fa parte del package *java.rmi*):

```
public static Remote lookup ( String nome )
        throws NotBoundException , MalformedURLException,
                RemoteException
```

devono essere indicati con un'unica stringa (il formato è simile a quello visto per il metodo *rebind()*) l'host su cui è in esecuzione il registro, la porta (se diversa da 1099) e il nome simbolico associato al riferimento (come già indicato in precedenza la classe *MalformedURLException* fa parte del package *java.net*). Il riferimento restituito deve essere convertito al tipo dell'interfaccia remota.

Per esempio, se *ClasseRemota* implementa *InterfacciaRemota*, e il registro si trova sull'host del processo servitore *pc-frosini.iet.unipi.it*, si può scrivere:

Processo servitore:

```
ClasseRemota sr1 = new ClasseRemota();
Naming.rebind("//localhost/servizio", sr1);
```

oppure

```
InterfacciaRemota sr2 = new ClasseRemota();
Naming.rebind("//localhost/servizio", sr2);
```

Nel primo caso, il processo servitore può invocare tutti i metodi di *ClasseRemota*, sia remoti che non, attraverso il riferimento *sr1*; nel secondo caso, può invocare tramite *sr2* solo i metodi dell'interfaccia remota (all'interno del processo servitore le chiamate ai metodi dell'oggetto remoto sono sempre locali e avvengono secondo le normali regole del linguaggio). In entrambi i casi, il processo cliente può invocare solo i metodi dell'interfaccia remota.

Processo cliente:

```
InterfacciaRemota cr = (InterfacciaRemota)Naming.lookup
    ("//pc-frosini.iet.unipi.it/servizio");
```

Il registro RMI in realtà è implementato come un oggetto remoto (gestito da un apposito processo), dotato dei metodi remoti *rebind()* e *lookup()*. I trasferimenti da e verso il registro non riguardano solo un riferimento remoto ma un intero oggetto stub (che contiene un riferimento remoto). Più precisamente, il metodo *rebind()* costruisce un oggetto stub e ne trasferisce una copia dal processo servitore al registro, mentre il metodo *lookup()* trasferisce una copia di tale oggetto stub dal registro al processo cliente (l'oggetto stub, dovendo attraversare la rete, è serializzato). Il trasferimento di un oggetto stub richiede, naturalmente, la presenza di una classe stub da entrambi i lati della comunicazione (vedi paragrafo 2.4).

2.3. Realizzazione ed utilizzo di un oggetto remoto

Una classe remota, oltre a implementare un'interfaccia remota, estende

tipicamente la classe *UnicastRemoteObject* (appartenente al package *java.rmi.server*)*:* quest'ultima possiede il protocollo di comunicazione predefinito di RMI, e ridefinisce per l'ambiente distribuito i metodi *hashCode()*, *equals()* e *toString()* ereditati dalla classe *Object*.

Il costruttore della classe remota richiama comunemente *super()*, il costruttore default della superclasse *UnicastRemoteObject,* il quale provvede ad *esportare* l'oggetto creato (ovvero abilita l'oggetto a ricevere invocazioni remote dei suoi metodi): anche il costruttore, quindi, deve dichiarare di poter lanciare eccezioni di tipo *RemoteException*.

Le invocazioni provenienti da più processi clienti possono essere eseguite in parallelo (il sistema di supporto a tempo di esecuzione di RMI genera automaticamente dei thread aggiuntivi). Per questo motivo, il programmatore deve preoccuparsi che le porzioni di codice relative ai metodi remoti di una classe siano eseguite in mutua esclusione quando esistono rischi derivanti da una esecuzione concorrente di tali metodi.

A titolo di esempio riportiamo una semplice applicazione che realizza un servizio di calcolo. I metodi remoti che possono essere invocati sono dichiarati nell'interfaccia remota *CalcInterf* che estende *Remote*.

```
import java.rmi.*;
public interface CalcInterf extends Remote
{ public int add(int a, int b) throws RemoteException;
  public int sub(int a, int b) throws RemoteException;
  public int mul(int a, int b) throws RemoteException;
  public int div(int a, int b) throws RemoteException;
}
```

La classe remota *CalcImpl* implementa l'interfaccia *CalcInterf* ed estende la classe *UnicastRemoteObject*. Oltre a definire il corpo dei metodi *add()*, *sub()*, *mul()* e *div()*, la classe *CalcImpl* ha un costruttore che richiama il costruttore della superclasse:

```
import java.rmi.*;
import java.rmi.server.*;
public class CalcImpl extends UnicastRemoteObject
                      implements CalcInterf
{ public CalcImpl() throws RemoteException
  { super(); }
  public int add(int a, int b) throws RemoteException
  { return a + b; }
```

```
  public int sub(int a, int b) throws RemoteException
  { return a - b; }
  public int mul(int a, int b) throws RemoteException
  { return a * b; }
  public int div(int a, int b) throws RemoteException
  { return a / b; }
}
```

La classe *CalcServitore* crea un oggetto di tipo *CalcImpl* e iscrive il suo riferimento nel registro RMI con il nome "*calcserv*" (la classe remota e la classe che effettua il servizio (contenente il metodo *main()*) sono in genere distinte, ma niente vieta di usare una unica classe per entrambi gli scopi):

```
import java.rmi.*;
import java.net.*;
public class CalcServitore
{ public static void main(String args[])
  { try
    { CalcImpl cs = new CalcImpl();
      Naming.rebind("//localhost/calcserv", cs);
    }
    catch (RemoteException e)
    { Console.scriviStringa("Errore remoto: " + e);   }
    catch (MalformedURLException e)
    { Console.scriviStringa("Errore URL: " + e); }
  }
}
```

Un processo cliente si procura il riferimento remoto associato alla stringa "*calcserv*" attraverso il processo che gestisce il registro RMI, in esecuzione sull'host del processo servitore (supponiamo che abbia indirizzo IP 131.114.9.226). Notare che il riferimento remoto viene assegnato ad un riferimento di tipo interfaccia remota (*CalcInterf*) e non del tipo della classe che la implementa (*CalcImpl*).

```
import java.rmi.*;
import java.net.*;
public class CalcCliente
{ public static void main(String[] args)
  { try
    { CalcInterf cc = (CalcInterf) Naming.lookup
                      ("//131.114.9.226/calcserv");
```

```
      Console.scriviIntero(cc.sub(4, 3));
      Console.scriviIntero(cc.add(4, 5));
      Console.scriviIntero(cc.mul(3, 6));
      Console.scriviIntero(cc.div(9, 3));
    }
    catch (MalformedURLException murle)
    { Console.scriviStringa("Errore: " + murle); }
    catch (RemoteException re)
    { Console.scriviStringa("Errore: " + re);  }
    catch (NotBoundException nbe)
    { Console.scriviStringa("Errore: " + nbe); }
    catch (ArithmeticException ae)
    { Console.scriviStringa("Errore: " + ae);  }
  }
}
```

Nel caso in cui il processo cliente invochi il metodo *div()* passando *0* come divisore, il processo servitore solleva l'eccezione (*unchecked*) *ArithmeticException*, che viene passata al processo cliente.

2.4. Compilazione ed esecuzione di una applicazione RMI

Come detto nel paragrafo 2.2, un riferimento remoto, richiesto al registro *rmiregisty* da un processo cliente con il metodo *lookup()* della classe *Naming*, consiste in sostanza nell'oggetto surrogato stub, già serializzato, inserito nel registro stesso dal processo servitore con il metodo *rebind()* della classe *Naming*. Per poter quindi eseguire una applicazione distribuita, è necessario produrre la classe relativa a tale oggetto (e, per le versioni del linguaggio 1.2 e precedenti, anche la classe relativa all'oggetto skeleton): questo si ottiene con il comando di compilazione *rmic*, che richiede l'indicazione del nome della classe remota (*ClasseRemota*), e produce in uscita i file *ClasseRemota_Stub.class* e *ClasseRemota_Skel.class*.

Con riferimento alla classe remota *CalcImpl* del paragrafo precedente, la produzione dei due file oggetto *CalcImpl_Stub.class* e

CalcImpl_Skel.class avviene con il seguente comando (devono essere raggiungibili i file *CalcImpl.class* e *CalcInterf.class*):

rmic *CalcImpl*

Una applicazione basata su RMI si compone di almeno tre processi: quello relativo al registro RMI, quello cliente e quello servitore. Cerchiamo quindi di elencare quali siano le classi necessarie all'esecuzione dei tre processi, con riferimento all'esempio del paragrafo precedente (omettiamo di indicare la classe *Console*, ritenendo evidente quando si renda necessaria la sua presenza):

- *registro*: devono essere raggiungibili i file *CalcInterf.class* e *CalcImpl_Stub.class* (il registro deve memorizzare l'oggetto stub serializzato);
- *processo servitore*: oltre al file *CalcServitore.class,* devono essere raggiungibili i file *CalcInterf.class*, *CalcImpl.class*, *CalcImpl_Stub.class*, e per alcune versioni del linguaggio *CalcImpl_Skel.class* (il processo servitore invia al registro una copia dell'oggetto stub serializzato);
- *processo cliente:* oltre al file *CalcCliente.class* devono essere raggiungibili i file *CalcInterf.class* e *CalcImpl_Stub.class* (il processo cliente carica dal registro una copia dell'oggetto stub serializzato).

Nel caso più semplice, su un host è presente una cartella *servitore* e su un altro host (anche lo stesso) una cartella *cliente*. Nella cartella *servitore* devono essere inizialmente presenti i file sorgente *CalcServitore.java, CalcInterf.java* e *CalcImpl.java*: con il comando ***javac*** *CalcServitore.java* si ottengono i corrispondenti file oggetto, e con il comando ***rmic*** *CalcImpl* i file *CalcImpl_Stub.class* e *CalcImpl_Skel.class*. Nella cartella *cliente* devono essere inizialmente presenti i file sorgente *CalcCliente.java* e *CalcInterf.java*: con il comando ***javac*** *CalcCliente.java* si ottengono i corrispondenti file oggetto, e dalla cartella *servitore* deve essere ricopiato il file *CalcImpl_Stub.class*.

Per eseguire l'applicazione si deve operare come segue:

- nella cartella *servitore* eseguire il comando *rmiregistry*;

- nella cartella *servitore* eseguire il comando *java CalcServitore*;
- nella cartella *cliente* eseguire il comando *java CalcCliente*.

Lo scambio del riferimento (oggetto stub) dell'oggetto remoto *cs* avviene come mostrato in Fig. 2.2 (linee unite). Le linee tratteggiate rappresentano le invocazioni di metodi remoti.

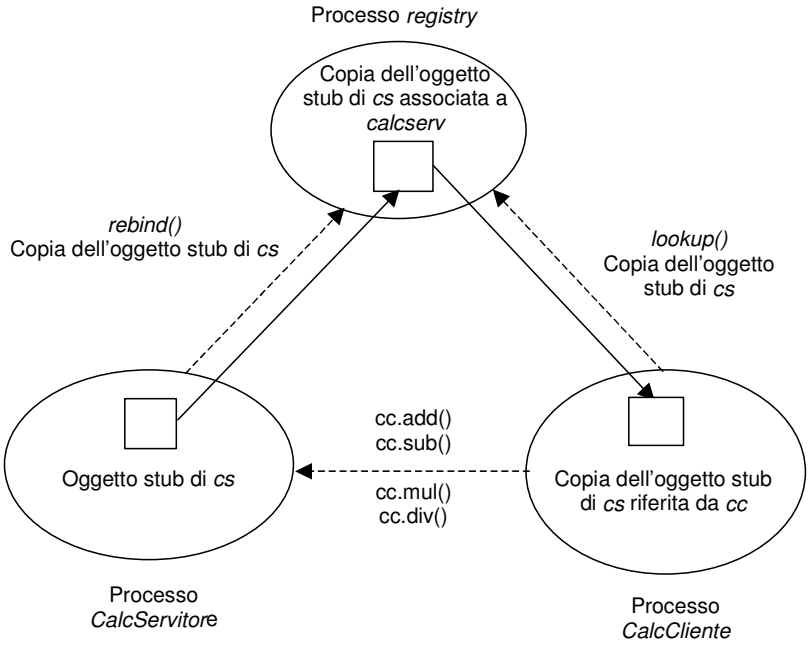


Figura 2.2. Scambio di riferimenti per l'oggetto remoto *cs*

2.5. Argomenti e risultato di metodi remoti

Nell'ambito dello stesso processo, l'invocazione di un metodo con

argomenti comporta il passaggio del loro valore, siano essi di un tipo primitivo o di un tipo classe (riferimento). Lo stesso dicasi per il risultato.

Nel caso di metodi remoti, argomenti e risultato subiscono un trattamento diverso (non avrebbe significato scambiare un riferimento per valore, dal momento che la validità del riferimento stesso è limitata allo spazio di indirizzamento in cui l'oggetto risiede). Le regole per ogni argomento e per il risultato sono le seguenti:

- tipo primitivo: lo scambio avviene *per valore*;
- tipo riferimento di un oggetto non remoto (oggetto che non implementa l'interfaccia *Remote*): i) la classe che specifica il tipo deve implementare l'interfaccia *Serializable*, ii) la definizione della classe deve essere presente sia dal lato servitore che dal lato cliente, iii) lo scambio avviene *per copia dell'oggetto riferito* (l'oggetto riferito viene prima serializzato e poi scambiato);
- tipo riferimento di un oggetto remoto: i) il tipo deve essere quello dell'interfaccia remota, dichiarata sia dal lato servitore che dal lato cliente, ii) lo scambio avviene *per riferimento remoto* (tale riferimento è contenuto in un oggetto stub che viene serializzato e scambiato).

Notare che un processo cliente può ottenere un riferimento remoto, oltre che dal registro RMI (tramite il metodo *lookup()*), anche come risultato di un metodo remoto di un oggetto remoto gestito da un servitore (secondo esempio di questo paragrafo). Inoltre, un processo servitore può ottenere un riferimento remoto (di un oggetto gestito dal cliente) come argomento di un proprio metodo di un proprio oggetto (per il cliente, metodo remoto di un oggetto remoto) (esempio del paragrafo 2.6).

Come primo esempio, riportiamo il codice di un'applicazione in cui i processi cliente e servitore comunicano scambiandosi oggetti di tipo classe. In particolare, gli oggetti scambiati sono istanze di una classe di sistema (*String*) e di una classe definita dal programmatore (*MiaClasse*).

L'interfaccia remota *ElabInterf* specifica il servizio esportato dal servitore ed è composta dai metodi *elabora1()* e *elabora2()*:

```
import java.rmi.*;
public interface ElabInterf extends Remote
```

```
{ public String elabora1(String s)
                  throws RemoteException;
  public MiaClasse elabora2(MiaClasse m)
                  throws RemoteException;
}
```

Il metodo *elabora1()* consente al processo servitore di ricevere una copia dell'oggetto stringa passato dal cliente come argomento attuale (*String* non implementa l'interfaccia *Remote*, pertanto il passaggio avviene per copia dell'oggetto). In maniera analoga, il valore di ritorno viene usato per trasmettere un dato di tipo stringa al cliente. La classe *String*, essendo una classe di sistema, è automaticamente definita per tutte le macchine virtuali. Inoltre, come molte delle classi di sistema, implementa l'interfaccia *Serializable*.

Tramite il metodo e*labora2()* il servitore riceve una copia dell'istanza di *MiaClasse* passata dal cliente come argomento attuale. In modo simile, il cliente riceve una copia di un oggetto di tipo *MiaClasse* locale al servitore come valore di ritorno. *MiaClasse* deve implementare l'interfaccia *Serializable*, in modo che sue istanze possano essere trasferite dal processo cliente al processo servitore e viceversa:

```
import java.io.*;
public class MiaClasse implements Serializable
{ private int i; private double d;
  public MiaClasse()
  { i = 0; d = 0; }
  public MiaClasse(int a, double b)
  { i = a; d = b; }
  public int giveInt()
  { return i; }
  public double giveDouble()
  { return d; }
  public void stampa()
  { Console.scriviStringa("Valore: " + i + "  " + d); }
}
```

Poiché sia il processo cliente che il processo servitore usano oggetti di tipo *MiaClasse*, il file *MiaClasse.class* deve essere raggiungibile da entrambi.

La classe *ElabImpl* implementa l'interfaccia *ElabInterf* e realizza il servizio di chiamata remota dei metodi *elabora1()* e *elabora2()*:

```
import java.rmi.*;
import java.rmi.server.*;
public class ElabImpl extends UnicastRemoteObject
                      implements ElabInterf
{ public ElabImpl() throws RemoteException
  { super(); }
  public String elabora1(String s)
                throws RemoteException
  { return "Viva " + s; }
  public MiaClasse elabora2(MiaClasse mo)
                 throws RemoteException
  { int ii = mo.giveInt() + 1;
    double dd =  mo.giveDouble() + 1.5;
    return (new MiaClasse(ii, dd));
  }
}
```

Il metodo *elabora1()* appende la stringa *s* ricevuta dal cliente alla stringa "*Viva* " e restituisce il risultato al chiamante. Il metodo *elabora2()* preleva il valore dei campi dati dell'oggetto *mo* ricevuto dal cliente, modifica tali valori e con questi crea una nuova istanza di *MiaClasse*, quindi la restituisce al cliente come valore di ritorno.

La classe *ElabServitore* crea un oggetto di tipo *ElabImpl* e iscrive il suo riferimento nel registro RMI con il nome "*servizio*":

```
import java.rmi.*;
import java.net.*;
public class ElabServitore
{ public static void main(String[] args)
  { try
    { ElabInterf el = new ElabImpl();
      Naming.rebind("//localhost/servizio", el);
    }
    catch (RemoteException re)
    { Console.scriviStringa("Errore remoto: " + re); }
    catch (MalformedURLException me)
    { Console.scriviStringa("Errore URL: " + me); }
  }
}
```

Il comportamento del processo cliente è definito dalla classe *ElabClient*:

```
import java.rmi.*;
import java.net.*;
public class ElabClient
{ public static void main(String[] args)
  { MiaClasse uno = new MiaClasse(1, 2);
    try
    { ElabInterf el = (ElabInterf) Naming.lookup
          ("//131.114.9.226/servizio");
      String s1 = el.elabora1("Java");
      Console.scriviStringa(s1);
      MiaClasse due = el.elabora2(uno);
      due.stampa();
    }
    catch (MalformedURLException me)
    { Console.scriviStringa("Errore URL: " + me); }
    catch (RemoteException re)
    { Console.scriviStringa("Errore remoto: " + re); }
    catch (NotBoundException nbe)
    { Console.scriviStringa("Err. di corrispondenza: "
                            + nbe); }
  }
}
```

Il cliente si procura, tramite il registro RMI, il riferimento dell'oggetto remoto associato al nome "*servizio*" (supponiamo che il servitore sia eseguito su un calcolatore che ha indirizzo IP 131.114.9.226). Quindi invoca il metodo remoto *elabora1()*, utilizzando come argomento attuale la stringa "*Java*", e stampa il risultato ottenuto. Infine, invoca il metodo *elabora2()* passando come argomento attuale un oggetto di tipo *MiaClasse*, ottiene come risultato una istanza della stessa classe e ne stampa il valore.

Per compilare l'applicazione, supponiamo che su un host sia presente una cartella *servitore* e su un altro host (anche lo stesso) una cartella *cliente*. Nella cartella *servitore* devono essere inizialmente presenti i file sorgente *ElabInterf.java*, *ElabImpl.java, ElabServitore.java* e *MiaClasse.java*: con il comando ***javac*** *ElabServitore.java* si ottengono i corrispondenti file oggetto, e con il comando ***rmic*** *ElabImpl* i file *ElabImpl_Stub.class* e *ElabImpl_Skel.class*. Nella cartella *cliente* devono

essere inizialmente presenti i file sorgente *ElabInterf.java, ElabCliente.java* e *MiaClasse.java*: con il comando ***javac*** *ElabCliente.java* si ottengono i corrispondenti file oggetto, e dalla cartella *servitore* deve essere ricopiato il file *ElabImpl_Stub.class*.

Per eseguire l'applicazione si deve operare come segue:

- nella cartella *servitore* eseguire il comando *rmiregistry*;
- nella cartella *servitore* eseguire il comando *java ElabServitore*;
- nella cartella *cliente* eseguire il comando *java ElabCliente*.

Come secondo esempio riportiamo una applicazione cliente/servitore, dove il processo servitore ha la funzione di gestore di due calcolatrici, che hanno la specifica riportata nel paragrafo 2.3:

```
import java.rmi.*;
public interface CalcInterf extends Remote
{ public int add(int a, int b) throws RemoteException;
  public int sub(int a, int b) throws RemoteException;
  public int mul(int a, int b) throws RemoteException;
  public int div(int a, int b) throws RemoteException;
}

import java.rmi.*;
import java.rmi.server.*;
public class CalcImpl extends UnicastRemoteObject
                      implements CalcInterf
{ public CalcImpl() throws RemoteException
  { super(); }
  public int add(int a, int b) throws RemoteException
  { return a + b; }
  public int sub(int a, int b) throws RemoteException
  { return a - b; }
  public int mul(int a, int b) throws RemoteException
  { return a * b; }
  public int div(int a, int b) throws RemoteException
  { return a / b; }
}
```

L'interfaccia remota *GestoreInterf* specifica i metodi che i clienti possono adoperare per ottenere e per rilasciare una risorsa:

```
import java.rmi.*;
public interface GestoreInterf extends Remote
{ public CalcInterf richiestaCalc()
             throws RemoteException,
                    CalcNonDisponibileExc;
  public void rilascioCalc(CalcInterf ci)
             throws RemoteException,
                    CalcSconosciutaExc;
}
```

Le classi di eccezione che i due metodi possono lanciare sono definite come segue:

```
public class CalcNonDisponibileExc extends Exception{}

public class CalcSconosciutaExc extends Exception{}
```

La classe *GestoreImpl* realizza il gestore di due risorse di tipo *CalcImpl*. Il metodo *richiestaCalc()* è usato dai clienti per acquisire una risorsa (lancia un'eccezione di classe *CalcNonDisponibileExc* se non ci sono risorse disponibili). Il metodo *rilascioCalc()* restituisce al gestore la risorsa il cui riferimento remoto viene passato come argomento attuale (lancia un'eccezione di classe *CalcSconosciutaExc* se il riferimento remoto non corrisponde a nessuna delle due risorse gestite):

```
import java.rmi.*;
import java.rmi.server.*;
public class GestoreImpl extends UnicastRemoteObject
                         implements GestoreInterf
{ private CalcInterf risorsa1;
  private CalcInterf risorsa2;
  private boolean libera1; private boolean libera2;
  public GestoreImpl() throws RemoteException
  { risorsa1 = new CalcImpl();
    risorsa2 = new CalcImpl();
    libera1 = true;
    libera2 = true;
  }
  public synchronized CalcInterf richiestaCalc()
       throws RemoteException, CalcNonDisponibileExc
  { if(libera1)
    { libera1 = false; return risorsa1; }
```

```
    if(libera2)
    { libera2 = false; return risorsa2; }
    throw new CalcNonDisponibileExc();
  }
  public synchronized void rilascioCalc(CalcInterf ci)
      throws RemoteException, CalcSconosciutaExc
  { if(ci.equals(risorsa1))
    { libera1 = true; return;  }
    if(ci.equals(risorsa2))
    { libera2 = true; return;  }
    throw new CalcSconosciutaExc();
  }
}
```

Poiché le invocazioni provenienti da più processi clienti possono essere eseguite in parallelo, è opportuno dichiarare sincronizzati i due metodi della classe *GestoreImpl* in modo tale da mantenere consistente lo stato delle variabili istanza (*risorsa1*, *risorsa2*, *libera1*, *libera2*).

Il processo servitore, con la classe *GestoreCalc*, gestisce l'oggetto remoto *gs* appartenente alla classe *GestoreImpl*, e registra il suo riferimento col nome *calc_gestore*:

```
import java.rmi.*;
import java.net.*;
public class GestoreCalc
{ public static void main(String args[])
  { try
    { GestoreInterf gs = new GestoreImpl();
      Naming.rebind("//localhost/calc_gestore", gs);
    }
    catch (RemoteException e)
    { Console.scriviStringa("Errore remoto: " + e);  }
    catch (MalformedURLException e)
    { Console.scriviStringa("Errore URL: " + e);}
  }
}
```

Un processo cliente utilizza la classe *ClienteCalc* la quale si procura un riferimento del gestore tramite il registro di RMI, quindi entra in un ciclo in cui tenta di acquisire un riferimento *c* ad una risorsa di tipo *CalcInterf*.

Una volta ottenuto il riferimento, esce dal ciclo, esegue alcune operazioni, quindi rilascia la risorsa.

```
import java.rmi.*;
import java.net.*;
public class ClienteCalc
{ public static void main(String[] args)
  { try
    { GestoreInterf gc = (GestoreInterf)
      Naming.lookup("//131.114.9.226)/calc_gestore");
      CalcInterf c = null;
      while(true)
      { try
        { c = gc.richiestaCalc();
          break;
        }
        catch (CalcNonDisponibileExc e)
        { Console.scriviStringa
            ("Risorse non disponibili, riprovare");
          Thread.sleep(3*1000);
        }
      }
      Console.scriviIntero(c.add(24, 15));
      Console.scriviIntero(c.mul(31, 2));
      gc.rilascioCalc(c);
    }
    catch (InterruptedException e)
    { Console.scriviStringa("Errore: " + e);}
    catch (CalcSconosciutaExc e)
    { Console.scriviStringa
      ("Rilascio di risorsa non riconosciuta");
    }
    catch (Exception e)
    { Console.scriviStringa("Errore: " + e);}
  }
}
```

Nella cartella *servitore* devono essere inizialmente presenti i seguenti file sorgente: *CalcInterf.java, CalcImpl.java, GestoreInterf.java, GestoreImpl.java, CalcNonDisponibileExc.java, CalcSconosciutaExc.java, GestoreCalc.java*. Con il comando ***javac*** *GestoreCalc.java* si ottengono tutti i corrispondenti file oggetto. Con il comando ***rmic*** *CalcImpl* si

ottengono i file *CalcImpl_Stub.class* e *CalcImpl_Skel.class*, e con il comando ***rmic*** *GestoreImpl* si ottengono i file *GestoreImpl_Stub.class* e *GestoreImpl_Skel.class*.

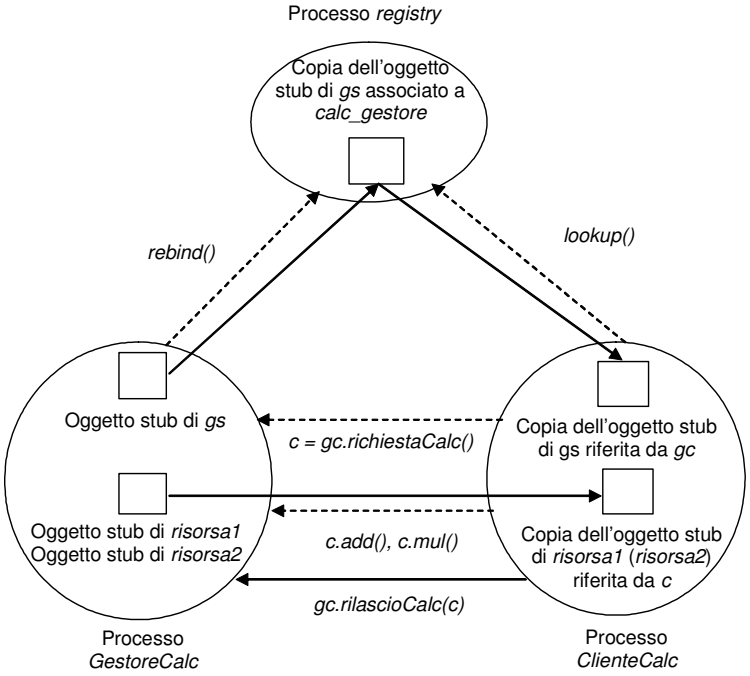


Figura 2.3. Scambio di riferimenti per i due oggetti remoti.

Nella cartella *cliente* devono essere inizialmente presenti i seguenti file sorgente: *CalcInterf.java*, *GestoreInterf.java*, *CalcSconosciutaExc.java*, *CalcNonDisponibileExc.java*, *ClienteCalc.java*. Con il comando ***javac*** *ClienteCalc.java* si ottengono tutti i corrispondenti file oggetto. Dalla cartella *servitore* devono essere poi ricopiati i file *CalcImpl_Stub.class* e *GestoreImpl_Stub.Class*.

I tre processi vengono attivati nel seguente modo:

- nella cartella *servitore* dare il comando *rmiregistry*;

- nella cartella *servitore* dare il comando *java GestoreCalc;*
- nella cartella *cliente* dare il comando *java ClienteCalc.*

Lo scambio di riferimenti di oggetti remoti (oggetti stub) avviene come mostrato in Fig. 2.3 (linee unite). Le linee tratteggiate rappresentano le l'invocazione di metodi remoti .

2.6. Callback

Nel caso in cui il processo servitore abbia necessità di comunicare con un processo cliente, è sufficiente che quest'ultimo crei un oggetto remoto e ne passi il riferimento al servitore. Successivamente, il servitore userà il riferimento per invocare i metodi dell'oggetto del cliente, per esempio al verificarsi di condizioni specifiche. Questo meccanismo è noto con il nome di *callback*.

Supponiamo di voler realizzare un servizio che consiste nell'invocare con un certa cadenza un metodo dei clienti registrati. L'interfaccia remota *ClienteBInterf* contiene la dichiarazione del metodo *battito()* che viene invocato dal servitore:

```
import java.rmi.*;
public interface ClienteBInterf extends Remote
{ public void battito() throws RemoteException;  }
```

L'interfaccia remota *ServizioBInterf* prevede il metodo *registra()* attraverso il quale i processi clienti possono registrarsi al servizio:

```
import java.rmi.*;
public interface ServizioBInterf extends Remote
{ public void registra
     (ClienteBInterf c, long intervallo)
                          throws RemoteException;
}
```

La classe *ServizioBImpl* realizza il servizio. Quando il metodo *registra()* viene invocato da un cliente, viene creata una nuova istanza di

GestoreCliente, inizializzata con il riferimento remoto del cliente e con l'intervallo con cui il metodo *battito()* deve essere invocato. Il costruttore della classe *GestoreCliente*, che estende *Thread*, genera un nuovo thread responsabile dell'invocazione periodica del metodo *battito()* del cliente.

```
import java.rmi.*;
import java.rmi.server.*;
public class ServizioBImpl
         extends UnicastRemoteObject
         implements ServizioBInterf
{ class GestoreCliente extends Thread
  { private ClienteBInterf cliente;
    private long intervallo;
    GestoreCliente(ClienteBInterf c, long i)
    { cliente = c; intervallo = i;
      start();
    }
    public void run()
    { try
      { while(true)
        { Thread.sleep(intervallo);
          cliente.battito();
        }
      }
      catch (RemoteException e)
      { Console.scriviStringa("Errore: " + e);}
      catch (InterruptedException e)
      { Console.scriviStringa("Errore: " + e);}
    }
  }
  ServizioBImpl() throws RemoteException {}
  public void registra(ClienteBInterf ci, long i)
  { new GestoreCliente(ci, i); }
}
```

La classe *ServizioB* crea un oggetto di tipo *ServizioBImpl* e registra il suo riferimento utilizzando il nome *cadenza*:

```
import java.rmi.*;
import java.net.*;
public class ServizioB
{ public static void main(String[] args)
```

```
  { try
    { ServizioBInterf sbi = new ServizioBImpl();
      Naming.rebind("//localhost/cadenza", sbi);
    }
    catch (MalformedURLException e)
    { Console.scriviStringa("URL errato: " + e); }
    catch (RemoteException e)
    { Console.scriviStringa("Errore remoto: " + e);  }
  }
}
```

La classe *ClienteBImpl* estende la classe *UnicastRemoteObject* e implementa l'interfaccia *ClienteBInterf*: il metodo *battito()* stampa la data e l'ora di invocazione del metodo stesso:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
public class ClienteBImpl extends UnicastRemoteObject
                implements ClienteBInterf
{ public ClienteBImpl() throws RemoteException
  {}
  public void battito() throws RemoteException
  { Console.scriviStringa("Data: " + new Date());  }
}
```

La classe *ClienteB* crea un istanza di *ClienteBImpl* e si procura tramite il registro (host 131.114.9.226) un riferimento al servizio corrispondente al nome *cadenza*. Quindi richiama il metodo *registra()* (previsto dall'interfaccia remota implementata dal servitore) specificando che desidera un intervallo di un secondo tra una invocazione di *battito()* e la successiva da parte del servitore:

```
import java.rmi.*;
import java.net.*;
public class ClienteB
{ public static void main(String[] args)
  { try
    { ClienteBInterf c = new ClienteBImpl();
      ServizioBInterf sb = (ServizioBInterf)
          Naming.lookup("//131.114.9.226/cadenza");
      sb.registra(c, 1000);
```

```
    }
    catch (RemoteException e)
    { Console.scriviStringa("Errore remoto: " + e);   }
    catch (MalformedURLException e)
    { Console.scriviStringa("URL errato: " + e); }
    catch (NotBoundException e)
    { Console.scriviStringa
          ("Servizio non registrato: " + e);
    }
  }
}
```

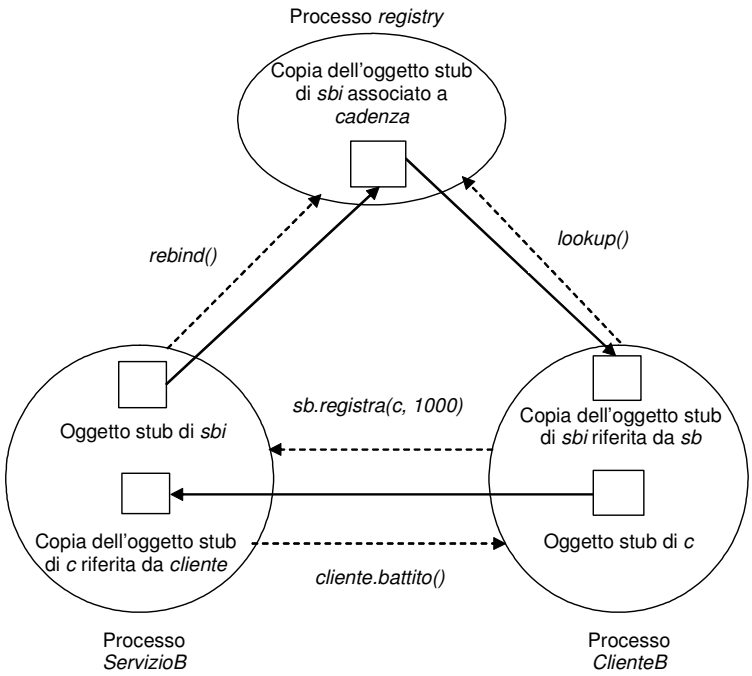


Figura 2.4. Scambio di riferimenti per i due oggetti remoti.

Nella cartella *servitore* devono essere inizialmente presenti i seguenti file sorgente: *ServizioBInterf.java, ServizioBImpl.java, ServizioB.java,*

ClienteBInterf.java. Con il comando ***javac*** *ServizioB.java* si ottengono tutti i corrispondenti file oggetto. Con il comando ***rmic*** *ServizioBImpl* si ottengono i file *ServizioBImpl_Stub.class* e *ServizioBImpl_Skel.class*. Dalla cartella cliente deve essere poi ricopiato il file *ClienteBImpl_Stub.class*.

Nella cartella *cliente* devono essere inizialmente presenti i seguenti file sorgente: *ClienteBInterf.java, ClienteBImpl.java, ClienteB.java, ServizioBInterf.java.* Con il comando ***javac*** *ClienteB.java* si ottengono tutti i corrispondenti file oggetto. Con il comando ***rmic*** *ClientBImpl* si ottengono i file *ClienteBImpl_Stub.class* e *ClienteBImpl_Skel.class*. Dalla cartella *servitore* devono essere poi ricopiato il file *ServitoreBImpl_Stub.class*.

I tre processi vengono attivati nel seguente modo:

- nella cartella *servitore* dare il comando *rmiregistry*;
- nella cartella *servitore* dare il comando *java ServizioB;*
- nella cartella *cliente* dare il comando *java ClienteB.*

Lo scambio di riferimenti di oggetti remoti (oggetti stub) avviene come mostrato in Fig. 2.4 (linee unite). Le linee tratteggiate rappresentano le l'invocazione di metodi remoti .

2.7. Mobilità del codice in RMI

Finora abbiamo supposto che le classi necessarie all'esecuzione del processo cliente e del processo servitore vengano distribuite manualmente sugli host coinvolti nell'esecuzione dell'applicazione distribuita (notare che il meccanismo RMI *non* invia classi ma solo oggetti serializzati). Esiste però una metodologia di caricamento dinamico dalla rete prevista dal meccanismo RMI, che consente di trasferire a tempo di esecuzione le classi necessarie ad una applicazione sfruttando il *class loader* di RMI.

Osservazione. Un class loader è un oggetto che si occupa del caricamento delle classi all'interno della Java Virtual Machine: normalmente il caricamento avviene dal disco, ma esistono anche class loader in grado di caricare le classi dalla rete. Il

supertipo di tutti i class loader è costituito dalla classe *ClassLoader* (package *java.lang*): definendo opportune sottoclassi, il programmatore può definire nuove politiche di caricamento.

Per comprendere questa metodologia, è necessario anzitutto introdurre il concetto di *codebase*: esso è un'URL da cui un processo può caricare le classi. Da un certo punto di vista, il classpath associato ad un processo rappresenta una sorta di codebase locale, in quanto indica l'insieme delle cartelle da cui il processo stesso può caricare le classi che gli servono.

Quando un processo viene mandato in esecuzione (comando *java*), può essere specificato un codebase con la seguente opzione:

-Djava.rmi.server.codebase = *codebase*

Supponiamo che un processo B invochi un metodo remoto di un oggetto gestito da un processo A, specificando come argomento il riferimento di un oggetto locale di B. Se il processo B è stato attivato con l'opzione codebase, l'oggetto locale viene serializzato e marcato con il valore codebase specificato nell'opzione stessa. Quando il processo A riceve l'oggetto, il class loader di RMI è in grado di conoscere, attraverso la marcatura, l'URL in cui si trova la classe relativa all'oggetto ricevuto (se non presente nel classpath locale), e quindi di farne richiesta. Il trasferimento della classe da tale URL avviene utilizzando un file server, tipicamente un server HTTP.

Per esempio, consideriamo il caso di un processo servitore, attivato con la specifica codebase, che gestisce un oggetto remoto appartenente a una classe *ClasseObj* che implementa l'interfacca remota *Interf*, e di un processo cliente che non disponga inizialmente della classe *ClasseObj_stub.class*. (Figura 2.5).

Supponiamo che il server HTTP che effettua il trasferimento delle classi venga attivato sull'host su cui risiede il processo servitore, in ascolto sulla porta 8080: tale server fornisce ai richiedenti le classi contenute nella cartella *radice-doc*, che contiene una copia dei file *ClasseObj_stub.class* e *Interf.class*.

Supponiamo inoltre che:

- le classi che implementano il servitore siano contenute nella cartella *servitore* (host A);

- le classi relative al cliente siano contenute in una cartella *cliente* (host B), nella quale non è però presente la classe *ClasseObj_stub.class*;
- il registro RMI sia stato lanciato da una cartella diversa da *radice-doc* e da *servitore* (host A);

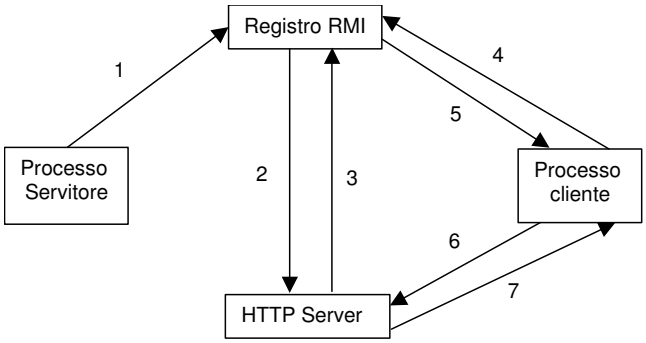


Figura 2.5. Caricamento dinamico dello stub.

A questo punto facciamo partire il processo servitore attivandolo con il comando (*ClasseServitore.class* è la classe principale):

java -Djava.rmi.server.codebase = *http://hostA:8080/ ClasseServitore*

Il sistema evolve secondo i seguenti passi:

1. il processo servitore, con il metodo *rebind()*, registra il riferimento remoto nel registro RMI in esecuzione sull'host A; il registro riceve lo stub serializzato dell'oggetto remoto, marcato con l'URL *http://hostA:8080/*;
2. il registro RMI cerca le classi *Interf.class* e *ClasseObj_stub.class* nel suo classpath locale: non trovandole le chiede al server HTTP indicato dal codebase con cui è stato marcato l'oggetto stub;
3. il server HTTP invia le classi richieste al registro RMI;
4. il processo cliente, una volta attivato, invoca il metodo *lookup()* sul registro RMI;

5. lo stub serializzato dell'oggetto remoto, marcato con l'URL *http://hostA:8080/*, viene trasferito al processo cliente;
6. il processo cliente cerca la classe *ClasseObj_stub.class* nel suo classpath locale: non trovandola la chiede al server HTTP indicato dal codebase con cui è stato marcato l'oggetto stub;
7. il server HTTP invia la classe richiesta al cliente, che a questo punto è pronto ad invocare i metodi dell'oggetto remoto.

Il registro RMI, come detto in precedenza, utilizza l'interfaccia remota e la classe stub. Se viene attivato con un percorso uguale a quello della cartella del servitore, il registro carica i file necessari dal classpath locale e non memorizza il codebase impostato dal servitore. Successivamente il cliente, una volta eseguito il metodo *lookup()*, non riuscirà a procurarsi la classe stub, in quanto il codebase impostato dal servitore non è stato registrato. Pertanto è opportuno attivare il registro al di fuori della cartella del servitore.

Per poter scaricare le classi, il programma cliente deve essere modificato includendo un *security manager* (il cliente scarica dalla rete del codice che potrebbe eseguire delle azioni illegali): il security manager effettua dei controlli sulle azioni eseguite dal codice, utilizzando un file di *policy* che specifica quali siano le azioni lecite e quali quelle proibite per il codice che proviene dall'esterno.

Per attivare un security manager è sufficiente includere nel programma cliente un frammento di codice del tipo:

```
if (System.getSecurityManager() == null)
  System.setSecurityManager(new RMISecurityManager());
```

Il file di policy da usare viene indicato mediante un comando del tipo:

java -Djava.security.policy = *nomefile*

Il file di policy è un semplice file di testo che può essere generato con l'ausilio dello strumento *policytool*, disponibile con il sistema di sviluppo Java. A titolo di esempio, riportiamo il contenuto di un file di policy che abilita il codice scaricato dalla rete ad eseguire una qualunque azione (e che è quindi sconsigliato usare al di fuori dell'ambito didattico):

```
grant
{ permission java.security.AllPermission;  };
```

Il meccanismo descritto è utile non solo per distribuire dinamicamente la classe stub e l'interfaccia, ma anche altre classi, estendendo al caso distribuito alcune proprietà che valgono all'interno di un singolo processo.

Con riferimento alle invocazioni di metodo locali, se un metodo ha un argomento formale di un tipo classe, in Java il chiamante ha la facoltà di passare come argomento attuale non solo un riferimento di quel tipo, ma anche di un suo sottotipo. Analogamente, se un argomento formale è di un tipo interfaccia, l'argomento attuale può essere un riferimento appartenente a una qualunque classe che implementi tale interfaccia.

Nel caso distribuito, si ottiene un comportamento analogo scaricando a tempo di esecuzione le classi relative agli argomenti attuali, avendo l'obbligo di distribuire staticamente le sole classi relative agli argomenti formali.

Prendiamo in considerazione un metodo remoto *m()* che restituisce un oggetto di tipo interfaccia *I*. Sfruttando il caricamento dinamico delle classi, un processo cliente può invocare il metodo *m()* ed ottenere come valore di ritorno una istanza di una classe *X* che implementa *I*. Tale processo è in grado di proseguire la sua esecuzione anche se il file *X.class* non è stato inizialmente incluso nell'insieme di classi relative al processo cliente: infatti la classe *X*, necessaria a tempo di esecuzione, può essere caricata dal class loader dall'insieme di classi relative al processo servitore.

Come primo esempio riportiamo il codice di un'applicazione cliente/servitore in cui il processo cliente carica dinamicamente dalla rete sia la classe dello stub che la classe degli oggetti restituiti come valori di ritorno nelle invocazioni di metodo remoto. L'interfaccia remota *GenScalInterf* definisce il servizio esportato dal servitore e si compone di un solo metodo che i clienti possono invocare per ottenere oggetti di tipo *Scalare*:

```
import java.rmi.*;
public interface GenScalInterf extends Remote
{ Scalare dammiScalare() throws RemoteException;
}
```

dove il tipo interfaccia *Scalare* è definito come segue:

```
public interface Scalare
{ double valore();
}
```

La classe *GenScalImpl* fornisce l'implementazione dell'unico metodo dell'interfaccia *GenScalInterf* restituendo un oggetto di tipo *ClasseScalare*:

```
import java.rmi.*;
import java.rmi.server.*;
public class GenScalImpl extends UnicastRemoteObject
                              implements GenScalInterf
{ public GenScalImpl() throws RemoteException
  { super(); }
  public Scalare dammiScalare()
  { return new ClasseScalare(1, 2); }
}
```

dove il tipo *ClasseScalare*, che implementa l'interfaccia *Scalare*, è così definito:

```
import java.io.*;
public class ClasseScalare implements Serializable,
Scalare
{ double x; double y;
  public ClasseScalare(){}
  public ClasseScalare(double x, double y)
  { this.x = x; this.y = y;  }
  public double valore()
  { return Math.sqrt(x*x+y*y); }
}
```

La classe *ClasseScalare* implementa anche l'interfaccia *Serializable* in quanto sue istanze devono essere trasferite dal processo servitore al processo cliente.

La classe *GenScalServitore* crea un oggetto di tipo *GenScalImpl* e registra il suo riferimento con il nome "*genscal*":

```
import java.rmi.*;
```

```
import java.net.*;
public class GenScalServitore
{ public static void main(String[] args)
  { try
    { GenScalInterf gs = new GenScalImpl();
      Naming.rebind("//localhost/genscal", gs);
    }
    catch (RemoteException re)
    { Console.scriviStringa("Errore remoto: " + re); }
    catch (MalformedURLException mue)
    { Console.scriviStringa("URL errato: " + mue); }
  }
}
```

Il comportamento del processo cliente è definito dalla classe *ScalCliente*:

```
import java.rmi.*;
import java.net.*;
public class ScalCliente
{ public static void main(String[] args)
  { try
    { if (System.getSecurityManager() == null)
        System.setSecurityManager(
          new RMISecurityManager());
      GenScalInterf g = (GenScalInterf)
        Naming.lookup("//131.114.9.226/genscal");
      Scalare s = g.dammiScalare();
      Console.scriviStringa("Il valore e' " +
                            s.valore());
    }
    catch(RemoteException re)
    { Console.scriviStringa("Errore remoto: " + re); }
    catch (MalformedURLException mue)
    { Console.scriviStringa("URL errato: " + mue); }
    catch (NotBoundException nbe)
    { Console.scriviStringa("Servizio non registrato:"
                            + nbe);
    }
  }
}
```

Il metodo *main()*, dopo aver attivato un security manager, si procura il riferimento remoto associato al nome "*genscal*", quindi invoca il metodo *dammiScalare()*. Infine stampa il valore dell'oggetto *Scalare* ricevuto come valore di ritorno. Quando l'applicazione viene eseguita, l'oggetto ricevuto come valore di ritorno del metodo *dammiScalare()* sarà in realtà una istanza di *ClasseScalare*.

Per compilare l'applicazione, nella cartella *servitore* devono essere inizialmente presenti i file *GenScalServitore.java*, *GenScalImpl.java*, *GenScalInterf.java*, *Scalare.java*, e *ClasseScalare.java*. Tramite il comando ***javac*** *GenScalServitore.java* vengono ottenuti tutti i file oggetto. Quindi, mediante il comando ***rmic*** *GenScalImpl* vengono generati i file *GenScalImpl_Stub.class* e *GenScalImpl_Skel.class*.

Nella cartella *cliente* devono essere inizialmente presenti i file *ScalCliente.java*, *Scalare.java*, *GenScalInterf.java*. Tramite il comando ***javac*** *ScalCliente.java* vengono prodotti i corrispondenti file oggetto. I file *GenScalImpl_Stub.class* e *ClasseScalare.class* vengono scaricati dall'URL specificato tramite l'opzione *codebase* del processo servitore. Supponiamo inoltre che la cartella *cliente* contenga un file di policy (*policy.txt*) che abilita il codice scaricato dalla rete ad eseguire una qualunque azione.

Per trasferire le classi usiamo un file server minimale, ossia un server HTTP in grado di trasferire solo file *.class*, disponibile all'URL:

http://java.sun.com/products/jdk/rmi/class-server.zip

Per lanciare tale file server è necessario specificare la *porta* su cui si deve mettere in ascolto e la *cartella* che contiene i file da distribuire in copia agli utilizzatori:

java examples.classServer.ClassFileServer *porta cartella*

Supponiamo che *class-server.zip* sia stato decompresso nella cartella *servitore*, sull'host di indirizzo IP 131.114.9.226. I quattro processi vengono attivati nel seguente modo:

- in una cartella diversa da *servitore*, dare il comando *rmiregistry*;
- nella cartella *servitore*, dare il comando *java examples.classServer.ClassFileServer 2001 ./*;

- nella cartella *servitore*, dare il comando
  java –Djava.rmi.server.codebase=http://131.114.9.226:2001/ GenScalServitore;
- nella cartella *cliente*, dare il comando
  java –Djava.security.policy=policy.txt ScalCliente.

Il caricamento dinamico delle classi avviene quando:

- *GenScalServitore* invoca il metodo *rebind()* sul registro, che non trova i file *GenScalInterf.class* e *GenScalImpl_Stub.class* nel proprio classpath e li scarica utilizzando il file server;
- *ScalCliente* invoca il metodo *lookup()* e ottiene come valore di ritorno l'oggetto stub; non trovando la classe *GenScalImpl_Stub.class* nel classpath locale la scarica utilizzando il file server;
- *ScalCliente* invoca *dammiScalare()* che restituisce una istanza di *ClasseScalare*; il processo cliente non trova nel classpath locale la definizione di *ClasseScalare* e pertanto la scarica utilizzando il file server.

Il caricamento dinamico può avvenire anche in senso opposto: alcune classi non disponibili localmente al processo servitore possono essere caricate da un file server associato al cliente. In questo caso il processo cliente deve essere mandato in esecuzione specificando l'opzione codebase, mentre il processo servitore deve attivare un security manager ed avere un opportuno file di policy.

Come esempio conclusivo, riferiamoci a un sistema cliente/servitore fatto nel seguente modo: il cliente scarica dinamicamente lo stub dell'oggetto remoto con cui vuole interagire, mentre il servitore scarica dinamicamente la classe relativa all'argomento attuale su cui deve eseguire l'elaborazione. Il servitore utilizza l'interfacca *ElabInterf* che prevede il metodo *double elabora(MiaInterf oi)*, dove *MiaInterf* è una interfaccia in cui è dichiarato il metodo: *double giveDouble()*. La classe *ElabImpl* implementa il metodo *elabora()* di *ElabInterf* invocando sull'argomento *oi* il metodo *giveDouble()*, sommando *1.5* al valore ottenuto e restituendo il risultato al chiamante.

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

```
public interface MiaInterf
{ public double giveDouble();  }

import java.rmi.*;
public interface ElabInterf extends Remote
{ public double elabora(MiaInterf oi)
                  throws RemoteException;
}

import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public class ElabImpl extends UnicastRemoteObject
                      implements ElabInterf
{ public ElabImpl() throws RemoteException
  {}
  public double elabora(MiaInterf oi)
                   throws RemoteException
  { double dd = oi.giveDouble() + 1.5;
    return dd;
  }
}
```

La classe *ElabServer* costituisce l'implementazione del servitore:

```
public class ElabServer
{ public static void main(String[] args)
{ if (System.getSecurityManager() == null)
    System.setSecurityManager
           (new RMISecurityManager());
    try
    { ElabInterf el = new ElabImpl();
      Naming.rebind("//localhost/ServizioElab", el);
    }
    catch (RemoteException re)
    { Console.scriviStringa("Eccezione: " + re); }
    catch (MalformedURLException me)
    { Console.scriviStringa("Eccezione: " + me); }
  }
}
```

Il cliente, implementato dalla classe *ElabClient*, invoca il metodo *elabora()* del servitore passando come argomento attuale una istanza di *MiaClasseImpl*, che implementa l'interfaccia *MiaInterf* (il tipo dell'argomento formale del metodo *elabora()*). La classe *MiaClasseImpl* implementa il metodo *giveDouble()* semplicemente restituendo il valore passato in fase di costruzione dell'oggetto (o zero nel caso in cui venga usato il costruttore senza argomenti).

```
import java.io.*;
public class MiaClasseImpl implements MiaInterf,
                                      Serializable
{ private double d;
  public MiaClasseImpl() { d = 0;}
  public MiaClasseImpl(double b) {  d = b;}
  public double giveDouble()
  { return d;   }
}

import java.rmi.*;
import java.net.*;
public class ElabClient
{ public static void main(String[] args)
  { double d;
    MiaClasseImpl uno = new MiaClasseImpl(2);
    if (System.getSecurityManager() == null)
      System.setSecurityManager(
          new RMISecurityManager());
    try
    { ElabInterf el = (ElabInterf)
        Naming.lookup("//131.114.9.226/ServizioElab");
      d = el.elabora(uno);
      Console.scriviStringa("Il risultato e': " + d);
    } catch (MalformedURLException me) {
      Console.scriviStringa("Errore URL: " + me);
    } catch (RemoteException re) {
      Console.scriviStringa("Errore Remoto: " + re);
    } catch (NotBoundException ne) {
      Console.scriviStringa("Non registrato: " + ne);
    }
  }
}
```

Supponiamo che il processo servitore venga eseguito sull'host di indirizzo IP 131.114.9.226, e che nella cartella *servitore* siano presenti:

- i file *MiaInterf.class*, *ElabInterf.class*, *ElabImpl.class, ElabServer.class, ElabImpl_Stub.class;*
- il file *policy.txt* e il package del server HTTP.

Inoltre, supponiamo che il processo cliente sia eseguito sull'host di indirizzo IP 131.114.9.50, e che nella cartella *cliente* siano presenti:

- i file *MiaInterf.class*, *MiaClasseImpl.class, ElabInterf.class*, *ElabClient.class;*
- il file *policy.txt* e il package del server HTTP.

Per mandare in esecuzione il processo servitore ed il server HTTP che distribuisce le sue classi:

- da una cartella diversa da *servitore*, si attiva il registro RMI con il comando *rmiregistry*;
- dalla cartella *servitore*, si attiva il server HTTP con il comando:
  java examples.classServer.ClassFileServer 2001 ./
- dalla cartella *servitore*, si emette il comando:
  java –Djava.rmi.server.codebase=http://131.114.9.226:2001/ –Djava.security.policy=policy.txt ElabServer

Per mandare in esecuzione il processo cliente ed il server HTTP che distribuisce le sue classi:

- dalla cartella *cliente*, si attiva il server HTTP con il comando:
  java examples.classServer.ClassFileServer 2002 ./
- dalla cartella *cliente* si emette il comando:
  java –Djava.rmi.server.codebase=http://131.114.9.50:2002/ -Djava.security.policy=policy.txt ElabClient

Il caricamento dinamico delle classi ha luogo quando:

- *ElabServer* esegue il metodo *rebind()* sul registro: il registro carica da *http://*131.114.9.226*:2001/* i file *ElabImpl_Stub.class*, *ElabInterf.class* e *MiaInterf.class*;

- *ElabClient* esegue il metodo *lookup()* sul registro: *ElabClient* carica da *http://*131.114.9.226*:2001/* il file *ElabImpl_Stub.class*;
- *ElabClient* esegue il metodo *elabora()* passando come argomento attuale una istanza di *MiaClasseImpl*: *ElabServer* carica da *http://131.114.9.50:2002/* il file *MiaClasseImpl.class*.

2.8. Garbage collection distribuita

Il meccanismo RMI prevede un sistema di garbage collection distribuita che elimina un oggetto remoto quando sono vere entrambe le seguenti condizioni:

- non esistono più riferimenti locali dell'oggetto remoto;
- non ci sono più clienti che hanno un riferimento remoto relativo all'oggetto.

Quindi, anche se un oggetto remoto non è più riferito tramite riferimenti locali, l'oggetto remoto viene eventualmente tenuto in vita dai riferimenti remoti posseduti dai clienti (compreso il registro RMI).

Questa caratteristica è stata implicitamente adoperata in tutti gli esempi mostrati, in quanto il servitore dopo aver creato e registrato l'oggetto remoto termina l'esecuzione del metodo *main()*. Questo però non causa la terminazione del processo servitore, e non termina quindi la vita dell'oggetto remoto, grazie al riferimento posseduto dal registro.

Un oggetto remoto che desideri interagire con il processo di garbage collection distribuita può implementare l'interfaccia *Unreferenced* (package *java.rmi.server*):

```
public interface Unreferenced
{ public void unreferenced();
}
```

Il metodo *unreferenced()* viene invocato quando il numero dei riferimenti remoti validi diviene pari a zero.

3. Interfacce grafiche

3.1. Le librerie AWT e Swing

Il linguaggio Java include sin dalla prima versione una libreria per lo sviluppo di interfacce grafiche, detta AWT (*Abstract Windowing Toolkit*). Tale libreria è però caratterizzata da scarsa efficacia e limitata flessibilità, e i miglioramenti apportati nelle successive versioni conservano comunque alcuni dei difetti di origine. Per tale motivo è stata aggiunta una nuova libreria, nota con il nome di *Swing*.

In questo capitolo affronteremo lo studio delle interfacce grafiche adottando la libreria Swing, rimandando il lettore a testi più specifici nel caso in cui sia interessato anche alla libreria AWT. È comunque importante notare che molti dei componenti della libreria Swing hanno un analogo componente nella libreria AWT, rendendo non troppo complicato per il programmatore il passaggio dall'una all'altra (in molti casi il nome di una classe della libreria Swing può essere ottenuto dal nome della classe corrispondente della libreria AWT aggiungendo il prefisso *J*).

Per fare uso della libreria Swing è necessario importare il package *javax.swing*, che contiene la definizione delle classi relative a finestre, bottoni, menu, eccetera. Spesso è necessario utilizzare anche la libreria AWT, importando il package *java.awt*, in quanto alcune classi, tra cui quelle relative a colori e fonti, non sono state duplicate nella libreria Swing.

Oltre agli elementi grafici veri e propri, le librerie Swing e AWT mettono a disposizione del programmatore classi e interfacce che consentono di gestire l'interazione tra utente e programma mediante un

modello a *eventi* (un evento rappresenta un'azione compiuta dall'utente attraverso un'interfaccia grafica e che il programma deve gestire, quale per esempio la pressione di un bottone, la selezione di una scelta, la pressione di un tasto sulla tastiera, e così via).

3.2. Colori e fonti

La classe *Color* (package *java.awt*) rappresenta colori ottenuti da quelli fondamentali *red*, *green* e *blue*, con una data opacità.

Nella classe sono presenti i seguenti costruttori:

public Color (int** red **, int** green **, int** blue **)
crea un colore completamente opaco, con tre interi nel rango 0-255;

public Color (int** red **, int** green **, int** blue **, int** alfa **)
crea un colore, con 4 interi nel rango 0-255; il valore di *alfa* determina l'opacità del colore (0 completamente trasparente, 255 completamente opaco).

public Color (float** red **, float** green **, float** blue **)
crea un colore completamente opaco, con 3 reali nel rango 0.0-1.0.

public Color (float** red **, float** green **, float** blue **, float** alfa **)
crea un colore, con 4 reali nel rango 0.0-1.0; il valore di *alfa* determina l'opacità del colore (0.0f completamente trasparente, 1.0f completamente opaco).

Per esempio, si può scrivere:

```
Color co = new Color(200, 200, 200, 200);
```

Esiste un insieme di costanti definite nella classe *Color* (campi dato statici), utilizzate per rappresentare i colori più comuni (completamente opachi). Di seguito, per ognuna di tali costanti, sono riportati i valori delle componenti intere *red*, *green*, *blue* e *alfa*):

black	0, 0, 0, 255
blue	0, 0, 255, 255
cyan	0, 255, 255, 255
darkGray	64, 64, 64, 255
gray	128, 128, 128, 255
green	0, 255, 0, 255
lightGray	192, 192, 192, 255
magenta	255, 0, 255, 255
orange	255, 200, 0, 255
pink	255, 175, 175, 255
red	255, 0, 0, 255
white	255, 255, 255, 255
yellow	255, 255, 0, 255

La classe *Font* (package *java.awt*) rappresenta fonti, costituite da un nome, uno stile e una dimensione. Nella classe è definito il seguente costruttore:

public Font (String *nomeFonte* ***, int*** *stile* ***, int*** *dimensione* ***)***
crea una fonte, dove *nomeFonte* indica il nome della fonte come una stringa (esempi: "*Arial*", "*Times New Roman*", eccetera); *stile* può assumere come valore una delle costanti (definite nella classe *Font) PLAIN*, *BOLD*, *ITALIC* o l'OR delle ultime due; *dimensione* determina il corpo del carattere.

Per esempio, si può scrivere:

```
Font fo = new Font("Arial", Font.ITALIC|Font.BOLD, 24);
```

3.3. Contenitori e componenti

Una qualunque interfaccia grafica, per quanto semplice, è costituita da almeno un contenitore (oggetto della classe *Container*), al quale si possono aggiungere componenti (oggetti della classe *JComponent*). Alcuni componenti possono fungere da contenitori (per esempio *JPanel*) per cui possono essere contenuti all'interno di altri contenitori e possono

contenere altri componenti. Contenitori e componenti costituiscono gli elementi grafici (Fig. 3.1).

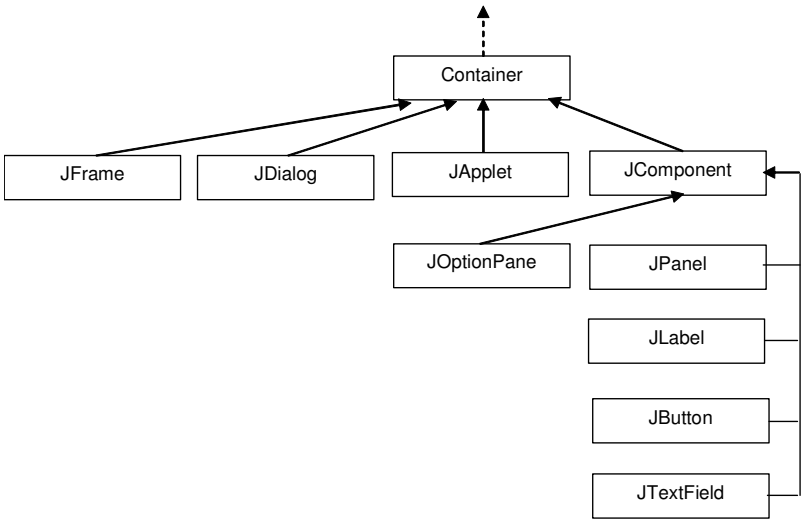


Figura 3.1. Contenitori e componenti.

Gli elementi grafici possono far uso di colori e fonti: per esempio, si possono usare i metodi (validi per tutte le classi coinvolte nella costruzione di interfacce grafiche) ***void setBackground(Color c)***, ***void setForeground(Color c)***, e ***void setFont(Font f)***: i primi due servono a cambiare il colore usato per lo sfondo (*background*) e per gli elementi in primo piano (*foreground*) rispettivamente, mentre l'ultimo cambia la fonte associata al componente.

Il modo con cui i componenti sono disposti in un contenitore viene genere stabilito attraverso un *gestore di layout*. Esso è un oggetto che stabilisce e modifica la dimensione, la forma, e il posizionamento dei componenti in funzione delle dimensioni del contenitore stesso. Sono previsti diversi gestori di layout, ognuno caratterizzato da una propria strategia nella disposizione dei componenti. Ogni contenitore ha un suo gestore di layout predefinito, che può tuttavia essere cambiato dal programmatore.

La classe *Container* (a cui appartengono tutti i contenitori) prevede il seguente metodo:

public void setLayout (LayoutManager** mgr **)
imposta un nuovo gestore di layout al posto di quello predefinito per quel contenitore.

Il gestore di layout più semplice è il flow layout (oggetto appartenente alla classe *FlowLayout*, che suddivide il contenitore in zone contigue la cui dimensione è la minima richiesta dal componente inserito in quella zona).

Per aggiungere un componente a un contenitore (con un gestore di layout di tipo flow layout), si usa il metodo della classe *Container* ***void add(Component** c)*.

3.3.1. Contenitori di alto livello

Alcuni contenitori sono detti *contenitori di alto livello* (o *contenitori principali*), e ogni interfaccia grafica deve averne almeno uno che agisca da elemento grafico più esterno.

Nella libreria Swing, i contenitori di alto livello appartengono a tre tipologie:

- *Frame*, per implementare una finestra principale;
- *Finestra di dialogo*, per implementare una finestra utile per interagire con l'utente in maniera semplice;
- *Applet*, per implementare un'area di visualizzazione all'interno della finestra di un browser.

Per poter descrivere in dettaglio le proprietà di tali contenitori, occorre dare qualche cenno ad alcuni componenti di base che vengono allo scopo utilizzati.

3.3.2. Componenti di base

Introduciamo in questo sottoparagrafo alcuni componenti di base in forma semplificata, che verranno utilizzati negli esempi dei paragrafi seguenti.

Etichette

Una etichetta è costituita da una stringa non selezionabile e non modificabile dall'utente. Essa è un oggetto della classe *JLabel*, e la stringa viene specificata come argomento del costruttore, che ha la forma ***JLabel(String s)***. Per esempio, si può scrivere:

```
JLabel la = new JLabel("Ciao");
```

Bottoni

Un bottone è costituito da un pulsante con etichetta (nome), che può essere premuto facendovi click col mouse. Esso è un oggetto della classe *JButton*, e il nome viene specificato come argomento del costruttore, che ha la forma ***JButton(String s)***. Per esempio, si può scrivere:

```
JButton bu = new JButton("OK");
```

Casella di testo

Una casella di testo è un'area in cui può essere scritto del testo. Essa è un oggetto della classe *JTextField*, i cui costruttori (***JTextField()***, ***JTextField(String s)***, ***JTextField(int n)***, ***JTextField(Strig s, int n)***) prevedono la possibilità di specificare sia il testo iniziale che compare nella casella sia il numero di colonne di cui è composta la casella stessa. Il testo contenuto nella casella viene restituito dal metodo ***String getText()***. Per esempio, si può scrivere:

```
JTextField tf = new JTextField("Ini", 10);
```

3.4. Contenitori principali

3.4.1 Frame

La classe *JFrame* fornisce l'implementazione di un *frame*, un contenitore principale che realizza una finestra bidimensionale dotata di una cornice, una barra in alto contenente un eventuale titolo, tre pulsanti (tipici di ogni finestra), e un pannello (corpo della finestra).

I costruttori della classe ***JFrame()*** e ***JFrame(String s)*** generano una finestra, inizialmente invisibile, con un eventuale titolo *s*. La finestra viene quindi resa visibile col metodo ***void setVisible(true)***. La dimensione iniziale della finestra può essere stabilita con il metodo ***void setSize(int x, int y)*** (*x* e *y* sono espressi in *pixel*). Generalmente la finestra può essere ridimensionata posizionando il mouse ai margini della stessa (per rendere la finestra insensibile alle operazioni di ridimensionamento si può usare il metodo ***void setResizable(boolean b)***, con argomento attuale uguale a *false*).

Esiste un thread di sistema (chiamiamolo thread *swing*) responsabile della fase di disegno delle finestre sullo schermo e della gestione degli eventi: questo diviene automaticamente attivo quando viene eseguito il metodo *setVisible()* della classe *JFrame*. Il thread *swing* rimane attivo anche quando il thread *main* termina: occorre pertanto che la sua fine sia comandata esplicitamente quando si vuole far terminare l'applicazione.

L'azione che viene effettuata quando si chiude la finestra può essere stabilita col metodo ***void setDefaultCloseOperation(int op)***, usando come argomento attuale una delle seguenti costanti (campi dato statici della classe *JFrame*):

DO_NOTHING_ON_CLOSE
la pressione del bottone di chiusura della finestra non produce alcun effetto;

HIDE_ON_CLOSE
la finestra viene nascosta ma il thread *swing* rimane attivo, pertanto l'applicazione non termina (è l'azione di default);

DISPOSE_ON_CLOSE
la finestra viene nascosta e se non ci sono altre finestre il thread *swing* termina la propria esecuzione (in quest'ultimo caso la Java Virtual Machine rimane in vita solo se ci sono altri thread ancora in esecuzione);

EXIT_ON_CLOSE
produce, con la chiusura della finestra, la terminazione del programma.

Il metodo ***Container getContentPane()*** restituisce un riferimento al *pannello* della finestra (che costituisce il suo contenitore vero e proprio). Nel pannello si possono fare disegni, e al pannello si possono aggiungere altri elementi grafici, quali etichette, bottoni, caselle di testo, eccetera.

Il seguente programma crea una semplice finestra principale (Fig. 3.2), all'interno della quale vengono inseriti l'etichetta "Ciao", il bottone "OK" e l'area di testo contenente la stringa "abc". Eseguendo l'applicazione si può notare che il processo non termina quando viene completato il metodo *main()*, in quanto il thread *swing* rimane in vita: esso viene fatto terminare chiudendo la finestra.

Figura 3.2. Una finestra principale dotata di tre componenti, come prodotta dal programma e ridimensionata.

```
import javax.swing.*; import java.awt.*;
public class Finestra
{ public static void main(String[] args)
  { Font f = new Font("Arial", Font.ITALIC, 30);
    JFrame finestra = new JFrame("Titolo");
    finestra.setDefaultCloseOperation
         (JFrame.DISPOSE_ON_CLOSE);
    Container c = finestra.getContentPane();
    c.setLayout(new FlowLayout());
    JLabel la = new JLabel("Ciao");
    la.setForeground(Color.red);
    la.setFont(f);c.add(la);
    JButton bu = new JButton("OK");
    bu.setBackground(Color.yellow);
    bu.setForeground(Color.red);c.add(bu);
    JTextField te = new JTextField("abc", 5);
    te.setBackground(Color.orange);
    te.setForeground(Color.blue); te.setFont(f);
    c.add(te);
    finestra.setSize(300, 100);
    finestra.setVisible(true);
  }
}
```

3.4.2. Finestre di dialogo

Il package *swing*, supporta l'utilizzo di *finestre di dialogo*, istanze della classe *JDialog*, che spesso vengono utilizzate in connessione con una finestra principale.

Finestre di dialogo opportunamente configurate possono essere anche ottenute utilizzando metodi statici della classe *JOptionPane*: tali finestre consentono di realizzare forme elementari di ingresso/uscita, indicando all'utente che deve inserire un dato o fornendo all'utente un messaggio di risposta. L'utente deve dare conferma dell'avvenuto inserimento o di presa visione del messagio ricevuto premendo o il bottone "OK" o il tasto "Enter". Le finestre di dialogo così ottenute non effettuano alcuna azione quando vengono chiuse.

Il primo metodo statico della classe *JOptionPane* è il seguente:

static void showMessageDialog (Component** com **, Object** mesg **,
String** titolo **, int** tipo **)

visualizza una finestra che mostra un messaggio all'utente, con i seguenti argomenti: *com* è un riferimento ad un componente (la finestra di dialogo viene visualizzata vicino alla finestra principale a cui il componente appartiene, ovvero al centro dello schermo se tale argomento vale *null*), *mesg* specifica l'oggetto che viene visualizzato nell'area principale della finestra di dialogo (normalmente una stringa); *titolo* costituisce il titolo della finestra; *tipo* indica l'icona che si vuole associare al messaggio (possibili valori sono le costanti definite nella classe *JOptionPane*: *PLAIN_MESSAGE* (nessuna icona), *ERROR_MESSAGE* (icona con il simbolo "-"), *INFORMATION_MESSAGE* (icona con il simbolo "i"), *WARNING_MESSAGE* (icona con il simbolo "!"), e *QUESTION_MESSAGE* (icona con il simbolo "?").

Figura 3.3 Una finestra di dialogo per mostrare un messaggio.

Consideriamo, per esempio, il seguente frammento di programma:

```
JOptionPane.showMessageDialog(null, "Risu",
                    "Risultato:",
                    JOptionPane.INFORMATION_MESSAGE);
```

Esso produce la finestra di dialogo mostrata in Fig. 3.3.

Un altro metodo statico della classe *JOptionPane* è il seguente:

static String showInputDialog (Component *com* ***,*** ***Object*** *mesg* ***,***
String *titol* ***,*** ***int*** *tipo* ***)***

visualizza una finestra di dialogo che attende l'immissione di una stringa da parte dell'utente; gli argomenti sono simili a quelli del caso precedente, e la stringa immessa viene restituita come valore di ritorno.

Una forma semplificata di questo metodo è la seguente:

static String showInputDialog (Object *mesg* ***)***

la finestra viene visualizzata al centro dello schermo, il titolo è *Input* e il tipo è *QUESTION_MESSAGE*.

Consideriamo, per esempio, il seguente frammento di programma:

```
String str = JOptionPane.showInputDialog
                    ("Scrivi una stringa:");
```

Esso produce la finestra di dialogo mostrata in Fig. 3.4.

Figura 3.4. Una finestra di dialogo per prelevare una stringa.

Per gli altri metodi della classe *JOptionPane* rimandiamo alla documentazione delle API Java.

Notare che il programmatore può usare direttemene la classe *JDialog* per produrre finestre di dialogo che abbiano un aspetto e un comportamento diverso da quelli che possono essere ottenuti con i metodi della classe *JOptionPane*. La classe *JDialog*, tra l'altro, dispone di un metodo *setDefaultCloseOperation()* analogo a quello della classe *JFrame*.

Quando viene eseguito un metodo *showXXXDialog()* si attiva il thread *swing*: questo non termina quando la finestra viene chiusa, ma deve essere fatto terminare esplicitamente, per esempio usando nel thread *main* il comando *System.exit()*.

Un esempio riepilogativo sull'uso della classe *JOptionPane* è il seguente (vedi anche la Fig. 3.5):

Figura 3.5 Finestre di dialogo prodotte dall'esecuzione della classe *Operazione*.

```
import javax.swing.*;
public class Prova
{ public static void main (String[] args)
  { String str, str1; int n=0; double d=0, s;
    str = JOptionPane.showInputDialog
         ("Scrivi un numero intero:");
    str1 = JOptionPane.showInputDialog
         ("Scrivi un numero reale:");
    try {  n = Integer.parseInt(str);
           d = Double.parseDouble(str1);
         }
    catch (NumberFormatException e)
    { Console.scriviStringa(e.getMessage());
      System.exit(1);
    }
    // la gestione dell'ecccezione non e` obbligatoria
    s = n + d;
    JOptionPane.showMessageDialog
      (null, "Somma: " + s, "Risultato della somma",
       JOptionPane.INFORMATION_MESSAGE);
    System.exit(0);
  }
}
```

La classe *JFileChooser* consente di creare delle finestre di dialogo utili a navigare all'interno del file-system e a scegliere file o cartelle. I costruttori ***JFileChooser()***, ***JFileChooser(File** c)* e ***JFileChooser(String** c)* creano un oggetto per la selezione dei file, eventualmente specificando la cartella iniziale *c*. La visualizzazione della finestra si ottiene col metodo:

***public int showOpenDialog (Component** com)*
com è il riferimento del componente vicino al quale deve essere visualizzata la finestra di selezione; il valore restituito appartiene all'insieme delle costanti della classe *JFileChooser APPROVE_OPTION*, *CANCEL_OPTION* e *ERROR_OPTION*, a seconda che l'utente abbia rispettivamente selezionato un file, abbia cancellato l'operazione, o si sia verificata una situazione di errore.

L'indicazione del file o dei file selezionati dall'utente si ottiene con i seguenti metodi:

public File getSelectedFile ()
public File [] getSelectedFiles ()

A titolo di esempio, riportiamo un semplice programma che crea una finestra per la selezione dei file e stampa sulla console il nome del file scelto (Fig. 3.6).

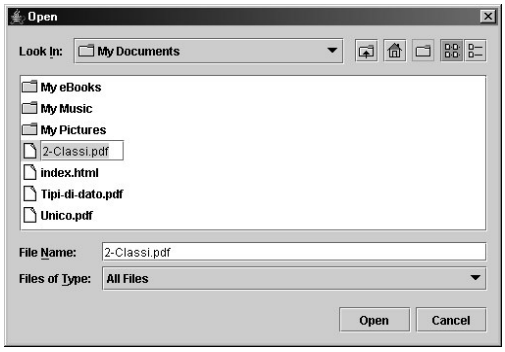


Figura 3.6. Una finestra per la selezione di file.

```
import javax.swing.*;
public class ScegliFile
{ public static void main(String[] args)
  { JFileChooser fc = new JFileChooser();
    int ris = fc.showOpenDialog(null);
    if(ris == JFileChooser.APPROVE_OPTION)
    { Console.scriviStringa("Il file scelto e': " +
                            fc.getSelectedFile());
    }
  }
}
```

3.4.3. Applet

Un applet è una finestra integrata all'interno dell'area di visualizzazione di un browser. Pertanto un applet non è in grado di avere vita propria, ma deve essere necessariamente eseguito all'interno di un browser che ne regola il ciclo di vita. Per realizzare un nuovo applet il

programmatore deve definire una sottoclasse di *javax.swing.JApplet* (o di *java.applet.Applet* se vuole far uso della libreria AWT). Dal punto di vista della gestione degli elementi grafici, un applet è costituito da un pannello, il cui riferimento si ottiene col metodo ***Container getContentPane(),*** nel quale si possono fare disegni e al quale si possono aggiungere etichette, bottoni, eccetera.

Per quanto riguarda invece gli altri aspetti degli applet, legati allo sviluppo di applicazioni Internet, rimandiamo il lettore al Capitolo 4.

3.5. Pannelli

Un pannello è un componente costituito da uno spazio senza alcuna cornice, e costituisce un contenitore in cui possono essere inseriti altri componenti. Un pannello non appartiene al gruppo dei contenitori ad alto livello, e pertanto non può essere usato per creare delle finestre a se stanti (principali, di dialogo o all'interno di un browser), ma può essere invece adoperato per inserirvi una serie di componenti, raggruppandoli in un'unica entità (il pannello può essere trattato come un singolo componente).

Figura 3.7. Pannello con due etichette in una finestra.

Il gestore di layout default di un pannello è di tipo *FlowLayout*. Invece, il gestore di layout default di una finestra (o meglio, del suo *ContentPane*) è tale per cui (vedi paragrafo successivo) l'inserimento di un componente (pannello), senza altre specifiche, produce l'occupazione di tutta l'area del contenitore (finestra).

I pannelli possono essere semplici o a scorrimento, in relazione alla classe a cui appartengono (*JPanel* o *JScrollPane*).

La classe *JPanel* serve a creare pannelli semplici e possiede un costruttore senza argomenti. L'aggiunta di componenti ad un pannello avviene in maniera simile a quanto visto nel caso dei contenitori visti in precedenza.

Come esempio, consideriamo il seguente programma, che produce un risultato del tipo di quello illustrato in Fig. 3.7.

```
import javax.swing.*; import java.awt.*;
public class Pann
{ public static void main(String[] args)
  { JFrame finestra = new JFrame("Vai");
    finestra.setDefaultCloseOperation
      (JFrame.DISPOSE_ON_CLOSE);
    finestra.setSize(200, 150);
    Container c = finestra.getContentPane();
    JPanel  pannello = new JPanel();
    pannello.setBackground(Color.red);
    JLabel la = new JLabel("Uno");
    la.setForeground(Color.yellow); pannello.add(la);
    Label lb = new JLabel("Due");
    lb.setForeground(Color.green); pannello.add(lb);
    c.add(pannello);
    finestra.setVisible(true);
  }
}
```

La classe *JScrollPane* serve invece a creare pannelli dotati di barre di scorrimento. Nel caso in cui un pannello contenga un componente di dimensioni superiori a quelle del pannello stesso, le barre consentono di visualizzare la parte non visibile del componente. Un pannello a scorrimento è quindi utile quando si vogliono visualizzare componenti di grosse dimensioni o componenti di dimensione variabile. Oltre al costruttore senza argomenti, la classe *JScrollPane* consente di creare un

pannello a scorrimento che contiene il componente *c* mediante il costruttore ***JScrollPane(Component c)***.

3.6. Gestori di layout

Come abbiamo visto, la classe *Container* prevede un metodo per impostare un nuovo gestore di layout al posto di quello predefinito per quel contenitore. Nel caso dei contenitori ad alto livello (appartenenti alle classi *JFrame*, *JDialog* e *JApplet*) tale metodo deve essere applicato al panello restituito da *getContentPane()*. Nel caso di un oggetto appartenente alla classe *JPanel*, tale metodo può essere invocato direttamente.

Oltre al gestore di layout di tipo *FlowLayout*, sono di largo utilizzo alcuni altri gestori, come illustrato nei sottoparagrafi seguenti.

3.6.1. BorderLayout

Un gestore *BorderLayout* divide il contenitore in cinque zone, ognuna delle quali può contenere un componente, come mostrato in Fig. 3.8 (non è necessario usare tutte le zone). Le dimensioni di ogni zona sono la massima possibile per la zona centrale, e le minime necessarie per contenere il componente per le altre zone. Le cinque zone sono identificate dalle costanti della classe *BorderLayout NORTH*, *SOUTH*, *WEST, CENTER*, *EAST*. Il gestore *BorderLayout* è predefinito per i contenitori di alto livello.

Nota: Nelle versioni del linguaggio a partire dalla 1.4, le cinque zone sono identificate dalle costanti *PAGE_START*, *PAGE_END*, *LINE_START*, *CENTER, LINE_END*. I nuovi nomi supportano in maniera più semplice lo sviluppo di applicazioni che adottano lingue in cui la scrittura avviene con direzioni diverse da quella occidentale.

Quando si aggiunge un componente ad un contenitore gestito con la strategia *BorderLayout*, è opportuno usare una versione del metodo *add()* che specifica la zona a cui il componente deve essere associato (se la zona

non viene specificata, l'aggiunta avviene sempre nella zona centrale). Come esempio, consideriamo il seguente programma, in cui si aggiungono pannelli (istanze della classe *JPanel*) a una finestra:

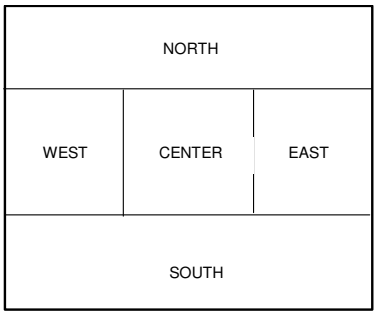


Figura 3.8. Regioni previste da un gestore *BorderLayout*.

```
import javax.swing.*; import java.awt.*;
public class Bord
{ public static void main (String[] args)
  { JFrame finestra = new JFrame("Ciao");
    finestra.setDefaultCloseOperation
         (JFrame.DISPOSE_ON_CLOSE);
    finestra.setSize(200, 200);
    Container c = finestra.getContentPane();
    JPanel pan1 = new JPanel(), pan2 = new JPanel(),
    pan3 = new JPanel(), pan4 = new JPanel(),
    pan5 = new JPanel();
    pan1.setBackground(Color.red);
    pan2.setBackground(Color.black);
    pan3.setBackground(Color.green);
    pan4.setBackground(Color.blue);
    pan5.setBackground(Color.yellow);
    c.add(pan1, BorderLayout.EAST );
    c.add(pan2, BorderLayout.WEST);
    c.add(pan3, BorderLayout.NORTH);
    c.add(pan4, BorderLayout.SOUTH);
    c.add(pan5, BorderLayout.CENTER);
    finestra.setVisible(true);
  }
}
```

Figura 3.9. Finestra gestita con strategia *BorderLayout*.

In esecuzione otteniamo una finestra simile a quella di Fig. 3.9. Il pannello centrale occupa la massima area, mentre gli altri la minima possibile.

3.6.2. GridLayout

Un gestore *GridLayout* divide il contenitore in rettangoli di uguale dimensione disposti come in una griglia. Il numero di righe e di colonne che formano la griglia è definito attraverso il costruttore ***GridLayout(int** righe**, int** colonne**)**.*

Prendiamo come esempio il seguente programma (vedi anche la Fig. 3.11):

```
import javax.swing.*;
import java.awt.*;
public class Grid
{ public static void main(String[] args)
  { JFrame finestra = new JFrame("Titolo");
    finestra.setDefaultCloseOperation
      (JFrame.DISPOSE_ON_CLOSE);
    Container c = finestra.getContentPane();
    c.setLayout(new GridLayout(3, 2));
    c.add(new JButton("Uno"));
    c.add(new JButton("Due"));
```

```
    c.add(new JButton("Tre"));
    c.add(new JButton("Quattro"));
    c.add(new JButton("Cinque"));
    finestra.setSize(200, 100);
    finestra.setVisible(true);
  }
}
```

Figura 3.11. Una finestra gestita con la strategia *GridLayout*.

3.6.3. Layout complessi

Per disporre i componenti in maniera più complessa si devono usare più contenitori, ognuno con il suo gestore di layout, annidati l'uno dentro l'altro.

Il seguente programma crea due pannelli, *pan1* e *pan2*, e li associa rispettivamente alle zone *SOUTH* e *CENTER* della finestra principale. Il pannello *pan1* contiene due bottoni, *bot1* e *bot2*, e li gestisce secondo la strategia *FlowLayout*. Il pannello *pan2* contiene i bottoni *bot3*, *bot4* e *bot5*, e li dispone in una griglia composta da tre righe e una colonna. Il bottone *bot6* è associato alla zona *EAST* della finestra principale.

```
import javax.swing.*;
import java.awt.*;
public class Annidamento
{ public static void main(String[] args)
  { JFrame finestra = new JFrame("Annidamento");
    finestra.setDefaultCloseOperation
             (JFrame.DISPOSE_ON_CLOSE);
    JPanel pan1 = new JPanel();
    JPanel pan2 = new JPanel();
```

```
        pan1.setLayout(new FlowLayout()); // superfluo
        pan2.setLayout(new GridLayout(3, 1));
        JButton bot1 = new JButton("Bottone 1");
        JButton bot2 = new JButton("Bottone 2");
        JButton bot3 = new JButton("Bottone 3");
        JButton bot4 = new JButton("Bottone 4");
        JButton bot5 = new JButton("Bottone 5");
        JButton bot6 = new JButton("Bottone 6");
        pan1.add(bot1);
        pan1.add(bot2);
        pan2.add(bot3);
        pan2.add(bot4);
        pan2.add(bot5);
        Container c = finestra.getContentPane();
        c.add(pan1, BorderLayout.SOUTH);
        c.add(pan2, BorderLayout.CENTER);
        c.add(bot6, BorderLayout.EAST);
        finestra.setSize(300, 300);
        finestra.setVisible(true);
    }
}
```

Eseguendo il programma si ottiene una finestra in cui i componenti sono disposti come indicato in Fig. 3.12.

Figura 3.12. Finestra con layout complesso.

3.7. Gestione degli eventi

Un'azione effettuata dall'utente su un elemento grafico può produrre un *evento*. Alcune azioni tipiche che producono un evento sono le seguenti:

- premere il tasto "enter" dopo aver scritto in un campo di testo;
- fare click col mouse su un bottone;
- selezionare un elemento di una lista.

Ogni evento ha un suo tipo predefinito (tipo classe): per esempio, premere il tasto "enter" in un campo di testo o premere un bottone producono eventi della classe *ActionEvent.*

Un'azione dell'utente produce la creazione di un oggetto evento (*firing*), il quale può venir catturato da un ascoltatore (*listener*), che specifica le azioni da eseguire (ogni elemento grafico può essere associato a un determinato ascoltatore). Un ascoltatore è un oggetto di una classe relativa a quel tipo di eventi: tale classe implementa un'interfaccia specifica, che contiene la dichiarazione di un metodo per la gestione degli eventi di quel tipo. Le classi relative ai vari tipi di eventi e le interfacce di gestione dei tipi di eventi appartengono di norma al package *java.awt.event.*

Supponiamo di voler scrivere un programma che gestisca gli eventi prodotti da un elemento grafico, quale un bottone. È opportuno strutturare il programma come illustrato di seguito.

1. Definire l'elemento grafico.
   Esempio:

   ```
   JButton b = new JButton("Premi");
   ```

   Esso, quando viene premuto, produce un evento di tipo *ActionEvent*.

2. Definire una classe ascoltatrice del tipo di evento prodotto, la quale deve implementare un'interfaccia legata al tipo dell'evento stesso (la classe ascoltatrice di eventi del tipo *ActionEvent* deve implementare l'interfaccia *ActionListener,* che prevede il metodo ***void actionPerformed(ActionEvent*** e***)***).

Esempio:

```
class AscoltatoreBottoni implements ActionListener
{ public void actionPerformed(ActionEvent e)
  {
    // gestione del tipo di eventi
  }
}
```

3. Creare un oggetto ascoltatore (appartenente alla classe precedente), e registrarvi l'elemento grafico considerato (per la classe *JButton* quest'ultima operazione viene eseguita per mezzo del metodo ***void addActionListner(ActionListener l)***).
   Esempio:

```
AscoltatoreBottoni ab = new AscoltatoreBottoni();
b.addActionListener(ab);
```

Come esempio completo di gestione di eventi, riportiamo il seguente semplice programma, che incrementa e scrive il nuovo valore di un contatore ogni volta che si preme il bottone:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Fin extends JFrame
{ private int cont = 0;
  private JButton bb = new JButton("++");
  public class AscoltatoreBB
         implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { cont++;
      Console.scriviStringa("Contatore = " + cont);
    }
  }
  Fin(String s)
  { super(s);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    Container c = getContentPane();
    c.add(bb);
    AscoltatoreBB as = new AscoltatoreBB();
```

```
    bb.addActionListener(as);
  }
}

public class ProvaEv
{ public static void main(String[] args)
  { Fin ff = new Fin("Bottone");
    ff.setSize(200, 100);
    ff.setVisible(true);
  }
}
```

In un contenitore possono esserci più elementi grafici, dello stesso tipo o di tipo diverso, che generano eventi appartenenti alla stessa classe o a classi diverse. Per ogni classe di eventi, esistono metodi che consentono di risalire all'elemento grafico che ha prodotto quel tipo di evento. Per esempio, nella classe *ActionEvent*, il metodo ***Object getSource()*** restituisce il riferimento all'oggetto che ha prodotto l'evento. Per oggetti delle classi *JButton* e *JTextField*, il rispettivo metodo ***String getText()*** restituisce o l'etichetta contenuta nel bottone o il testo contenuto nella casella di testo.

Come esempio riepilogativo riportiamo un programma con una finestra dotata di due bottoni (inseriti nelle zone NORTH e SOUTH) e due pannelli (inseriti nelle zone WEST e EAST). Quando viene premuto, un bottone cambia la sua etichetta da "SI" a "NO" e viceversa, mentre l'altro produce l'incremento e la stampa su video di un contatore. I due pannelli, ciascuno con una etichetta e un'area di testo, consentono di acquisire e stampare su video le coordinate x e y di un punto del piano.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Finestra extends JFrame
{ private int cont = 0;
  private JButton uno = new JButton("SI"),
              incr = new JButton("Incrementa");
  private JLabel x = new JLabel("x = ");
  private JTextField xx = new JTextField(5);
  private JLabel y = new JLabel("y =");
  private JTextField yy = new JTextField(5);
  public class AscoltatoreBottoni
                    implements ActionListener
  { public void actionPerformed(ActionEvent e)
```

```
    { JButton b = (JButton)e.getSource();
      if(b == uno)
      { if (b.getText()=="SI") b.setText("NO");
        else b.setText("SI"); }
      else { cont++; Console.scriviStringa
               ("Contatore = " + cont); }
    }
  }
  public class AscoltatoreTesti
                    implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { JTextField f = (JTextField)e.getSource();
      String s = f.getText();
      if (f == xx) Console.scriviStringa
                              ("Ascissa = "  + s);
      else Console.scriviStringa("Ordinata = " + s);
    }
  }
  Finestra(String s)
  { super(s);
    setDefaultCloseOperation
                    (DISPOSE_ON_CLOSE);
    Container c = getContentPane();
    JPanel pw = new JPanel(); pw.add(x); pw.add(xx);
    pw.setBackground(Color.yellow);
    JPanel pe = new JPanel(); pe.add(y); pe.add(yy);
    pe.setBackground(Color.green);
    c.add(uno, BorderLayout.NORTH);
    c.add(incr, BorderLayout.SOUTH);
    c.add(pw, BorderLayout.WEST);
    c.add(pe, BorderLayout.EAST);
    AscoltatoreTesti at  = new AscoltatoreTesti();
    xx.addActionListener(at);
    yy.addActionListener(at);
    AscoltatoreBottoni ab = new AscoltatoreBottoni();
    uno.addActionListener(ab);
    incr.addActionListener(ab);
  }
}

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

```
  { Finestra ff = new Finestra("Eventi");
    ff.setSize(250, 120); ff.setVisible(true);
  }
}
```

In esecuzione, il programma produce una finestra del tipo di quella riportata in Fig. 3.13.

Figura 3.13. Bottoni e pannelli in una finestra.

E' importante notare che il codice dei metodi di gestione degli eventi viene sempre eseguito dal thread *swing*. Questo implica che il sistema è in grado di gestire un solo evento alla volta. Pertanto per mantenere elevata la reattività dell'interfaccia grafica, in risposta ai comandi dell'utente, è necessario che il codice di gestione non richieda troppo tempo per essere completato (gli eventuali eventi generati mentre è ancora in corso la gestione di un evento precedente vengono accodati e gestiti solo quando arriva il loro turno).

3.7.1. Eventi generati da mouse e tastiera

Gli eventi generati dall'utente attraverso il mouse (movimento, click, eccetera) sono oggetti dalla classe *MouseEvent*. Ogni oggetto contiene varie informazioni, comprese le coordinate *x* e *y* della posizione in cui l'evento si è verificato (per conoscere questi valori esistono i metodi ***int getX()*** e ***int getY()***). Per catturare eventi appartenenti alla classe *MouseEvent* è necessario definire classi ascoltatrici che implementino almeno una delle due seguenti interfacce:

- *MouseListener* se si è interessati a eventi quali la pressione di un tasto del mouse o l'entrata o uscita del mouse nell'area occupata da un componente;
- *MouseMotionListener* se si è interessati a ricevere notifiche riguardanti il movimento del mouse.

Ogni componente, in quanto può generare eventi legati al mouse, deve quindi registrarsi presso un oggetto ascoltatore.

L'interfaccia *MouseListener* prevede i seguenti metodi di gestione:

public void mouseClicked (MouseEvent e)
invocato quando viene effettuato un click (viene premuto e rilasciato uno dei tasti del mouse, senza lasciare l'area occupata dal componente);

public void mousePressed (MouseEvent e)
invocato quando uno dei tasti del mouse viene premuto;

public void mouseReleased (MouseEvent e)
invocato quando uno dei tasti del mouse viene rilasciato;

public void mouseEntered (MouseEvent e)
invocato quando il puntatore entra nell'area occupata dal componente;

public void mouseExited (MouseEvent e)
invocato quando il puntatore esce dall'area del componente.

L'interfaccia *MouseMotionListener* prevede due metodi di gestione:

public void mouseMoved (MouseEvent e)
invocato quando il puntatore viene spostato senza che alcun tasto sia premuto;

public void mouseDragged (MouseEvent e)
invocato quando uno dei tasti è premuto e il puntatore viene spostato (trascinamento).

Quando si verifica un evento determinato dalla pressione o dal rilascio di un tasto del mouse, la classe *SwingUtilities* prevede i seguenti metodi per risalire al tasto che ha prodotto l'evento:

```
public static boolean isLeftMouseButton ( MouseEvent e )
public static boolean isMiddleMouseButton ( MouseEvent e )
public static boolean isRightMouseButton ( MouseEvent e )
```

A titolo di esempio, riportiamo un programma che crea una finestra (vuota) e le associa un ascoltatore di eventi. Al verificarsi di uno degli eventi definiti nell'interfaccia *MouseListener*, l'ascoltatore stampa sul video alcune informazioni riguardanti l'evento:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Mouse extends JFrame
{ public class AscoltatoreM implements MouseListener
  { private String qualeBottone(MouseEvent e)
    { if(SwingUtilities.isLeftMouseButton(e))
          return "Bottone sinistro";
      if(SwingUtilities.isRightMouseButton(e))
          return "Bottone destro";
      if(SwingUtilities.isMiddleMouseButton(e))
          return "Bottone centrale";
      return "Bottone sconosciuto";
    }
    public void mouseClicked(MouseEvent e)
    { Console.scriviStringa(qualeBottone(e) +
        " click: (X, Y) = (" + e.getX() + ", " +
        e.getY() + ")" );
    }
    public void mousePressed(MouseEvent e)
    { Console.scriviStringa
          (qualeBottone(e) + " premuto");}
    public void mouseReleased(MouseEvent e)
    { Console.scriviStringa
          (qualeBottone(e) + " rilasciato");}
    public void mouseEntered(MouseEvent e)
    { Console.scriviStringa("Ingresso");  }
    public void mouseExited(MouseEvent e)
```

```
    { Console.scriviStringa("Uscita");  }
  }
  Mouse(String s)
  { super(s);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    AscoltatoreM asc = new AscoltatoreM();
    addMouseListener(asc);
    // il metodo viene applicato a un oggetto Mouse
  }
}

public class ProvaMouse
{ public static void main(String[] args)
  { Mouse ff = new Mouse("Mouse");
    ff.setSize(200, 100);
    ff.setVisible(true);
  }
}
```

In maniera analoga è possibile gestire eventi causati dalla pressione o dal rilascio di tasti sulla tastiera, che appartengono alla classe *KeyEvent*. Questi vengono catturati da oggetti di una classe che implementa l'interfaccia *KeyListener*, che prevede tre metodi di gestione:

public void keyPressed (KeyEvent e)
invocato quando viene premuto un tasto (il metodo ***int getKeyCode()*** della classe *KeyEvent* restituisce il codice del tasto premuto; nella classe sono anche definite delle costanti associate ai tasti, per esempio *VK_LEFT* (codice del tasto direzione sinistra));

public void keyReleased (KeyEvent e)
simile al caso precedente, ma è invocato quando un tasto viene rilasciato;

public void keyTyped (KeyEvent e)
invocato quando viene immesso un carattere Unicode (l'immissione di un singolo carattere può richiedere la pressione di più tasti, come per esempio "shift" + "a") (il metodo ***char getKeyChar()*** della classe *KeyEvent* restituisce il carattere immesso).

3.8. La classe *JComponent*

La classe *JComponent* è la classe base di tutti i componenti grafici previsti dalla libreria *Swing*, con esclusione dei contenitori di alto livello. Essa è una classe astratta e pertanto non può essere istanziata. I nuovi componenti definiti dal programmatore devono appartenere a classi derivate, direttamente o indirettamente, da *JComponent*, e vengono in tal modo a far parte di una gerarchia di componenti.

Alcune delle funzionalità principali della classe *JComponent* sono:

- *messaggi di aiuto*: la stringa specificata con il metodo ***void setToolTipText(String s)*** viene visualizzata quando il puntatore del mouse si sofferma su un componente;
- *bordi*: ad ogni componente possono essere associati dei bordi, eventualmente caratterizzati da decorazioni, mediante il metodo ***void setBorder(Border b)***; la classe *BorderFactory* fornisce metodi per la creazione di oggetti *Border* con caratteristiche diverse: per esempio, la seguente istruzione associa un bordo dotato di una riga blu al componente *c*:

```
c.setBorder(
      BorderFactory.createLineBorder(Color.blue));
```

- *gestione di posizione e dimensione*: i metodi ***int getWidth()*** e ***int getHeight()*** restituiscono la larghezza e l'altezza del componente espresse in pixel; i metodi ***int getX()*** e ***int getY()*** restituiscono le coordinate dell'origine (in alto a sinistra) del componente; metodi analoghi consentono di impostare la dimensione e la posizione di un componente, anche se sono di scarso utilizzo grazie all'uso dei gestori di layout, che provvedono a dimensionare e a posizionare i componenti in maniera più flessibile;
- *gestione di colori e fonti*: i metodi (già visti nel paragrafo 3.3) ***void setBackground(Color c)***, ***void setForeground(Color c)*** e ***void setFont(Font f)*** possono essere usati per il colore dello sfondo (*background*) o il colore degli elementi in primo piano (*foreground*), e per definire la fonte associata al componente.

Passiamo a descrivere alcuni tra i principali componenti inclusi nella libreria *Swing*.

3.9. Componenti della libreria *Swing*

Tra i componenti di largo utilizzo sono compresi le etichette, i bottoni e le caselle di testo, già introdotti in forma semplificata nel sottoparagrafo 3.3.2.

3.9.1. Etichette e loro proprietà

La classe *JLabel* consente di specificare sia il testo dell'etichetta che un'eventuale icona associata al testo stesso. Questo può essere inizialmente ottenuto con un costruttore (***JLabel(String** s)*, ***JLabel(Icon** ic)* e ***JLabel(String** s, **Icon** ic)*), ovvero con i metodi ***void setText(String** s)* e ***void setIcon(Icon** ic)*, che possono essere impiegati anche per effettuare modifiche.

Nota: un'icona è una piccola immagine di dimensioni fisse usata per decorare etichette, bottoni ed altri componenti grafici. Per creare una icona a partire da una immagine memorizzata in un file è opportuno usare la classe *ImageIcon* (che implementa l'interfaccia *Icon*). Il costruttore ***ImageIcon(String** nomefile)* ha come argomento il nome del file contenente l'immagine (in formato *GIF*, *JPG* o *PNG*).

Invece di utilizzare metodi appositi per impostare la fonte, il colore ed altre proprietà, il testo dell'etichetta può includere dei tag HTML, come nel seguente esempio:

Figura 3.14. Esecuzione della classe *Etichette*.

```
import java.awt.*;
import javax.swing.*;
public class Etichette
```

```
{ public static void main(String[] args)
  { JFrame fr = new JFrame("Esempio etichette");
    fr.setDefaultCloseOperation
             (JFrame.DISPOSE_ON_CLOSE);
    Container c = fr.getContentPane();
    c.setLayout(new FlowLayout());
    JLabel l1 = new JLabel("Etichetta semplice");
    JLabel l2 = new JLabel
         ("<html><h1>Etichetta grande</h1></html>");
    c.add(l1);
    c.add(l2);
    fr.setSize(350, 80);
    fr.setVisible(true);
  }
}
```

In esecuzione, viene prodotta una finestra del tipo di quella riportata in Fig. 3.14.

3.9.2. Bottoni e relative classi

I bottoni possono appartenere a tre classi: *JButton*, *JCeckBox* e *JRadioButton*.

La classe *JButton* prevede costruttori (***JButton(String** s), **JButton(Icon** ic)* e ***JButton(String** s, **Icon** ic)*) per creare dei bottoni dotati di una stringa di testo, di una icona o di entrambe. In maniera analoga alle etichette, il testo associato ad un bottone può essere espresso in HTML. La stringa associata al bottone può essere modificata con il metodo ***void setText(String** s)*.

All'interno di un contenitore principale, uno dei bottoni può essere indicato come il bottone default: in tal caso viene visualizzato in maniera evidenziata ed una eventuale pressione del tasto "Enter" è equivalente all'esecuzione di un click con il mouse sul bottone stesso. Il bottone default viene specificato a mezzo del metodo ***void setDefaultButton(JButton** b)*, che deve essere invocato sul pannello radice del contenitore di alto livello che contiene il bottone stesso (un riferimento al pannello radice viene ottenuto invocando il metodo ***JRootPane getRootPane()*** sul contenitore di alto livello, per esempio un *JFrame*).

La classe *JCeckBox* rappresenta caselle di controllo con doppio valore di stato (*true* e *false*). Il costruttore ***JCheckBox(String** s)* costruisce un

oggetto di tipo *JCheckBox* dove *s* è la stringa associata alla casella. Altri costruttori consentono di specificare lo stato iniziale della casella ed una eventuale icona. Lo stato di una casella di controllo, restituito dal metodo ***boolean isSelected()***, può essere modificato facendo click con il mouse sulla casella stessa.

Quando viene variato lo stato della casella di controllo, si genera un evento appartenente alla classe *ItemEvent*. Essa prevede i metodi ***Object getSource()*** e ***int getStateChange()*** che restituiscono, rispettivamente, l'oggetto che ha prodotto l'evento e una delle costanti *ItemEvent.SELECTED* o *ItemEvent.DESELECTED.* Un gestore di eventi della classe *ItemEvent* deve implementare l'interfaccia *ItemListner*, che prevede il metodo ***void itemStateChanged(ItemEvent e)***.

La classe *JRadioButton* rappresenta caselle radio con doppio valore di stato da accomunare in un insieme, tale che solo una alla volta risulta selezionata: una nuova selezione determina la deselezione della precedente. Le singole caselle radio possono venir create con il costruttore ***JRadioButton(String s)***, dove *s* specifica la stringa da associare alla casella, o con altri costruttori che consentono di specificare lo stato iniziale della casella ed una eventuale icona. Successivamente, le caselle radio create devono essere associate a un oggetto appartenente alla classe *ButtonGroup* utilizzando il metodo ***add()*** definito in questa classe. Lo stato di una casella radio, restituito dal metodo ***boolean isSelected()***, può essere modificato facendo click con il mouse sulla casella stessa.

Riportiamo un programma riepilogativo che impiega due caselle di controllo, due caselle radio, un'etichetta e un bottone default. Quando viene cambiato lo stato di una caselle, la stringa visualizzata dall'etichetta viene modificata per riflettere il nuovo stato delle caselle. Quando invece viene premuto il bottone o premuto il tasto "Enter", viene incrementato un contatore il cui valore viene visualizzato dal bottone stesso.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Bottoni extends JFrame
{ private int cont = 0;
  private JButton bot = new JButton
                  ("Contatore: " + cont);
  private JLabel etic = new JLabel
                    ("Fare click sulle caselle");
```

```
  private JCheckBox cb1 = new JCheckBox("Scelta 1");
  private JCheckBox cb2 = new JCheckBox("Scelta 2");
  private JRadioButton rb1 = new JRadioButton("Si");
  private JRadioButton rb2 = new JRadioButton("No");
  private ButtonGroup gruppo = new ButtonGroup();
  private Container c;
  public class AscoltatoreB
         implements ActionListener, ItemListener
  { public void actionPerformed(ActionEvent e)
    { cont++;
      bot.setText("Contatore: " + cont);
    }
    public void itemStateChanged(ItemEvent e)
    { String s = "Stato dei bottoni: " +
                 cb1.isSelected() + ", " +
                 cb2.isSelected() + ", " +
                 rb1.isSelected() + ", " +
                 rb2.isSelected();
      etic.setText(s);
    }
  }
  Bottoni(String titolo)
  { super(titolo);
    c = getContentPane();
    c.setLayout(new GridLayout(6, 1));
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    etic.setBorder
      (BorderFactory.createLineBorder(Color.blue));
    gruppo.add(rb1); gruppo.add(rb2);
    c.add(cb1); c.add(cb2);
    c.add(rb1); c.add(rb2);
    c.add(etic);
    c.add(bot);
    getRootPane().setDefaultButton(bot);
    AscoltatoreB asc = new AscoltatoreB();
    bot.addActionListener(asc);
    cb1.addItemListener(asc);
    cb2.addItemListener(asc);
    rb1.addItemListener(asc);
    rb2.addItemListener(asc);
  }
}
```

```
public class ProvaB
{ public static void main(String[] args)
  { Bottoni bb = new Bottoni("Bottoni");
    bb.setSize(280, 200);
    bb.setVisible(true);
  }
}
```

Il programma visualizza una finestra (Fig. 3.15) contenente due *checkbox*, due *radio button*, un'etichetta e un bottone.

Figura 3.15. Una finestra contenente due caselle di controllo, due caselle radio, un'etichetta e bottone default.

3.9.3. Aree di testo

La classe *JTextField* consente di gestire singole linee di testo, come illustrato nel sottoparagrafo 3.2.2.

Il seguente programma definisce la classe *InvertiTesto*, sottoclasse di *JTextField*, che alla pressione del tasto di ritorno carrello reagisce invertendo la stringa digitata. Il programma inoltre crea una semplice finestra contenente una etichetta e una istanza di *InvertiTesto* (vedi anche la Fig. 3.16):

```
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
class InvertiTesto extends JTextField
{ class AscoltatoreT implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { String s = getText();
      char[] car = s.toCharArray();
      char[] res = new char[car.length];
      for(int i=0; i<car.length; i++)
        res[res.length - i-1] = car[i];
      String inv = new String(res);
      setText(inv);
    }
  }
  InvertiTesto(int colonne)
  { super(colonne);
    AscoltatoreT at = new AscoltatoreT();
    addActionListener(at);
  }
}

class FrameTF extends JFrame
{ FrameTF(String s)
  { super(s);
    setDefaultCloseOperation
        (JFrame.DISPOSE_ON_CLOSE);
    Container c = getContentPane();
    c. setLayout(new GridLayout(2, 1));
    JLabel la = new JLabel("Inserisci del testo "+
        "e premi enter");
    c.add(la);
    InvertiTesto it = new InvertiTesto(20);
    c.add(it);
  }
}

public class ProvaIT
{ public static void main(String[] args)
  { FrameTF fr = new FrameTF("Esempio testo");
    fr.setSize(250, 100);
    fr.setVisible(true);
  }
}
```

Figura 3.16. Inversione di testo.

La classe *JPasswordField* è definita come una sottoclasse di *JTextField* ed è utile per richiedere l'immissione di una password, in quanto i caratteri immessi vengono sostituiti da asterischi.

La visualizzazione e manipolazione di testo su più righe sono supportate dalla classe *JTextArea*. I costruttori ***JTextArea(String** s)*, ***JTextArea(String** s, **int** r, **int** c)* e ***JTextArea(int** r, **int** c)* consentono di specificare la stringa iniziale *s*, il numero di righe *r* e di colonne *c*. Oltre ai metodi ***void setText(String** s)* e ***String getText()***, che hanno un funzionamento analogo a quanto visto per la classe *JTextField*, la classe *JTextArea* dispone dei metodi ***void append(String** s)*, che aggiunge la stringa specificata in fondo al testo, ***void insert(String** s, **int** i)*, che inserisce la stringa *s* nella posizione specificata e ***void replaceRange(String** s, **int** i, **int** f)*, che sostituisce il testo tra le posizioni specificate con una nuova stringa.

3.9.4. Elenchi

La classe *JList* implementa un componente costituito da un elenco di oggetti (tipicamente di stringhe). Il costruttore ***JList(Object[]** elenco)* crea un elenco di oggetti, a selezione singola o a selezione multipla (in questo ultimo caso, per effettuare la selezione, bisogna tenere premuto il tasto *Ctrl*). Il metodo per la scelta del tipo di selezione è ***void setSelectionMode(int** selectionMode)* dove *selectionMode* è una delle seguenti costanti:

ListSelectionModel.SINGLE_SELECTION
un solo elemento può essere selezionato;

ListSelectionModel.SINGLE_INTERVAL_SELECTION

si possono selezionare più elementi a patto che siano contigui;

ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
si possono selezionare più elementi anche non contigui tra di loro.

I metodi che restituiscono gli indici degli elementi selezionati sono:

public int getSelectedIndex ()
restituisce l'indice del primo elemento selezionato;

public int [] getSelectedIndices ()
restituisce un array con gli indici di tutti gli elementi selezionati (in ordine crescente).

I metodi che restituiscono i riferimenti degli oggetti selezionati sono:

public Object getSelectedValue ()
restituisce il riferimento del primo oggetto selezionato;

public Object [] getSelectedValues ()
restituisce un array con i riferimenti di tutti gli oggetti selezionati (in ordine di indice crescente).

Quando un elemento viene selezionato o deselezionato, viene prodotto un evento della classe *ListSelectionEvent* (package *javax.swing.event*). Il gestore deve implementare l'interfaccia *ListSelectionListener,* che prevede il metodo ***void valueChanged(ListSelectionEvent e)***.

Come esempio riportiamo il seguente programma:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
class Elenco extends JFrame
{ private JList lista;
  private JLabel etic;
  Elenco(Object[] oa)
  { super("Esempio elenco");
    lista = new JList(oa);
    getContentPane().add(lista);
```

```
    etic = new JLabel("Scegli");
    getContentPane().add(etic, BorderLayout.SOUTH);
    lista.addListSelectionListener(new Ascoltatore());
    lista.setSelectionMode
              (ListSelectionModel.SINGLE_SELECTION);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  }
  public class Ascoltatore
           implements ListSelectionListener
  { public void valueChanged(ListSelectionEvent e)
    { Object o = lista.getSelectedValue();
      etic.setText(o.toString());
    }
  }
}

public class ProvaElenco
{ public static void main(String[] args)
  { String[] list = {"uno", "due", "tre", "quattro"};
    Elenco el = new Elenco(list);
    el.setSize(200, 200);
    el.setVisible(true);
  }
}
```

Figura 3.17. Esempio di finestra con un elenco.

La classe *Elenco* è definita come una estensione di *JFrame* e contiene un oggetto di tipo *JList* e una etichetta. Quando l'utente seleziona uno degli elementi della *JList*, viene invocato il metodo *valueChanged()* della classe interna *Ascoltatore*. Tale metodo provvede ad aggiornare il testo visualizzato dall'etichetta sostituendolo con la stringa selezionata.

Una volta mandato in esecuzione il programma visualizza una finestra simile a quella di Fig. 3.17.

3.9.5. Menu

Un contenitore di alto livello può essere dotato di una barra dei menu, implementata dalla classe *JMenuBar*. La barra è in grado di ospitare più menu, ognuno dei quali è composto da un certo numero di elementi. Una barra dei menu non viene aggiunta al *content pane* come visto per gli altri componenti, ma viene associata direttamente alla finestra tramite il metodo ***void setJMenuBar(JMenuBar** b)* delle classi *JFrame*, *JDialog* e *JApplet*. Una volta creata una istanza di *JMenuBar* (la classe dispone di un costruttore senza argomenti), il metodo ***void add(JMenu** m)* permette di popolare la barra. Gli oggetti di tipo *JMenu* rappresentano i menu e possono essere creati con il costruttore ***JMenu(String** s)*, dove *s* costituisce il titolo del menu. A questo punto è possibile popolare i menu con i loro elementi, rappresentati da istanze di *JMenuItem*. Il più semplice dei costruttori della classe *JMenuItem* prende come argomento la stringa da associare all'elemento. Il metodo ***void addSeparator()*** della classe *JMenu* inserisce un elemento grafico che funge da separatore.

Quando l'utente seleziona uno degli elementi di un menu viene generata una istanza di *ActionEvent*. Pertanto per gestire gli eventi generati da un menu è sufficiente creare una istanza di *ActionListener* e associarlo ai *JMenuItem* usando il metodo *addActionListener()*, in maniera analoga a quanto visto per i bottoni.

Il seguente programma di esempio crea una finestra dotata di una barra dei menu. La barra contiene il solo menu "*File*" costituito dagli elementi "*Apri*", "*Salva*" e "*Esci*". Quando uno degli elementi viene selezionato, la stringa corrispondente viene visualizzata nella parte centrale della finestra.

```
import javax.swing.*;
```

```
import java.awt.event.*;
class MenuFrame extends JFrame
{ private JMenuBar bar = new JMenuBar();
  private JMenu menu = new JMenu("File");
  private JMenuItem apri = new JMenuItem("Apri");
  private JMenuItem salva = new JMenuItem("Salva");
  private JMenuItem esci = new JMenuItem("Esci");
  private Ascoltatore asc = new Ascoltatore();
  private JLabel lab = new JLabel("Agisci sul menu");
  MenuFrame(String s)
  { super(s);
    getContentPane().add(lab);
    apri.addActionListener(asc);
    salva.addActionListener(asc);
    esci.addActionListener(asc);
    bar.add(menu);
    menu.add(apri);
    menu.add(salva);
    menu.addSeparator();
    menu.add(esci);
    setJMenuBar(bar);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  }
  public class Ascoltatore implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { lab.setText("Hai scelto " +
                ((JMenuItem) e.getSource()).getText());
    }
  }
}

public class ProvaMenu
{ public static void main(String[] args)
  { MenuFrame m = new MenuFrame("Esempio menu");
    m.setSize(200, 200);
    m.setVisible(true);
  }
}
```

Mandando in esecuzione il programma si ottiene una finestra simile a quella di Fig. 3.18.

Figura 3.18. Una finestra dotata di menu.

3.10. Grafica

Il linguaggio Java consente in maniera relativamente semplice di fare uso di primitive grafiche di basso livello, quali tracciare linee, disegnare poligoni, eccetera. Prima di introdurre i meccanismi legati alla gestione della grafica a basso livello è però necessario comprendere il processo di visualizzazione dei componenti sullo schermo.

Un componente deve essere disegnato o ridisegnato sullo schermo per cause dipendenti dal sistema di visualizzazione o legate all'applicazione:

- *cause dipendenti dal sistema*: il sistema di visualizzazione si accorge che il componente deve essere disegnato o ridisegnato sullo schermo; per esempio, quando viene visualizzato per la prima volta (richiamato il metodo *setVisible()*), quando viene ridimensionato, oppure quando torna visibile dopo essere stato coperto da un'altra finestra;
- *cause legate all'applicazione*: l'applicazione decide che l'aspetto grafico di un componente deve essere alterato, per esempio per realizzare un'animazione.

Ogni componente ha un proprio ambiente grafico (spazio sul video assegnato al componente), sul quale avviene il disegno (o il nuovo disegno): questo è un oggetto appartenente a una sottoclasse della classe astratta *Graphics* (package *java.awt*) che viene associato al componente stesso (in dipendenza della classe a cui appartiene) dal sistema di visualizzazione.

Ogni qual volta un componente deve essere disegnato o ridisegnato, viene generato un evento che viene catturato dal thread *swing*, il quale esegue il metodo ***void paint(Graphics g)*** della classe *JComponent*. Nel caso in cui l'operazione sia comandata dal sistema di visualizzazione, questo genera direttamente l'evento anzidetto. Nel caso in cui l'operazione sia comandata dall'applicazione, quest'ultima deve invocare il metodo ***void repaint()***, che genera l'evento di cui sopra.

Tutti i componenti che abbiamo esaminato (tranne i contenitori di alto livello) appartengono a classi derivate dalla classe *JComponent*. Il metodo *paint()* della classe *JComponent* richiama i seguenti tre metodi:

public void paintComponent (Graphics *g* ***)***
disegna il componente vero e proprio;

public void paintBorder (Graphics *g* ***)***
disegna il bordo del componente;

public void paintChildren (Graphics *g* ***)***
disegna i componenti contenuti nel componente che viene ridisegnato.

Quando si effettuano disegni su un componente (appartenente alla classe *JComponent*) occorre definire per quel componente una nuova classe derivata, e in questa ridefinire il metodo *paint()*: in realtà, è sufficiente ridefinire solo il metodo *paintComponent()*, conservando il comportamento predefinito per quanto riguarda la gestione del bordo e dei componenti figli. Per esempio, con riferimento alla classe *JPanel* la ridefinizione avviene quindi nel seguente modo:

```
class PP extends JPanel
{  PP()
   {  // istruzioni
   }
```

```
  public void paintComponent(Graphics g)
  {  super.paintComponent(g);
     // altre eventuali istruzioni
     // che disegnano
  }
}
```

Il processo di disegno di un componente dipende anche dalla sua *opacità*. In un componente *opaco* vengono ridipinti tutti i pixel dell'area che occupa, mentre in un componente non opaco vengono ridipinti solo una parte di pixel, lasciando intravedere i pixel sottostanti. Se un componente è opaco il metodo *paintComponent()*, prima di disegnare il componete vero e proprio, ridipinge tutta l'area rettangolare riempiendola con il colore di *background* del componente (per questo motivo è opportuno che in un componente la ridefinizione del metodo *paintComponent()* richiami il metodo *paintComponent()* della superclasse). I metodi della classe *JComponent* ***void setOpaque(boolean b)*** e ***boolean isOpaque()*** possono essere usati rispettivamente per impostare e per conoscere e l'opacità di un componente.

3.10.1. Classe *Graphics*

In un oggetto grafico associato a un componente, i punti dello spazio sono identificati mediante assi cartesiani. L'origine è nel punto in alto a sinistra, l'asse *x* va da sinistra verso destra e l'asse *y* dall'alto verso il basso (Fig. 3.19). Le coordinate sono espresse in pixel.

Per impostare la fonte corrente si usa il metodo ***void setFont(Font f)***, mentre per il colore corrente si adopera il metodo ***void setColor(Color c)***.

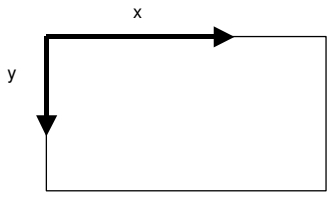


Figura 3.19. Il sistema di assi associato ad un componente.

Alcuni metodi della classe *Graphics* utili per disegnare sono i seguenti (le linee sono spesse un pixel):

public void drawString (String** str **, int** x **, int** y **)
scrive la stringa a partire dal punto *x*,*y*;

public void drawLine (int** x1 **, int** y1 **, int** x2 **, int** y2 **)
disegna una segmento tra i punti *x1*, *y1* e *x2*, *y2*;

public void drawRect (int** x **, int** y **, int** width **, int** height **)
disegna un rettangolo, con *x* e *y* coordinate del punto in alto a sinistra, di data larghezza (verso destra) e di data altezza (verso il basso);

public void fillRect (int** x **, int** y **, int** width **, int** height **)
come sopra, ma ne colora l'interno;

public void drawOval (int** x **, int** y **, int** width **, int** height **)
disegna un ovale, con *x* e *y* coordinate del punto in alto a sinistra del rettangolo che racchiude l'ovale, di data larghezza (verso destra) e di data altezza (verso il basso);

public void fillOval (int** x **, int** y **, int** width **, int** height **)
come sopra, ma ne colora l'interno.

Figura 3.20. Disegni in un pannello.

Come primo esempio, riportiamo un semplice programma che disegna un rettangolo, una stringa, e due cerchi in un pannello inserito in una

finestra. Il primo dei due cerchi ha colore rosso ed è opaco, il secondo ha colore blu ed è semitrasparente. Eseguendo il programma viene visualizzata una finestra simile a quella di Fig. 3.20.

```
import javax.swing.*;
import java.awt.*;
class PP extends JPanel
{ private Font f = new Font("Arial", Font.BOLD, 18);
  private Color semiTrasp = new Color(0, 0, 255, 128);
  PP()
  { setBackground(Color.yellow); }
  public void paintComponent(Graphics g)
  { super.paintComponent(g);
    g.setColor(Color.green);
    g.fillRect(40, 80, 180, 10);
    g.setColor(Color.red);
    g.setFont(f);
    g.drawString("Arial Bold 18", 80, 50);
    g.fillOval(60, 60, 50, 50);
    g.setColor(semiTrasp);
    g.fillOval(140, 60, 50, 50);
  }
}

class Fram extends JFrame
{ Fram(String s)
  { super(s);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    PP pan = new PP(); getContentPane().add(pan);
  }
}

public class ProvaGraph
{ public static void main (String[] args)
  { Fram fi = new Fram("Ciao");
    fi.setSize(250, 150);
    fi.setVisible(true);
  }
}
```

Come altro esempio, riportiamo un programma che definisce una classe *Etic*, sottoclasse di *JLabel*, il cui metodo *paintComponent()* disegna una

serie di cerchi concentrici di colore rosso e la stringa "*Il Centro*" in bianco come mostrato in Fig. 3.21 (viene anche visualizzato il testo dell'etichetta, comportamento ereditato dalla superclasse).

Figura 3.21. Etichetta con disegno.

```
import javax.swing.*;
import java.awt.*;
class Etic extends JLabel
{  Etic(String s)
   {  super(s);
      setOpaque(true);
      setBackground(Color.green);
   }
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      int h = getHeight(); int w = getWidth();
      int min;
      if (h < w) min = h; else min = w;
      g.setColor(Color.red);
      for(int r = 1; r < min/3; r+=4)
        g.drawOval(w/2-r, h/2-r, 2*r, 2*r);
      g.setFont
         (new Font("Arial", Font.BOLD|Font.ITALIC, 18));
      g.setColor(Color.white);
      g.drawString("Il Centro", w/2, h/2);
   }
}
```

```
class FinGrafica extends JFrame
{ private Container c;
  FinGrafica(String s)
  { super(s);
    Etic et = new Etic("Etichetta");
    getContentPane().add(et);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  }
}

public class ProvaL
{ public static void main(String[] args)
  { FinGrafica fi = new FinGrafica("Grafica");
    fi.setSize(300, 200);
    fi.setVisible(true);
  }
}
```

Come ultimo esempio, riportiamo un programma che definisce una classe *JP*, sottoclasse di *JPanel*, con quattro pulsanti di tipo *CheckBox* relativi alle quattro componenti di un colore, *red*, *green*, *blue* e *opacità* (semiopacità o opacità completa), e un rettangolo riempito col colore risultante. La classe *JP* contiene una classe ascoltatrice degli eventi prodotti da tali pulsanti, e ogni evento determina il valore della corrispondente componente colore e il richiamo del metodo *repaint()*. Il pannello viene inserito in una finestra (Fig. 3.22).

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class JP extends JPanel
{ private float cr = 0, cg = 0, cb = 0, co = 0.5F;
  private JCheckBox rr = new JCheckBox("Rosso");
  private JCheckBox gg = new JCheckBox("Verde");
  private JCheckBox bb = new JCheckBox("Blu");
  private JCheckBox oo = new JCheckBox("Opaco");
  public class AscoltatorePulsanti
               implements ItemListener
  { public void itemStateChanged(ItemEvent e)
    { JCheckBox ch = (JCheckBox)e.getSource();
      if (ch == rr)
      { if (e.getStateChange() == ItemEvent.SELECTED)
```

```
                cr = 1; else cr = 0;}
        else if (ch == gg)
        { if (e.getStateChange() == ItemEvent.SELECTED)
                cg = 1; else cg = 0;}
        else if (ch == bb)
        { if (e.getStateChange() == ItemEvent.SELECTED)
                cb = 1; else cb = 0;}
        else if (ch == oo)
        { if (e.getStateChange() == ItemEvent.SELECTED)
                co = 1; else co = 0.5F;  }
        repaint();
      }
    }
    JP()
    { setBackground(Color.white);
      add(rr); add(gg); add(bb); add(oo);
      AscoltatorePulsanti ap =
                          new AscoltatorePulsanti();
      rr.addItemListener(ap); gg.addItemListener(ap);
      bb.addItemListener(ap); oo.addItemListener(ap);
    }
    public void paintComponent(Graphics g)
    { super.paintComponent(g);
      Color cc = new Color(cr, cg, cb, co);
      g.setColor(cc); g.fillRect(50, 80, 240, 60);
    }
}

class JFF extends JFrame
{ JFF(String s)
    { super(s);
      setDefaultCloseOperation(DISPOSE_ON_CLOSE);
      JP pan = new JP(); getContentPane().add(pan);
    }
}

public class ProvaRGB
{ public static void main(String[] args)
    { JFF fi = new JFF("RGB");
      fi.setSize(350, 200);
      fi.setVisible(true);
    }
}
```

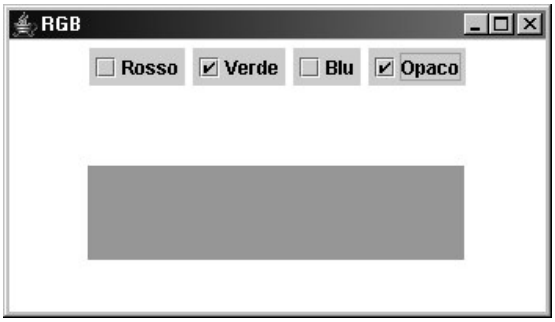


Figura 3.22. Colori RGB.

3.10.2. Immagini

In Java un'immagine è rappresentata da una istanza della classe *Image*. Il modo più semplice di creare istanze della classe *Image* è attraverso la classe *ImageIcon*. Oggetti di tipo *ImageIcon* possono essere creati mediante i seguenti costruttori:

public ImageIcon (String** nf **)
public ImageIcon (URL** loc **)
creano una istanza di *ImageIcon* a partire dai dati rispettivamente contenuti nel file il cui percorso è specificato dall'argomento *nf* o disponibili all'URL *loc* (vengono supportati i formati *GIF*, *JPG*, e *PNG*).

Entrambi i costruttori caricano completamente i dati dell'immagine prima di terminare.

Da una istanza di *ImageIcon*, invocando il metodo ***getImage()***, si ottiene una istanza di *Image*. Per esempio si può scrivere:

```
ImageIcon ic = new ImageIcon("immagini/prova.jpg");
Image im = ic.getImage();
```

Per conoscere le dimensioni di un'immagine sono disponibili due metodi della classe *Image*:

public int getWidth (null)
public int getHeight (null)
restituiscono rispettivamente la larghezza e l'altezza dell'immagine espresse in pixel.

Per visualizzare una immagine sono disponibili i seguenti metodi della classe *Graphics*:

public boolean drawImage (Image img , int x , int y , null)
public boolean drawImage (Image img , int x , int y ,
int width , int height , null)
img è l'immagine da disegnare, *x* e *y* identificano la posizione del punto in alto a sinistra dell'immagine all'interno del componente, e *width* ed *heigth* le dimensioni (se tali valori non vengono esplicitamente indicati, sono costituiti dai valori restituiti dai metodi *getWidth()* e *getHeight()*); il valore di ritorno è normalmente *true* (*false* se l'immagine non è completamente caricata).

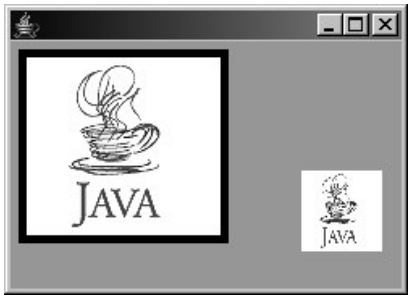


Figura 3.23. Una finestra contenente immagini.

I metodi precedenti *drawImage()*, *getWidth()* e *getHeight()* prevedono in realtà un argomento formale di tipo *ImageObserver* e non il simbolo *null*, come sarà illustrato nel Capitolo 4.

A titolo di esempio riportiamo il codice di un'applicazione che crea un'immagine (contenuta nel file *imgs/java.gif*), quindi la disegna due volte su uno sfondo verde: la prima volta con le sue dimensioni naturali e con intorno un bordo nero, la seconda volta con una dimensione di 50x50 pixel

(Fig. 3.23). Per eseguire il programma, l'interprete deve essere lanciato a partire dalla cartella che contiene la cartella *imgs*.

```
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;
class ImageFrame extends JFrame
{ private Image img;
  private ImageIcon ic;
  private int larghezza, altezza;
  class MioPanel extends JPanel
  { public void paintComponent(Graphics g)
    { super.paintComponent(g);
      g.fillRect(5, 5, larghezza+10, altezza+10);
      g.drawImage(img, 10, 10, null);
      g.drawImage(img, 180, 80, 50, 50, null);
    }
  }
  private MioPanel p;
  ImageFrame()
  { setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    ic = new ImageIcon("imgs/java.gif");
    img = ic.getImage();
    larghezza = img.getWidth(null);
    altezza = img.getHeight(null);
    p = new MioPanel();
    p.setBackground(Color.green);
    getContentPane().add(p);
  }
}

public class ProvaImage
{ public static void main(String[] args)
  { ImageFrame fr = new ImageFrame();
    fr.setSize(200, 200);
    fr.setVisible(true);
  }
}
```

3.10.3. Animazioni

Un'applicazione che vuole realizzare un'animazione deve disegnare ad

intervalli periodici i “fotogrammi” che la compongono. Questo richiede la presenza di un thread ciclico che, per tutta la durata dell’animazione, compie le seguenti azioni:

1. genera la prossima immagine dell’animazione modificando la struttura dati che la rappresenta;
2. comanda al thread *swing* di disegnare la nuova immagine nella zona dedicata all’animazione;
3. dorme per il tempo che intercorre tra due fotogrammi.

Occorre osservare che possono verificarsi accessi concorrenti alla struttura dati che descrive l’immagine: il thread *swing* deve leggerne il contenuto per eseguire l’operazione di ridisegno, mentre il thread ciclico deve modificarla per preparare l’immagine successiva. Una prima soluzione consiste nell’usare il costrutto *syncronized()*, facendo eseguire le due azioni in mutua esclusione. Il seguente esempio utilizza questa tecnica, utilizzando il thread *mioThread* per ottenere la data e il thread *swing* per visualizzarla, sincronizzando su uno stesso oggetto *o* le azioni di questi due thread. In esecuzione, il programma produce una fiestra del tipo di quella riportata in Fig. 3.24.

```
import javax.swing.*;
import java.awt.*;
import java.util.Date;
class ThreadFrame extends JFrame
{ private boolean b = true;
  private Date  miaData = new Date();
  private Font f = new Font("Arial", Font.BOLD, 16);
  private Object o = new Object();
  class ClaThread extends Thread
  { public void run()
    { while (b)
      { synchronized(o) // prelievo della data
        { miaData = new Date();  }
        repaint();
        try  { mioThread.sleep(500);  }
        catch (InterruptedException e) { }
      }
    }
  }
```

```
  private ClaThread mioThread = new ClaThread();
  class MioPanel extends JPanel
  { public MioPanel()
    { setBackground(Color.yellow); }
    public void paintComponent(Graphics g)
    { super.paintComponent(g); g.setFont(f);
      synchronized (o)  // visualizzazione della data
      { g.drawString(miaData.toString(), 30, 60);   }
    }
  }
  private MioPanel p = new MioPanel();
  ThreadFrame()
  { setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    getContentPane().add(p);
  }
  void inizio()
  { mioThread.start();  }
  void fine()
  { b = false; }
}

public class Prova
{ public static void main(String[] args)
  { ThreadFrame ft = new ThreadFrame();
    ft.setSize(300, 150);
    ft.setVisible(true);
    ft.inizio();
    try { Thread.sleep(50000); }
    catch(InterruptedException e) {}
    ft.fine();
  }
}
```

Notare che, con riferimento al programma di cui alla Fig. 3.22, le due azioni di i) determinare le componenti di un nuovo colore e ii) visualizzare un rettangolo con il nuovo colore, vengono entrambe eseguite dallo stesso thread *swing* e non danno quindi luogo a malfunzionamenti.

La seconda soluzione consiste nel far eseguire le due azioni sempre allo stesso thread (thread *swing*), anche quando l'animazione è comandata dal tempo e non da un evento. A tale proposito si utilizza un oggetto della classe *Timer* (package *javax.swing*), che ad intervalli periodici genera un evento gestito da un ascoltatore.

Figura 3.24. Prelievo e visualizzazione della data.

La classe *Timer* dispone del seguente costruttore:

public Timer (int** delay **, ActionListener** listener **)
crea un oggetto *Timer* che richiama il metodo *actionPerformed()* dell'oggetto *listener* ogni *delay* millisecondi.

Dopo aver creato un oggetto *Timer* è necessario attivarlo invocando il suo metodo *start()*, ed eventualmente disattivarlo invocando il suo metodo *stop()*.

Quindi, per eseguire un'animazione, è sufficiente inserire il codice che modifica la struttura dati che descrive l'immagine all'interno del metodo *actionPerformed()* dell'oggetto *listener* passato al *Timer*: poiché tale codice viene eseguito per mezzo del thread *swing*, che è responsabile anche dell'operazione di ridisegno, non possono verificarsi problemi legati ad un accesso concorrente alla struttura dati.

Come primo esempio, riportiamo un programma che mostra la rotazione di un segmento intorno a un estremo, simulando la lancetta di un orologio: la rotazione inizia e termina con la chiamata di metodi che attivano e disattivano il timer, rispettivamente (Fig. 3.25).

```
import java.awt.*;import java.awt.event.*;
import javax.swing.*;
class Fin extends JFrame
{ class Pan extends JPanel
```

```
  { private double angolo = 0, delta = 0.05, dx, dy;
    void ruota()
    { angolo +=delta;
      dx = 50*Math.cos(angolo);
      dy = 50*Math.sin(angolo);
    }
    public void paintComponent(Graphics g)
    { super.paintComponent(g);
      setBackground(Color.green);
      g.setColor(Color.yellow);
      g.fillOval(50, 50, 100, 100);
      g.setColor(Color.red);
      g.drawLine(100,100,(int)(100+dx),(int)(100+dy));
    }
  }
  private Pan p;
  public class AscoltatoreT implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { p.ruota(); p.repaint();  }
  }
  private int WAIT = 30; private AscoltatoreT gg;
  private Timer t;
  Fin()
  { setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    p = new Pan(); getContentPane().add(p);
    gg = new AscoltatoreT(); t = new Timer(WAIT, gg);
  }
  void partenza()
  { t.start(); }
  void arresto()
  { t.stop();  }
}

public class ProvaRotazione
{ public static void main(String[] args)
  { Fin ff = new Fin();
    ff.setSize(200, 220);ff.setVisible(true);
    ff.partenza();
    try { Thread.sleep(10000); }
    catch (InterruptedException e) { }
    ff.arresto();
  }
}
```

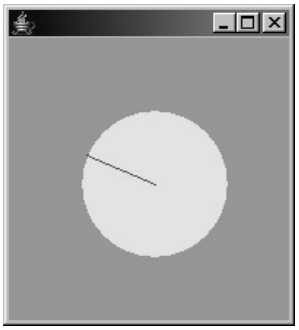


Figura 3.24. Simulazione della lancetta di un orologio.

Come ultimo esempio riportiamo il codice di un programma che visualizza dei fiocchi di neve che cadono su uno sfondo nero, utilizzando un pannello inserito in una finestra (Fig. 3.25). Per ognuna delle *h* righe che compongono il pannello c'è un fiocco di neve. La posizione dei fiocchi di neve è memorizzata nell'array *fi[]*, in particolare *fi[i]* contiene l'ascissa del fiocco di neve sulla riga *i*-esima. Il metodo *paintComponent()* del pannello disegna un asterisco in ognuna delle righe del pannello, mentre il metodo *trasla()* aggiorna lo stato della struttura dati spostando in *fi[i]* il contenuto di *fi[i-1]* e inserendo un valore casuale in *fi[0]*. Il metodo *inizializza()* del pannello, oltre a determinare le dimensioni *h* e *w* del componente, riempie il vettore *fi[]* di valori casuali compresi tra *0* e *w*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Neve extends JFrame
{  class NevePanel extends JPanel
   { private int h, w;
     private int fi[];
     void inizializza()
     {  w = getWidth();
        h = getHeight();
        fi = new int[h];
        for(int i=0; i<h; i++)
          fi[i] = (int)(Math.random()*w);
```

```
    }
    void trasla()
    { for(int i=h-1; i>0; i--)
        fi[i] = fi[i-1];
      fi[0] = (int)(Math.random()*w);
    }
    public void paintComponent(Graphics g)
    { super.paintComponent(g);
      for(int i=0; i<h; i++)
        g.drawString("*", fi[i], i);
    }
  }
  private final int WAIT = 30;
  private NevePanel p = new NevePanel();
  private Timer t;
  private Traslatore tras;
  public class Traslatore implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { p.trasla();
      p.repaint();
    }
  }
  Neve(int x, int y)
  { super("Neve");
    setSize(x, y);
    setResizable(false);
    p.setBackground(Color.black);
    p.setForeground(Color.white);
    p.setOpaque(true);
    getContentPane().add(p);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    tras = new Traslatore();
    t = new Timer(WAIT, tras);
  }
  void vai()
  { p.inizializza();
    t.start();
  }
  void ferma()
  { t.stop();
  }
}
```

```
public class ProvaNeve
{ public static void main(String[] args)
  { Neve neve = new Neve(400, 400);
    neve.setVisible(true);
    neve.vai();
    try { Thread.sleep(10000); }
    catch (InterruptedException e) { }
    neve.ferma();
  }
}
```

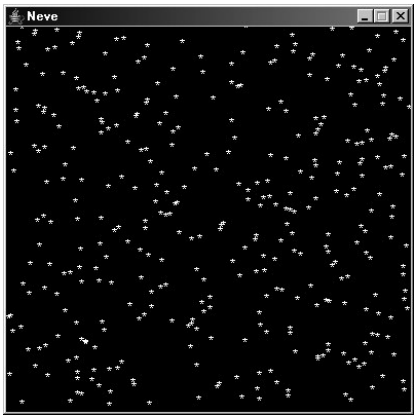


Figura 3.25. Un istante dell'animazione "neve".

3.11. Suoni

Le applicazioni Java sono in grado di generare suoni e riprodurre file audio. Il suono più semplice che si può generare è un *beep*. Per fare questo occorre prima procurarsi un riferimento all'ambiente grafico invocando il seguente metodo statico della classe *Toolkit*:

public static Toolkit getDefaultToolkit ()

Successivamente, sull'oggetto *Toolkit*, può essere invocato il metodo ***void beep()***. Pertanto, per generare un beep, si può scrivere:

```
Toolkit.getDefaultToolkit().beep();
```

Una applicazione può anche riprodurre un file audio attraverso l'interfaccia *AudioClip* (package *java.applet*). Per creare un oggetto di tipo *AudioClip* si può usare il seguente metodo statico della classe *JApplet*:

public static AudioClip newAudioClip (URL u)
l'argomento *u* corrisponde all'URL da cui caricare il file.

Il formato audio sicuramente supportato è quello AU (8 bit, 8000 Hertz, un canale), sviluppato dalla SUN. Generalmente sono supportati anche altri formati, come *WAVE*, *AIFF*, *MIDI* e *RMF* (in ambiente Windows le estensioni dei file sono rispettivamente *.wav*, *.aif*,. *mid* e *.rmf*).

Per iniziare ed arrestare la riproduzione del file audio è sufficiente invocare sull'oggetto *AudioClip* i metodi ***void play()*** e ***void stop()*** rispettivamente. E' inoltre disponibile un terzo metodo, ***void loop()***, che riproduce il file audio a ciclo continuo. I metodi *play()* e *loop()* quando vengono invocati ritornano immediatamente il controllo al chiamante e la riproduzione viene eseguita attraverso un thread separato (tale thread impedisce all'applicazione di terminare quando si conclude l'esecuzione del metodo *main()*).

Come esempio riportiamo un programma che prima genera un *beep*, quindi visualizza una finestra dotata di tre bottoni che consentono di interagire con un oggetto *AudioClip* (Fig. 3.26). Il file audio da riprodurre, *gong.au*, è contenuto nella cartella *audio*, che a sua volta deve essere contenuta nella cartella in cui viene eseguito il comando *java*.

```
import java.awt.*;
import java.net.*;
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
class SoundFrame extends JFrame
{ private AudioClip audio;
  private JButton play = new JButton("Play");
  private JButton stop = new JButton("Stop");
```

```
  private JButton loop = new JButton("Loop");

  private class Ascoltatore implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { if(e.getSource().equals(play))
        audio.play();
      else if(e.getSource().equals(stop))
        audio.stop();
      else if(e.getSource().equals(loop))
        audio.loop();
    }
  }
  public SoundFrame()
  { super("Suoni");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Toolkit.getDefaultToolkit().beep();
    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    try
    { audio = JApplet.newAudioClip
        (new URL("file:audio/gong.au"));
      c.add(play);
      c.add(stop);
      c.add(loop);
      Ascoltatore as = new Ascoltatore();
      play.addActionListener(as);
      stop.addActionListener(as);
      loop.addActionListener(as);
    }
    catch(MalformedURLException mue)
    { c.add(new JLabel("URL errato")); }
  }
}

public class ProvaSound
{ public static void main(String[] args)
  { SoundFrame fr = new SoundFrame();
    fr.setSize(200, 100);
    fr.setVisible(true);
  }
}
```

Si noti che l'azione eseguita in risposta alla pressione del bottone di chiusura della finestra è *EXIT_ON_CLOSE*, altrimenti il thread associato alla riproduzione del file audio impedirebbe all'applicazione di terminare.

Figura 3.25. Interfaccia per provare i suoni.

4. Applet e Servlet

4.1. Applet

Un *applet* è un programma Java che viene eseguito all'interno di un browser. In particolare, una porzione di una pagina HTML visualizzata dal browser viene usata per inserirvi un applet, che può visualizzare immagini, compiere animazioni, riprodurre suoni, eccetera.

Un applet è composto da una classe principale che estende la classe *JApplet* (package *javax.swing*) e da eventuali altre classi ausiliarie. A differenza delle applicazioni viste finora, la classe principale di un applet non è dotata di un metodo *main()*: il ciclo di vita dell'applet è regolato dal browser che, dopo aver caricato la classe principale dalla rete ed averne creato una istanza, chiama metodi opportuni al verificarsi di determinate condizioni.

Per inserire un applet in una pagina HTML si deve usare il tag <*applet*>. Gli attributi sono *code*, *height* e *width*, che specificano rispettivamente il nome della classe che implementa l'applet, l'altezza e la larghezza (in pixel) della porzione di pagina assegnata all'applet stesso. Per esempio, in una pagina HTML si può avere:

```
<applet code="NomeApplet.class" width=200 height=100>
</applet>
```

Insieme all'ambiente Java viene fornito il programma *appletviewer*, che consente di eseguire un applet al di fuori del contesto di un browser, rendendo più semplice la fase di sviluppo.

Tale programma riconosce la presenza del tag <*applet*> nel file che gli viene passato come parametro ed esegue l'applet in una finestra che ha le dimensioni dell'area specificata (il contenuto HTML della pagina che include l'applet non viene visualizzato).

4.1.1. Ciclo di vita

I seguenti metodi vengono richiamati automaticamente dal browser all'accadere delle condizioni specificate: essi sono definiti nella classe *JApplet* (non compiono alcuna azione) e possono essere ridefiniti per specializzare il comportamento di uno specifico applet:

public void init ()
viene richiamato subito dopo che l'applet è stato caricato (viene invocato almeno una volta, prima della prima esecuzione del metodo *start()*);

public void start ()
viene richiamato dopo che l'applet è stato caricato (dopo il metodo *init()*) e ogni volta che la pagina viene rivisitata;

public void stop ()
viene richiamato (per far cessare l'esecuzione dell'applet) ogni volta che la pagina che lo contiene viene lasciata o viene chiuso il browser;

public void destroy ()
viene richiamato quando viene chiuso il browser e l'applet viene scaricato (contiene generalmente codice di pulizia).

Il codice di inizializzazione, più che nel costruttore, è opportuno inserirlo all'interno del metodo *init()*, in quanto non è garantito che l'ambiente di esecuzione dell'applet sia già completamente specificato quando viene richiamato il costruttore stesso.

Quando viene caricata una pagina HTML contenente il tag <*applet*>, viene trasferita e caricata la classe indicata, viene generata un'istanza dell'applet e infine vengono richiamati metodi *init()* e *start()*. Se l'utente esce dalla pagina e rientra nella pagina vengono richiamati i metodi *stop()* e *start()*, rispettivamente. Se l'utente chiede al browser di aggiornare la pagina, prima vengono richiamati i metodi *stop()* e *destroy()*, quindi viene

caricato di nuovo l'applet e creata una nuova istanza, e infine vengono richiamati i metodi *init()* e *start()* (l'aggiornamento non comporta automaticamente anche il ritrasferimento dell'applet). All'uscita dal browser vengono richiamati i metodi *stop()* e *destroy()*.

In realtà, non tutti i browser si comportano allo stesso modo, e alcuni di loro eseguono delle azioni aggiuntive rispetto a quanto illustrato in precedenza. Per esempio, quando si esce dalla pagina viene anche richiamato il metodo *destroy()* e quando si rientra nella pagina viene anzitutto creato un nuovo oggetto e richiamato il metodo *init()*.

La classe *JApplet* è un contenitore di alto livello: è quindi in grado di contenere tutti i componenti visti nel Capitolo 3. Ogni volta che occorre disegnare o ridisegnare l'applet (per esempio, al rientro nella finestra di visibilità del browser o invocando il metodo *repaint()*) viene eseguito il metodo *paint()* che ridisegna il contenitore e richiama il metodo *paint()* dei componenti figli.

Sempre come detto nel Capitolo 3, se tutti i figli appartengono alla classe *JComponen*t, quando per un figlio si rende necessario ridefinire il metodo *paint()*, basta ridefinire per questo il metodo *paintComponent()*.

Un applet può visualizzare informazioni di stato sulla barra posta nella parte bassa del browser invocando il metodo ***void showStatus(String** info)*.

Come primo esempio, riportiamo il codice di un applet elementare che visualizza il numero di creazioni di oggetti della classe, di invocazioni dei metodi *init()*, *start()*, *stop()* e *destroy()* e il numero delle operazioni di ridisegno. Il codice è costituito da una classe *MioApplet* contenente un'altra classe *MioPane*l:

```
// file MioApplet.java
import java.awt.*;
import javax.swing.*;
public class MioApplet extends JApplet
{ private static int numeroOggetti = 0;
  private static int inizializzazioni = 0;
  private static int partenze = 0;
  private static int arresti = 0;
  private static int distruzioni = 0;
  private static int disegni = 0;
  public class MioPanel extends JPanel
  { MioPanel()
    { setBackground(Color.yellowW);
```

```
      setForeground(Color.blue);
    }
    public void paintComponent(Graphics g)
    { super.paintComponent(g);
      disegni ++;
      String s1, s2, s3, s4;
      s1 = "Oggetti creati: " + numeroOggetti;
      s2 = "Inizializzazioni: " + inizializzazioni +
           "; Partenze: " + partenze;
      s3 = "Arresti: " + arresti +
           "; Distruzioni: " + distruzioni;
      s4 = "Disegni: " + disegni;
      g.drawString(s1, 5, 20);
      g.drawString(s2, 5, 40);
      g.drawString(s3, 5, 60);
      g.drawString(s4, 5, 80);
    }
  }
  public MioApplet()
  { numeroOggetti++;  }
  public void init()
  { getContentPane().add(new MioPanel());
    inizializzazioni++;
  }
  public void start()
  { partenze++;}
  public void stop()
  { arresti++; }
  public void destroy()
  { distruzioni++;  }
}
```

Tale file va compilato, ottenendo i due file oggetto *MioApplet$MioPanel.class* e *MioApplet.class*.

Per esaminare il comportamento dell'applet usiamo i seguenti due file (inseriti nella stessa cartella contenente i due file oggetto precedenti) che contengono codice HTML (documenti o pagine HTML):

Applet.html

```
<html>
<title> File Applet.html </title>
<body>
```

```
<h1> Esempio </h1>
<br><br>
<applet code="MioApplet.class" width=500
height=300> </applet>
<br><br>
<A href = "AltraPagina.html"> Altra pagina </A>
</body>
</html>
```

AltraPagina.html

```
<html>
<title> File AltraPagina.html</title>
<body>
<h1> Altra pagina </h1>
</body>
</html>
```

Figura 4.1. Un semplice applet.

Aprendo con un browser il documento *Applet.html* la pagina visualizzata ha l'aspetto mostrato in Figura 4.1.

Il file HTML contenente il tag <*applet*> costituisce il *documento* o la *pagina tag*, mentre il file indicato nel tag (che ha sempre estensione *.class*) costituisce il *file principale* dell'applet: questo può anche far parte di un package, e in questo caso il tag specifica con la notazione a punto il nome del package e il nome del file principale.

4.1.2. Immagini

Come visto nel Capitolo 3, in Java un'immagine è rappresentata da una istanza della classe *Image*. Un applet può creare un'immagine invocando direttamente il metodo *getImage()* in una delle due seguenti forme:

```
public Image getImage ( URL url )
public Image getImage ( URL base , String perc )
```

Nella prima forma l'URL specifica completamente la locazione del file che contiene l'immagine, mentre nella seconda forma la locazione del file è espressa come un percorso relativo *perc* a partire dall'URL *base*. I formati di memorizzazione delle immagini supportati dalla classe *Image* sono *JPG*, *GIF* e *PNG*.

Capita spesso che i file delle immagini che l'applet vuole visualizzare siano memorizzati sullo stesso server in cui risiede l'applet. In questo caso l'applet può usare i metodi *getCodeBase()* e *getDocumentBase()* per conoscere l'URL a cui appartiene, rispettivamente, o la cartella che contiene il file principale dell'applet (o il package in cui tale file si trova) o la cartella che contiene la pagina *tag* dell'applet:

```
public URL getCodeBase ()
```
restituisce l'URL della cartella che contiene il file principale dell'applet (o il package di cui fa parte)

```
public URL getDocumentBase ()
```
restituisce l'URL della cartella che contiene il documento tag dell'applet.

Per esempio, se il file *img.gif* è contenuto nella cartella *imgs*, a sua volta contenuta, rispettivamente, o nella stessa cartella in cui si trova il

documento tag dell'applet o nella stessa cartella in cui si trova il file principale dell'applet, l'immagine contenuta in tale file si ottiene nel seguente modo:

```
Image i = getImage(getDocumentBase(), "imgs/img.gif"));
Image i = getImage(getCodeBase(), "imgs/img.gif"));
```

I metodi *getImage()* restituiscono immediatamente un oggetto di tipo *Image*, senza attendere che i dati dell'immagine vengano caricati. I dati cominciano ad essere caricati solo quando l'applet disegna l'immagine sullo schermo.

Per disegnare l'immagine l'applet può usare, come visto in forma semplificata nel Capitolo 3, uno dei metodi sovrapposti *drawImage()* della classe *Graphics*, per esempio il seguente:

public boolean drawImage (Image** img **, int** x **, int** y **,
ImageObserver** observer **)

img è l'immagine da disegnare, *x* e *y* identificano la posizione dell'angolo in alto a sinistra dell'immagine all'interno del componente dove essa deve essere disegnata, e *observer* è un oggetto che viene usato per informare dello stato di caricamento dei dati dell'immagine.

Quando l'applet invoca il metodo *drawImage()*, l'immagine viene disegnata sullo schermo utilizzando i dati caricati fino a tale istante. Il metodo ritorna immediatamente anche nel caso in cui l'immagine non sia stata completamente caricata e restituisce *true* se il caricamento è stato completato, *false* altrimenti: in quest'ultimo caso, l'oggetto *observer* viene notificato man mano che i dati dell'immagine vengono ricevuti.

Per conoscere le dimensioni di un'immagine sono disponibili i due seguenti metodi della classe *Image* (introdotti in forma semplificata nel Capitolo 3):

public int getWidth (ImageObserver** o **)
public int getHeight (ImageObserver** o **)

restituiscono rispettivamente la larghezza e l'altezza dell'immagine espresse in pixel; nel caso in cui l'immagine non sia stata ancora completamente caricata viene restituito il valore *-1*; successivamente

l'*ImageObserver o* viene notificato quando l'informazione diventa disponibile.

L'interfaccia *ImageObserver*, usata per il processo di notifica, comprende un solo metodo che viene invocato ogni volta che arrivano nuovi dati dell'immagine:

```
public boolean imageUpdate ( Image img , int infoflags ,
                             int x , int y, int width , int height )
```

img è un riferimento all'immagine sotto osservazione e *infoflags* indica quali informazioni siano disponibili; il valore di *infoflags* è pari all'OR bit a bit di costanti (potenze di due) dell'interfaccia *ImageObserver*, tra cui:

WIDTH
la larghezza dell'immagine è disponibile e il suo valore è pari a *width*;

HEIGHT
l'altezza dell'immagine è disponibile e il suo valore è pari a *height*;

SOMEBITS
altri pixel dell'immagine sono disponibili (*x*, *y*, *width* e *height* indicano la regione a cui tali pixel appartengono);

ALLBITS
l'immagine è completa e può essere disegnata nella sua forma finale (il valore di *x, y, width* e *height* deve essere ignorato);

ERROR
si è verificata una situazione di errore e il caricamento dei dati viene abortito;

ABORT
il processo di caricamento è stato abortito; se il bit *ERROR* non è settato, il processo riparte quando si tenta nuovamente di disegnare l'immagine o quando si invocano i metodi *getWidth()* e *getHeight()*.

L'implementazione del metodo *imageUpdate()* deve restituire al chiamante il valore *true* se le informazioni necessarie sono state acquisite, *false* quando si vogliono ricevere altre notifiche. La classe *JComponent* implementa l'interfaccia *ImageObserver* ridisegnando il componente.

A titolo esemplificativo, riportiamo come applet un programma già visto nel Capitolo 3.

```
import java.awt.*;
import javax.swing.*;
import java.awt.image.*;
public class ImageApplet extends JApplet
{ private Image img;
  private int larghezza, altezza;
  private Osservatore oss = new Osservatore();
  public class MioPanel extends JPanel
  { public MioPanel()
    { setBackground(Color.yellow);
    }
    public void paintComponent(Graphics g)
    { super.paintComponent(g);
      g.fillRect(5, 5, larghezza+10, altezza+10);
      g.drawImage(img, 10, 10, oss);
      g.drawImage(img, 180, 80, 50,50, oss);
    }
  }
  private MioPanel p;
  public void init()
  { img = getImage(getCodeBase(), "imgs/java.gif");
    larghezza = img.getWidth(oss);
    altezza = img.getHeight(oss);
    p = new MioPanel();
    getContentPane().add(p);
  }
  private class Osservatore implements ImageObserver
  { public boolean imageUpdate(Image img,
                               int infoflags,
                               int x, int y,
                               int width, int height)
    { if((infoflags & WIDTH) != 0)
        larghezza = width;
      if((infoflags & HEIGHT) != 0)
        altezza = height;
      repaint();
      if((infoflags & (ALLBITS|ABORT)) != 0)
        return false;
      return true;
    }
  }
}
```

La classe interna *MioPanel* ridefinisce il metodo *paintComponent()* in modo tale da disegnare due volte l'immagine *img*, la prima volta usando le dimensioni reali dell'immagine, la seconda scalandola ad una dimensione di *50x50* pixel. La prima delle due immagini viene racchiusa in un bordo nero (il colore default per gli elementi in foreground) andando a disegnare un rettangolo leggermente più grande dell'immagine prima di disegnare quest'ultima sullo schermo. Quando viene invocato il metodo *init()* dell'applet inizia il processo di caricamento dell'immagine a causa dell'invocazione dei metodi *getWidth()* e *getHeight()* che cercano di determinare le dimensioni dell'immagine (memorizzate in *larghezza* e *altezza*). Ai metodi *getWidth()* e *getHeight()* è associato una istanza di *Osservatore*, *oss*, che implementa l'interfaccia *ImageObserver* e viene notificato mano a mano che nuove informazioni dell'immagine diventano disponibili. La stessa istanza di *Osservatore* viene impiegata anche nell'invocazione dei metodi *drawImage()*. Quando il metodo *imageUpdate()* viene invocato, oltre a richiedere una nuova operazione di ridisegno dell'intera applet attraverso l'invocazione del metodo *repaint()*, si analizza lo stato dei flag impostando il valore delle variabili *altezza* e *larghezza* quando opportuno.

4.1.3. Suoni

Gli applet supportano la riproduzione di semplici file audio, sicuramente quelli con estensione *au*.

Per avviare la riproduzione di un file audio, un applet può invocare uno dei due metodi *play()* della classe *JApplet*:

public void play (URL** url **)
riproduce il file audio identificato da *url*;

public void play (URL** base **, String** perc **)
riproduce il file audio individuato dal percorso relativo *perc* a partire dall'URL *base*.

Alternativamente, un applet può ottenere un oggetto di tipo *AudioClip* (che fa parte del package *java.applet*) attraverso i metodi della classe *JApplet* ***AudioClip getAudioClip(URL** url**)*** e ***AudioClip getAudioClip(URL***

base, ***String*** *perc)*. Come detto nel Capitolo 3, per iniziare ed arrestare la riproduzione del file audio è sufficiente invocare sull'oggetto *AudioClip* i metodi ***void play()***, ***void stop()*** e ***void loop()***, rispettivamente.

Analogamente a quanto visto nel Capitolo 3, riportiamo come applet un programma che utilizza i file audio *gong.au* e *treno.au*, contenuti nella cartella *audio* contenuta nella cartella dell'applet. L'applet implementa il metodo *stop()* allo scopo di arrestare la riproduzione del file *treno.au* nel caso in cui l'utente cambi pagina.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class SoundApplet extends JApplet
{ private AudioClip ac;
  private JButton play = new JButton("Play");
  private JButton stop = new JButton("Stop");
  private JButton loop = new JButton("Loop");
  public class Ascoltatore implements ActionListener
  { public void actionPerformed(ActionEvent e)
    { if(e.getSource().equals(play)) ac.play();
      if(e.getSource().equals(stop)) ac.stop();
      if(e.getSource().equals(loop)) ac.loop();
    }
  }
  public void init()
  { play(getCodeBase(), "audio/gong.au");
    ac = getAudioClip
           (getCodeBase(), "audio/treno.au");
    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    c.add(play);
    c.add(stop);
    c.add(loop);
    Ascoltatore aa = new Ascoltatore();
    play.addActionListener(aa);
    stop.addActionListener(aa);
    loop.addActionListener(aa);
  }
  public void stop()
  { ac.stop(); }
}
```

4.1.4. Documenti HTML

Un applet può chiedere al browser di visualizzare il contenuto di una pagina HTML. Per poter inoltrare la richiesta al browser, l'applet deve prima di tutto ottenere un riferimento al suo contesto di esecuzione: questo è costituito da una istanza della classe *AppletContext* e il riferimento è dato dal valore di ritorno del metodo ***getAppletContext()*** della classe *JApplet*. La richiesta al browser viene effettuata applicando all'istanza di *AppletContext* il seguente metodo:

public void showDocument (URL *url)*
la pagina indicata dall'URL viene visualizzata al posto della pagina corrente.

Riportiamo il codice di un applet che visualizza un'interfaccia grafica simile a quella di Figura 4.2. Quando viene digitato un URL e premuto il tasto di ritorno carrello, l'applet chiede al browser di visualizzare la pagina indicata.

Figura 4.2. Una pagina HTML che contiene l'applet *ShowApplet*.

```
import java.applet.*;
import javax.swing.*;
import java.net.*;
```

```
import java.awt.event.*;
import java.awt.*;
public class ShowApplet extends JApplet
{ private JLabel etic =
                      new JLabel("Inserisci un URL");
  private JTextField testo = new JTextField("");
  public void init()
  { getContentPane().setLayout(new GridLayout(2, 1));
    getContentPane().add(etic);
    getContentPane().add(testo);
    testo.addActionListener(new Ascoltatore());
  }
  public class Ascoltatore implements ActionListener
  { public void actionPerformed(ActionEvent e){
    String u = testo.getText();
    try
    { getAppletContext().showDocument(new URL(u));
    }
    catch (MalformedURLException mue)
    { showStatus("L'URL " + u + " non e' valido");   }
    }
  }
}
```

Esiste una ulteriore versione del metodo *showDocument()*:

public void showDocument (URL** url **, String** finestra **)
visualizza la pagina indicata da *url* in una nuova finestra; il secondo argomento specifica il nome della nuova finestra (un esempio di utilizzo è mostrato nel paragrafo 4.3).

4.1.5. Tag <applet>

Oltre agli attributi obbligatori *code*, *width* e *height* introdotti all'inizio del capitolo, il tag <*applet*> dispone di altre opzioni che consentono di specificare la posizione dell'applet rispetto al testo e la localizzazione della classe che lo implementa. L'attributo *align* indica l'allineamento dell'applet nella pagina: quando assume i valori *left* e *right* l'applet viene allineato rispettivamente a sinistra e a destra (sono possibili anche altri valori, essenzialmente quelli previsti per il tag <*img*>). Gli attributi *hspace*

e *vspace* regolano la dimensione dello spazio tra l'applet e il testo (il valore è espresso in pixel). L'attributo *codebase* specifica l'URL che contiene la classe principale dell'applet: se omesso, ha un valore predefinito dato dall'URL della cartella contenente il documento HTML con il tag <*applet*>. Quando specificato, il *codebase* può essere sia relativo che assoluto.

A titolo di esempio riportiamo il seguente frammento di codice HTML:

```
<applet code="pack.MioApplet.class" width=100
height=100 codebase="http://unhost/app/" hspace=20
vspace=20 align="left"> </applet>
```

Il tag precedente assegna all'applet una regione di 100x100 pixel, la allinea a sinistra e le lascia intorno uno spazio vuoto di 20 pixel. La classe che implementa l'applet è *pack.MioApplet* e il file *.class* risiede sull'host *unhost*, dentro la cartella *app* (all'interno del package *pack*).

All'interno della coppia <*applet*></*applet*> può essere usato il tag <*param*> per passare dei valori di inizializzazione ad un applet. Questo meccanismo può risultare utile per personalizzare le stringhe visualizzate sui bottoni secondo lingue diverse o per indicare all'applet la locazione di immagini e di altre risorse: così facendo non è necessario modificare il codice sorgente dell'applet, ma solo la pagina HTML contenente il tag <*applet*>. Il tag <*param*> ha la seguente forma:

<param name="*nomeDelParametro*" ***value=****valoreDelParametro****>***

Per recuperare il valore di un parametro, un applet deve invocare il metodo della classe *JApplet* ***String getParameter(String n)*** passando come argomento il nome del parametro stesso.

Come esempio, consideriamo il seguente applet:

```
import javax.swing.*;
public class ParamApplet extends JApplet
{ public void init()
  { String s = getParameter("testoBottone");
    JButton b = new JButton(s);
    getContentPane().add(b);
  }
}
```

Supponiamo che il file HTML relativo al tag <*applet*> abbia il seguente contenuto:

```
<html><title> File Param.html </title><body>
<h1> Param </h1>
<br>
<applet code="ParamApplet.class" width=100 height=100>
<param name="testoBottone" value="namaste">
</applet></body></html>
```

Visualizzando la pagina si ottiene un risultato simile a quello di Figura 4.4.

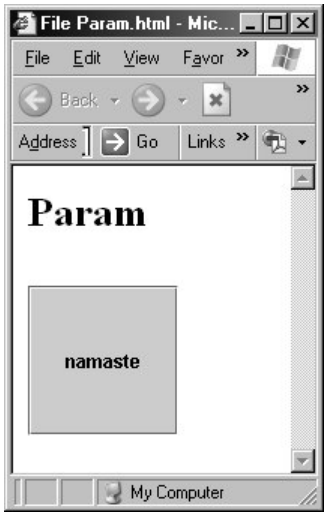


Fig. 4.4. Esempio di tag <*param*>.

4.1.6. Sicurezza

Quando un utente visita una pagina web che contiene un tag <*applet*>, sul proprio calcolatore viene scaricato il codice dell'applet ed eseguito all'interno del browser. Questo pone evidenti problemi di sicurezza: un programmatore malintenzionato potrebbe infatti scrivere degli applet con

lo scopo di leggere dati riservati memorizzati sulla macchina dell'utente, o per compiere azioni vandaliche sul contenuto della sua memoria di massa.

Per prevenire problemi di questo tipo i browser applicano agli applet delle restrizioni che impediscono loro di compromettere la sicurezza del sistema. Le politiche di sicurezza possono variare da browser a browser, ma in generale valgono le seguenti regole:

- un applet non può accedere al file system del calcolatore che lo esegue;
- un applet non può aprire connessioni con host diversi da quello da cui è stato scaricato;
- un applet non può mandare in esecuzione altri programmi;
- le finestre generate da un applet hanno un aspetto leggermente diverso da quello delle altre applicazioni (per esempio per evitare che un applet, facendo finta di essere un programma di sistema, chieda all'utente di inserire la propria password per poi comunicarla all'host da cui è stato scaricato).

Queste impostazioni predefinite possono essere variate, anche in maniera significativa, operando sui parametri di configurazione del browser. Per esempio, ad un applet possono essere garantiti tutti i privilegi di una applicazione standard nel caso in cui esso sia stato firmato digitalmente da un entità fidata ed il certificato sia riconosciuto come valido dal browser.

4.2. Servlet

Un *servlet* è una istanza di una classe Java, usato per estendere le funzionalità di un programma servitore, ed opera secondo un modello di programmazione di tipo richiesta/risposta: riceve le domande di servizio dai clienti, esegue la computazione richiesta e invia loro la risposta (non è in grado di operare se non quando stimolato). Nella maggior parte dei casi i servlet vengono impiegati per estendere le funzionalità di servitori HTTP, sebbene in linea teorica il servitore possa essere di tipo generico. Una

applicazione web (*web application*) è un insieme di servlet logicamente correlati.

Nel contesto delle applicazioni Internet, i servlet sono usati per produrre pagine HTML il cui contenuto è generato dinamicamente. I seguenti sono casi tipici in cui il contenuto di una pagina HTML non può essere statico:

- il contenuto della pagina dipende da dati forniti dall'utente, come avviene, per esempio, nell'ambito delle applicazioni di commercio elettronico relativamente al "cestino" dell'utente;
- il contenuto della pagina dipende da dati che cambiano frequentemente, come avviene, per esempio, nel caso delle pagine che contengono dati relativi agli indici di borsa;
- i dati relativi ad un sito Web sono contenuti in una base di dati e il servlet si occupa di estrarli e di presentarli all'utente sotto forma di pagine HTML.

Rispetto a tecnologie analoghe, i servlet godono della proprietà di poter essere eseguiti su macchine con architetture diverse senza la necessità di dover essere modificati o ricompilati, grazie alla portabilità del byte-code e alla standardizzazione delle interfacce di programmazione. Questo aspetto è particolarmente rilevante se si pensa che le applicazioni realizzate dal lato servitore sono in genere eseguite su un parco macchine caratterizzato da maggiore eterogeneità rispetto al caso delle applicazioni desktop.

Il ciclo di vita di un servlet è regolato dal processo servitore a cui è associato: al verificarsi di certe condizioni, tale processo invoca i metodi dell'interfaccia *Servlet* (package *javax.servlet*) che deve essere implementata dalla classe a cui appartiene il servlet stesso. L'interfaccia prevede i seguenti metodi:

public void init (ServletConfig sc) throws ServletException
viene invocato dal processo servitore una sola volta, all'atto del caricamento del servlet: il parametro di tipo *ServletConfig*, contiene informazioni riguardanti l'ambiente di esecuzione;

public void destroy ()
viene invocato dal processo servitore una sola volta, quando il servlet viene rimosso dall'ambiente di esecuzione: l'istante in cui un servlet viene rimosso dipende dal tipo e dalla configurazione del processo servitore (in

alcuni casi questo avviene quando il servlet rimane inattivo per una certa quantità di tempo, in altri solo durante la fase di arresto del processo servitore stesso); tale metodo contiene in genere codice "di pulizia";

public void service (ServletRequest** req **, ServletResponse** res **)
throws ServletException , IOException

viene invocato ogni volta che arriva una richiesta da parte di un cliente: il parametro *req* contiene i dati della richiesta, mentre il parametro *res* viene usato per rispondere al cliente;

public ServletConfig getServletConfig ( )

restituisce l'oggetto *ServletConfig*;

public String getServletInfo ()

restituisce una stringa che descrive il servlet, stringa che viene in genere usata dagli strumenti di amministrazione del servitore.

Piuttosto che implementare direttamente l'interfaccia, per il programmatore è più conveniente estendere la classe *GenericServlet* (package *javax.servlet*) che fornisce una implementazione di base dei vari metodi ad esclusione di *service()* (che deve pertanto essere ridefinito).

4.2.1. Richieste, risposte e ambiente di esecuzione

L'oggetto di tipo *ServletRequest* del metodo *service()* contiene le informazioni relative alla richiesta da servire. Le informazioni possono essere contenute nell'oggetto *ServletRequest* sotto forma di coppie nome/valore, oppure possono essere recuperate ottenendo dall'oggetto *ServletRequest* uno stream da cui leggere i dati. Il metodo ***String getParameter(String** nome**)*** restituisce il valore associato al nome passato come parametro, mentre i metodi ***ServletInputStream getInputStream()*** e ***BufferedReader getReader()*** restituiscono lo stream da cui leggere i dati (*ServletInputStream* è una sottoclasse di *InputStream* dotata di un metodo per leggere una riga alla volta). Altri metodi della classe *ServletRequest* sono:

public int getContentLength ()

restituisce la lunghezza dei dati da leggere dallo stream, oppure -1 se la lunghezza non è nota;

public String getContentType ()
restituisce il tipo dei dati associati alla richiesta (per esempio "text/plain", "text/html", eccetera);

public String getProtocol ()
restituisce una stringa che identifica il protocollo con cui è stata fatta la richiesta (per esempio, "HTTP/1.1").

L'oggetto di tipo *ServletResponse* del metodo *service()* deve essere usato dal servlet per confezionare la risposta da trasmettere al cliente. Se la risposta contiene dati in forma binaria, per esempio un'immagine, è necessario ottenere uno stream di uscita orientato al byte con il metodo ***getOutputStream()*** della classe *ServletResponse*, e quindi scrivere i dati nello stream stesso. Se invece la risposta è di tipo testuale, bisogna usare lo stream orientato al carattere restituito dal metodo ***PrintWriter getWriter()***. Altri metodi della classe *ServletResponse* sono:

public void setContentType (String s)
imposta il tipo della risposta (per esempio "text/plain", "image/jpg", eccetera);

public void setContentLength (int l)
imposta la lunghezza dei dati contenuti nella risposta: tale lunghezza viene impostata automaticamente nel caso in cui i dati in uscita sono interamente contenuti nel buffer di uscita, mentre negli altri casi è opportuno impostarla manualmente (a patto che la lunghezza della risposta sia nota).

Un servlet può avere informazioni sul proprio ambiente di esecuzione sia in fase di inizializzazione che durante l'esecuzione. Le informazioni di inizializzazione sono passate al servlet attraverso l'oggetto di tipo *ServletConfig* del metodo *init()*. I parametri di inizializzazione sono contenuti sotto forma di coppie nome/valore. Il metodo ***String getInitParameter(String*** *nome)* della classe *ServletConfig* restituisce il valore del parametro avente il nome indicato. Per esempio si può avere:

```
public void init(ServletConfig c)
```

```
{ String valore = c.getInitParameter("nomeParametro");
  ...
}
```

L'amministratore del servitore definisce le coppie nome/valore da passare ad un servlet con modalità che sono dipendenti dal tipo di servitore usato (vedremo un esempio concreto in seguito).

La classe *ServletConfig* è dotata del metodo ***ServletContext getServletContext()*** che restituisce un riferimento al contesto in cui il servlet viene eseguito. I servlet che appartengono alla stessa applicazione web condividono il medesimo contesto e possono usarlo per scambiare informazioni. Alcuni metodi della classe *ServletContext* sono:

public void log (String *msg* ***)***
scrive il messaggio specificato nel file di log del servitore;

public void setAttribute (String `s` ***, Object*** `o` ***)***
associa l'oggetto *o* alla stringa *s* (utile, insieme al metodo seguente, per scambiare informazioni tra servlet diversi);

public Object getAttribute (String `s` ***)***
restituisce l'oggetto associato alla stringa *s*;

public String getInitParameter (String `s` ***)***
restituisce il valore associato al parametro di inizializzazione identificato da *s* (i parametri di inizializzazione sono comuni a tutti i servlet del contesto e sono quindi utili per memorizzare informazioni di configurazione dell'intera applicazione web).

A titolo di esempio riportiamo il codice di un semplice servlet che produce come risposta una pagina di testo contenente la stringa "*Ciao*".

```
import java.io.*;
import javax.servlet.*;
public class SempliceServlet implements Servlet
{ private ServletConfig config;
  public void init(ServletConfig c)
              throws ServletException
  { config = c; }
  public void destroy() {}
```

```
  public ServletConfig getServletConfig()
  { return config;  }
  public String getServletInfo()
  { return "Il mio primo servlet";  }
  public void service(ServletRequest req,
                      ServletResponse res)
           throws ServletException, IOException
  { res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    out.println("Ciao");
    out.close();
  }
}
```

4.2.2. Servlet HTTP

Nella quasi totalità dei casi i servlet estendono le funzionalità di servitori HTTP. In tali circostanze è opportuno che il programmatore definisca i propri servlet come sottoclassi della classe astratta *HttpServlet* (package *javax.servlet.http*), la quale ridefinisce il metodo *service()* in modo che richiami uno dei seguenti metodi a seconda del tipo di richiesta HTTP ricevuta (*GET*, *POST*, *PUT*, *HEAD*, *OPTIONS*, *DELETE*, *TRACE*):

void doGet(HttpServletRequest** req**, HttpServletResponse** res**)
void doPost(HttpServletRequest** req**, HttpServletResponse** res**)
void doPut(HttpServletRequest** req**, HttpServletResponse** res**)
void doHead(HttpServletRequest** req**, HttpServletResponse** res**)
void doOptions(HttpServletRequest** req**, HttpServletResponse** res**)
void doDelete(HttpServletRequest** req**, HttpServletResponse** res**)
void doTrace(HttpServletRequest** req**, HttpServletResponse** res**)

Tutti i metodi possono lanciare *ServletException* e *IOException*.

La sottoclasse di *HttpServlet* deve implementare i metodi di cui vuole ridefinire il comportamento. Se si ridefinisce il metodo *doGet()* e non il metodo *doHead()*, il metodo *service()* della classe astratta *HttpServlet* è in grado di generare risposte anche per le richieste *HEAD*, come risposte per le richieste *GET* private del corpo (vedi Appendice A).

Le classi *HttpServletRequest* e *HttpServletResponse* sono sottoclassi, rispettivamente, di *ServletRequest* e *ServletResponse*. Inoltre, la classe *HttpServletResponse* dispone dei metodi:

public void setStatus (int** code **)
consente di impostare il codice di stato della risposta (da usare quando non si verificano errori); il valore default del codice di stato di una risposta è pari a 200 (OK).

public void sendError (int** code **) throws IOException
invia al cliente una risposta che segnala una situazione di errore; la pagina viene creata dal servitore.

HttpServletResponse dispone anche di costanti (membri statici) corrispondenti ai codici più comuni, per esempio *SC_OK* (con valore 200), *SC_NOT_FOUND* (con valore 404), eccetera.

Il seguente programma implementa un servlet minimale che risponde con la stringa *"Hello World"*:

```
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet
{ public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
                throws ServletException, IOException
  { PrintWriter out = res.getWriter();
    out.println("Hello World");
  }
}
```

Nel caso in cui si voglia produrre come risposta una pagina HTML è sufficiente indicare che il contenuto della risposta è di tipo *"text/html"*, quindi scrivere sullo stream di uscita le righe che compongono la pagina HTML stessa. Per esempio si può avere:

```
public class HelloWorldHTML extends HttpServlet
{ public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
              throws ServletException, IOException
  { res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println(  "<HTML>\n" +
                  "<HEAD><TITLE>" +
                  "Hello World" +
                  "</TITLE></HEAD>\n" +
```

```
                    "<BODY>\n" +
                    "<H1>Hello World</H1>\n" +
                    "</BODY></HTML>");
  }
}
```

4.2.3. Installazione ed esecuzione di un Servlet

In questo sottoparagrafo faremo riferimento all'ambiente di esecuzione *Tomcat*, un prodotto *open source* del progetto *Apache*, in grado di funzionare come servitore HTTP e di eseguire servlet. Il software può essere scaricato a partire dalla pagina:

http://jakarta.apache.org/tomcat/index.html

In particolare, considereremo la versione 5.0, al momento la più recente. Dentro la cartella di istallazione di Tomcat è presente una cartella *webapps*, che contiene le applicazioni web correntemente istallate. È anche presente una cartella *common/lib* che contiene le classi, sotto forma di archivio (*servlet-api.jar*), dei package *javax.servlet* e *javax.servlet.http*. Tale archivio deve essere incluso nel classpath per poter compilare un servlet, come illustrato dal seguente comando:

```
javac -classpath cartelladitomcat/common/lib/servlet-api.jar MioServlet.java
```

Una applicazione web è costituita da un insieme di file e cartelle organizzati secondo una certa struttura. La cartella principale ha il nome dell'applicazione web (nomeappweb) e contiene al suo interno:

- **.html*: i file HTML, ed eventuali altri file necessari quali le immagini, relativi alle pagine statiche dell'applicazione;
- */WEB-INF/web.xml*: il file che descrive i servlet che formano l'applicazione web (esso contiene anche le informazioni di inizializzazione dei servlet, ossia quelle passate tramite l'oggetto di tipo *ServletConfig*);
- */WEB-INF/classes/:* la cartella che contiene i file *.class* dei servlet che costituiscono l'applicazione web (nel caso in cui un servlet

faccia parte di un package, allora la cartella *classes* deve contenere una struttura di cartelle analoga alla struttura del package);
- */WEB-INF/lib/* la cartella che contiene eventuali classi di librerie aggiuntive usate dai servlet che compongono l'applicazione web.

Una volta preparata una cartella con tale struttura, per istallare l'applicazione è sufficiente copiare tale cartella all'interno della cartella *webapps* di Tomcat (in genere occorre riavviare il server).

Un servlet viene invocato mediante un URL del tipo:

http://nomehost:8080/nomeappweb/nomeServlet

Riportiamo un file *web.xml* di esempio (anche non conoscendo la sintassi del linguaggio XML non dovrebbe essere difficile comprenderne la struttura):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>

  <!-- Descrizione generale dell'applicazione -->
  <display-name>La mia applicazione</display-name>
  <description>
    Descrizione dell'applicazione ...
  </description>

  <!-- Parametri di inizializzazione del contesto.
    Possono essere recuperati dal Servlet
    come segue:
    String valore =
    getServletContext().getInitParameter("nome");
    E' possibile definire un numero arbitrario
    di parametri di inizializzazione.-->

  <context-param>
    <param-name>amministratore</param-name>
    <param-value>pippo@pluto.com</param-value>
    <description>
      L'indirizzo di posta elettronica
      dell'amministratore.
    </description>
  </context-param>
```

```
  <!-- Definizione dei servlet che compongono
    l'applicazione web.
    Ogni Servlet puo`recuperare i parametri di
    inizializzazione come segue:
    String valore =
      getServletConfig().getInitParameter("nome");
  -->

  <servlet>
    <servlet-name>nomeLogico1</servlet-name>
    <description>
      Descrizione del Servlet
    </description>
    <servlet-class>
      mypackage1.MyServlet1
    </servlet-class>
    <init-param>
      <param-name>nomeParametro1</param-name>
      <param-value>valoreParametro1</param-value>
    </init-param>
    <init-param>
      <param-name>nomeParametro2</param-name>
      <param-value>valoreParametro2</param-value>
    </init-param>
  </servlet>

  <!-- Definisce gli URI con cui i Servlet sono
    accessibili dall'esterno.
    Per esempio, per rendere disponibile il
    servlet precedente con un URI del tipo
    http://nomehost:8080/nomeapplicazioneweb/xyz
    si deve operare come segue:-->
  <servlet-mapping>
    <servlet-name>nomeLogico1</servlet-name>
    <url-pattern>/xyz</url-pattern>
  </servlet-mapping>

</web-app>
```

4.2.4. Lettura dei dati di un form

Un *form* è un'area di una pagina HTML che consente all'utente di

inserire dei dati e di inviarli al processo servitore. I form sono usati comunemente, per esempio per inviare le parole chiave ad un motore di ricerca, per gestire l'autenticazione di un utente, o immettere il prezzo in un'asta elettronica.

I dati possono essere inviati al processo servitore con una richiesta HTTP di tipo *GET* o *POST*. Nel caso in cui si usi una *GET*, in fondo all'URL vengono appesi i nomi e i valori dei parametri che l'utente invia al servitore opportunamente codificati. Per esempio, eseguendo una ricerca su *Google* con la parola chiave "*java*", il browser genera un URL del tipo *http://www.google.it/search?q=java*. Nel caso in cui i dati vengono inviati tramite una *POST*, i nomi e i valori dei parametri sono posti su una riga separata e non sono visibili nell'URL.

Un servlet è in grado di gestire i dati provenienti da un form in maniera molto semplice: è sufficiente invocare sull'oggetto *HttpServletRequest* il metodo ***String getParameter(String s)***, specificando il nome del parametro di cui si vuole ottenere il valore, indipendentemente dal fatto che i dati vengano inviati con una *GET* o con una *POST*.

Nel caso in cui il parametro non abbia nessun valore, il metodo *getParameter()* restituisce la stringa vuota. Nel caso invece in cui un parametro con tale nome non esista, viene restituito il valore *null*.

A titolo di esempio riportiamo il codice di un'applicazione web che inverte il testo inserito dall'utente. La pagina HTML che richiede il testo da invertire è la seguente:

```
<html>
<head><title>Invertitesto</title></head>
<body>
<p>
Immetti del testo nel campo sottostante e premi il
bottone "Invia"
</p>
<form name="input" action="invertitesto">
  <input type="text" name="dati" size="30">
  <br>
  <input type="submit" value="Invia">
</form>
</body>
</html>
```

Tale pagina è memorizzata all'interno del file *index.html*. Una volta premuto il bottone "*Invia*" i dati vengono inviati all'URL *invertitesto*.

L'inversione del testo è eseguita dal servlet *InvertiTesto*:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class InvertiTesto extends HttpServlet
{ private ServletConfig servletConfig;
  private ServletContext servletContext;
  public void init(ServletConfig sc)
  { servletConfig = sc;
    servletContext =
             servletConfig.getServletContext();
    servletContext.log("Servlet inizializzato");
  }
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
               throws IOException, ServletException
  { String s = req.getParameter("dati");
    char[] car = s.toCharArray();
    char[] tmp = new char[car.length];
    for(int i=0; i<car.length; i++)
      tmp[tmp.length - i - 1] = car[i];
    String inv = new String(tmp);
    PrintWriter out = res.getWriter();
    res.setContentType("text/html");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Rigiratesto</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<p>L'inverso di " + s +
                " e' " + inv + "</p>");
    out.println("</body>");
    out.println("</html>");
  }
}
```

L'URL *invertitesto* a cui il form invia i dati viene associato al servlet *InvertiTesto* con il seguente file *web.xml*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
  <display-name>
    Come gestire i dati di un form
  </display-name>
  <description>
    Questa applicazione mostra come
    gestire i dati di un form.
  </description>
  <servlet>
    <servlet-name>invertitesto</servlet-name>
    <description>
      Inverte la stringa ricevuta.
    </description>
    <servlet-class>InvertiTesto</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>invertitesto</servlet-name>
    <url-pattern>/invertitesto</url-pattern>
  </servlet-mapping>
</web-app>
```

Figura 4.5. La pagina contenente il form.

A questo punto è sufficiente creare dentro *webapps* una cartella avente il nome dell'applicazione web, per esempio *inv*. Quindi occorre i) copiare dentro *inv* il file *index.html*, ii) creare una sottocartella *WEB-INF* e copiarci dentro *web.xml*, iii) creare dentro WEB-INF una sottocartella *classes* e copiarci dentro *InvertiTesto.class*.

Per accedere all'applicazione dalla macchina locale è sufficiente lanciare un browser e digitare l'URL *http://localhost:8080/inv/index.html*, come mostrato in Figura 4.5.

4.2.5. Gestione dei cookie

Un *cookie* è un piccolo file di testo che un servitore HTTP invia ai suoi clienti e che questi ultimi restituiscono al servitore in occasione di visite successive alla stessa pagina (o allo stesso dominio). I cookie vengono salvati dai browser in memoria permanente e quindi risultano utili al servitore per ottenere informazioni relative a visite ad un determinato sito eseguite in precedenza.

Tra gli usi dei cookie citiamo i più importanti:

- *evitare all'utente di inserire nome e password*: la prima volta che l'utente si registra per poter accedere ad un sito Internet, il server invia un cookie al browser dell'utente; in occasione di visite successive l'utente non deve più digitare nome e password grazie alle informazioni contenute nel cookie restituito dal browser al server;
- *memorizzare le preferenze dell'utente*: le preferenze di un utente, per esempio riguardanti l'aspetto grafico di un sito Internet, possono essere memorizzate in un cookie in modo tale da essere applicate durante le visite successive;
- *gestione di sessioni*: poiché il protocollo HTTP è senza stato, all'arrivo di una richiesta il servitore non è in grado di conoscere la storia delle richieste precedenti eseguite dallo stesso utente (si pensi al carrello della spesa dei siti di commercio elettronico: il server ha bisogno di conoscere il contenuto del carrello di un utente quando questi decide di acquistare tutto ciò che ha selezionato in precedenza); i cookie possono essere usati per mantenere informazioni di sessione.

Un cookie è caratterizzato da un nome e da un valore, più alcuni attributi opzionali quali un commento, una data di scadenza e un numero di versione. Una volta scaricato un cookie da un host, il browser lo allega a

tutte le richieste eseguite successivamente verso tale host. Per inviare un cookie ad un cliente un servlet deve:

1. creare un istanza della classe *Cookie*, per esempio attraverso il costruttore ***Cookie(String** n, **String** v)* in cui vengono specificati il nome *n* e il valore *v* da associare al cookie;
2. impostare il valore di uno o più campi opzionali;
3. aggiungere il cookie alla risposta da inviare al cliente con il metodo ***void addCookie(Cookie** c)* della classe *HttpServletRequest*.

Per operare sui campi opzionali sono disponibili i seguenti metodi:

***public void setComment (String** s)*
public String getComment ()
servono rispettivamente a impostare e a conoscere il valore del campo commento;

***public void setDomain (String** s)*
public String getDomain ()
consentono rispettivamente di impostare e di recuperare l'insieme degli host (il dominio Internet) a cui il cookie deve essere restituito (normalmente il browser restituisce il cookie solo all'host da cui lo ha scaricato);

***public void setMaxAge (int** a)*
public int getMaxAge ()
servono rispettivamente a impostare e a conoscere l'intervallo di validità di un cookie (espresso in secondi).

Un servlet recupera gli oggetti cookie associati alla richiesta invocando il metodo ***Cookie[] getCookies()*** della classe *HttpServletRequest*. Quindi esamina l'array cercando gli elementi che hanno un nome di suo interesse.

Riportiamo il codice di un servlet che fa uso dei cookie per tener traccia del colore preferito degli utenti. La prima volta che l'utente visita la pagina, la richiesta *GET* viene gestita dal servlet generando una pagina HTML che chiede all'utente, mediante un form, di specificare il proprio colore preferito (Figura 4.6). Tale indicazione viene restituita al servlet con una seconda richiesta, di tipo *POST* (la stringa inserita dall'utente viene associata a un parametro di nome "*col*"). La richiesta viene gestita dal

servlet generando un oggetto cookie che ha per nome la stringa "*Colore*" e come valore la preferenza dell'utente. Il cookie viene quindi inviato al browser come parte della risposta. In occasione di visite successive alla stessa pagina, la richiesta *GET* generata dal browser conterrà il cookie precedentemente ricevuto. In questo caso l'implementazione del metodo *doGet()* trova un cookie con il nome "*Colore*" e produce un'opportuna pagina di risposta.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorePreferito extends HttpServlet
{ public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
              throws IOException, ServletException
  { PrintWriter out = res.getWriter();
    res.setContentType("text/html");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Colore preferito</title>");
    out.println("</head>");
    out.println("<body>");
    Cookie[] cookies = req.getCookies();
    if(cookies != null)
      for(int i=0; i<cookies.length; i++)
      { if(cookies[i].getName().equals("Colore"))
        { String preferito = cookies[i].getValue();
          out.println("<p>Il tuo colore preferito e':"
                      + preferito + "</p>");
          out.println("</body>");
          out.println("</html>");
          return;
        }
      }
    out.println("<p>Inserisci il tuo colore " +
                      "preferito</p>");
    out.println("<form name=\"input\" " +
                 "action=\"colorepreferito\" " +
                 "method=\"post\">");
    out.println("<input type=\"text\" name=\"col\">");
    out.println("<input type=\"submit\" "+
                "value=\"Invia\">");
```

```
    out.println("<form>");
    out.println("</body>");
    out.println("</html>");
  }
  public void doPost(HttpServletRequest req,
                     HttpServletResponse res)
              throws IOException, ServletException
  {  String preferito = req.getParameter("col");
    Cookie cookie = new Cookie("Colore", preferito);
    res.addCookie(cookie);
    PrintWriter out = res.getWriter();
    res.setContentType("text/html");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Colore preferito</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<p>Adesso ricarica la pagina</p>");
    out.println("</body>");
    out.println("</html>");
  }
}
```

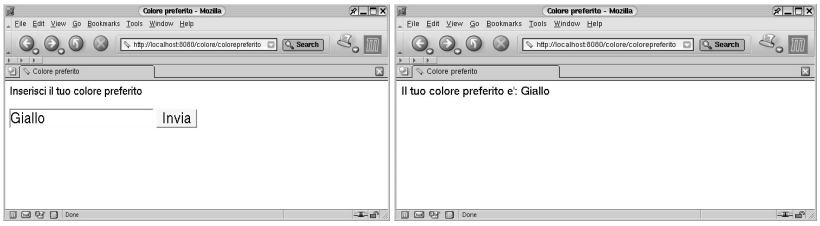


Figura 4.6. Pagine generate dal servlet *ColorePreferito*.

4.2.6. Sessioni

L'assenza di stato del protocollo HTTP complica la realizzazione di applicazioni Internet in cui è necessario capire se le richieste di un utente appartengono tutte ad una stessa sessione o meno (come già citato in precedenza, l'esempio più comune è quello del carrello della spesa

relativamente ai siti di commercio elettronico). Questo problema viene generalmente risolto o mediante l'uso di cookie, oppure attraverso una tecnica che consiste nell'appendere in fondo all'URL un identificatore di sessione (*URL rewriting*). Una terza modalità si basa sull'uso di form HTML in cui è presente un campo nascosto usato per memorizzare un identificatore di sessione.

I servlet dispongono di una insieme di metodi (*HttpSession API*) attraverso i quali è possibile capire quando le richieste appartengono alla medesima sessione. È inoltre possibile associare alla sessione alcune informazioni in modo tale che queste siano persistenti al di là della singola richiesta.

Il servitore HTTP rappresenta una sessione come una istanza della classe *HttpSession* e ne passa il riferimento al servlet come parte della richiesta.

Il metodo ***HttpSession getSession(boolean*** *create)* della classe *HttpRequest* restituisce l'oggetto *HttpSession* associato alla sessione corrente. Il parametro *create* indica se un nuovo oggetto sessione deve essere creato automaticamente nel caso in cui non ne esista già uno.

I seguenti metodi della classe *HttpSession* consentono di conoscere le caratteristiche di un oggetto sessione:

public String getId ()
restituisce una stringa contenente l'identificatore univoco della sessione;

public long getCreationTime ()
restituisce l'istante di creazione della sessione espresso come il numero di millisecondi trascorsi a partire dalla mezzanotte del 1/1/1970;

public long getLastAccessedTime ()
simile al metodo precedente, ma restituisce l'istante dell'ultima richiesta;

public boolean isNew ()
restituisce *true* se il cliente non ha ancora preso parte alla sessione.

Le informazioni che si vogliono rendere persistenti al di là della singola connessione sono memorizzate sotto forma di coppie chiave/valore (la chiave è una stringa che identifica l'attributo da memorizzare, mentre il valore può essere un qualunque oggetto). Per aggiungere attributi ad una

sessione e per conoscerne il valore sono disponibili i seguenti metodi della classe *HttpSession*:

public Object getAttribute (String** chiave **)
restituisce l'oggetto associato alla chiave passata come parametro (*null* se nessun oggetto è registrato sotto tale nome);

public void setAttribute (String** chiave **, Object** valore **)
memorizza l'oggetto *valore* associandolo alla chiave specificata.

A titolo di esempio riportiamo il codice di un servlet che usa il meccanismo delle sessioni per memorizzare il numero di visite precedenti eseguite dall'utente. La prima volta che l'utente visita la pagina, il servlet crea un nuovo oggetto sessione e vi memorizza la coppia chiave/valore costituita dalla stringa "*visite*" e da un *Integer* inizializzato con il valore *1*. Viene inoltre visualizzata una pagina contenente un messaggio di benvenuto. In occasione di visite successive, se la sessione non è ancora scaduta, il servlet recupera l'attributo associato alla stringa "*visite*", lo converte ad *Integer*, lo incrementa di uno e lo memorizza nuovamente nell'oggetto sessione sotto lo stesso nome. Viene inoltre visualizzata una pagina che indica il numero di visite precedenti, pari al valore dell'attributo.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class NumeroVisite extends HttpServlet
{ public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
                throws IOException, ServletException
  { String body;
    Integer n;
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println(  "<html>" +
                  "<head>" +
                  "<title>Numero visite</title>" +
                  "</head>" +
                  "<body>");
    HttpSession ses = req.getSession(true);
```

```
    if(ses.isNew())
    {  n = new Integer(1);
       body = "Benvenuto !";
    }
    else
    {  n = (Integer) ses.getAttribute("visite");
       n = new Integer(n.intValue() + 1);
       body = "Questa e' la tua visita numero " + n;
    }
    ses.setAttribute("visite", n);
    out.println(body +
                "</body>" +
                "</html>");
  }
}
```

4.2.7. Concorrenza

L'ambiente di esecuzione dei servlet è in molti casi multi-thread. Come conseguenza i metodi *doGet()*, *doPost()*, eccetera, possono essere invocati da più thread in maniera simultanea allo scopo di rispondere alle richieste provenienti da più clienti. Il programmatore deve quindi preoccuparsi di strutturare il codice dei metodi in modo tale che non si creino problemi legati ad una loro esecuzione concorrente, per esempio introducendo dei blocchi sincronizzati.

4.3. Applicazione web completa

In questa sezione presentiamo come esempio conclusivo una applicazione web completa, costituita da pagine HTML statiche, applet e servlet. L'applicazione ha lo scopo di permettere all'utente di inserire del codice HTML usando un semplice browser, quindi di visualizzare in una finestra separata il codice HTML precedentemente introdotto. L'applicazione si compone di:

- un file *index.html* che costituisce il punto di ingresso dell'applicazione web;
- un applet (*MyApplet*) contenuto in *index.html* che consente all'utente di inserire del codice HTML;
- un servlet (*Uno*) che riceve il codice HTML dell'utente e lo memorizza tra le sue informazioni di sessione;
- un servlet (*Due*) che quando richiesto dal cliente restituisce una pagina HTML costruita con il codice sottomesso in precedenza.

Figura 4.6.
Finestra di sinistra: visualizza la pagina *index.html*
Finestra di destra: visualizza la pagina HTML il cui codice è stato inserito nel campo di testo dell'applet.

Il file *index.html* ha il seguente contenuto:

```
<html>
  <head>
    <title>Un'applicazione web completa</title>
  </head>
  <body bgcolor='yellow'>
    <p> Scrivi del codice HTML nell'area
```

```
        sottostante e premi "Invia"<p>
    <applet code="MyApplet.class"
      width='400' height='400'>
    </applet>
  </body>
</html>
```

L'interfaccia grafica dell'applet (Fig. 4.6) si compone di una *JTextArea*, usata per inserire il codice HTML, e un *JButton* che, quando premuto, invia il codice che è stato immesso nell'area di testo. In particolare, il codice viene inviato all'URL relativo "*uno*" individuato a partire dal *codebase* dell'applet mediante una *HttpURLConnection*. L'esecuzione del metodo *setDoOutput(true)* sull'oggetto di tipo *HttpURLConnection* indica che il trasferimento dei dati avviene dal cliente al servitore (e non viceversa come accade di default) mediante una richiesta HTTP di tipo *POST* (che è l'azione predefinita quando si accede ad un URL in scrittura). Dopo aver inviato il testo, l'applet ottiene un riferimento al proprio contesto di esecuzione e chiede al browser di visualizzare, in una nuova finestra, il documento HTML individuato dall'URL relativo "*due*" (sempre calcolato a partire dal proprio *codebase*).

```
import java.net.*;
import java.io.*;
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
public class MyApplet extends JApplet
{ private JButton JButton1;
  private JTextArea JTextArea1;
  public class Ascoltatore
                implements ActionListener
  { public void actionPerformed(ActionEvent evt)
    { String testo = JTextArea1.getText();
      URL codebase = getCodeBase();
      try
      { URL url = new URL(codebase, "uno");
        HttpURLConnection c =
             (HttpURLConnection) url.openConnection();
        c.setDoOutput(true);
        OutputStream os = c.getOutputStream();
```

```
          PrintWriter p = new PrintWriter(os, true);
          p.print(testo);
          p.close();
          c.connect();
          int resp = c.getResponseCode();
          if(resp == HttpURLConnection.HTTP_OK)
          { AppletContext context = getAppletContext();
            context.showDocument
                 (new URL(codebase, "due"), "Nuova");
          }
          else { showStatus("Errore!"); }
        }
        catch(MalformedURLException e)
        { showStatus("Errore URL: " + e); }
        catch(IOException e)
        { showStatus("Errore: " + e);   }
      }
  }
  public void init()
  { JTextArea1 = new JTextArea();
    JButton1 = new JButton("Invio");
    getContentPane().add(JTextArea1,
                         BorderLayout.CENTER);
    getContentPane().add(JButton1,
                         BorderLayout.SOUTH);
    JButton1.addActionListener(new Ascoltatore());
  }
}
```

Il servlet *Uno* implementa il solo metodo *doPost()*, che viene invocato quando l'applet invia il codice HTML. Tale metodo legge i dati inviati dall'applet e li memorizza all'interno di un *ByteArrayOutputStream*, quindi ne ricava una stringa e la memorizza nell'oggetto sessione sotto forma di un attributo di nome "*testo*". Il servlet è il seguente:

```
package pack;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Uno extends HttpServlet
{ public void doPost(HttpServletRequest request,
```

```
                    HttpServletResponse response)
              throws ServletException, IOException
  { String testo;
    ByteArrayOutputStream baos =
                      new ByteArrayOutputStream();
    byte[] buf = new byte[1024];
    HttpSession ses = request.getSession(true);
    InputStream is = request.getInputStream();
    int lun = request.getContentLength();
    int quanti;
    if(lun != -1)
    { // la lunghezza della risposta e' nota
      int rimasti = lun, d;
      while(rimasti > 0)
      { // numero massimo di byte da leggere
        // con la prossima read
        if (buf.length > rimasti) d = rimasti;
             else d = buf.length;
        quanti = is.read(buf, 0, d);
        if(quanti == -1) break;
        // non si riesce a leggere i dati rimanenti
        rimasti -= quanti;
        baos.write(buf, 0, quanti);
      }
    }
    else
    { // la lunghezza della risposta non e' nota
      while((quanti = is.read(buf, 0, buf.length))
             != -1)
          baos.write(buf, 0, quanti);
    }
    testo = baos.toString();
    ses.setAttribute("testo", testo);
    response.setStatus(HttpServletResponse.SC_OK);
  }
}
```

Il servlet *Due* implementa il solo metodo *doGet()*, che viene invocato quando l'applet chiede al browser di visualizzare il documento individuato dall'URL relativo "*due*". All'interno di tale metodo il servlet recupera il codice HTML precedentemente sottomesso dall'utente, memorizzato come

attributo all'interno dell'oggetto sessione, quindi produce la pagina di risposta. Il servlet è il seguente:

```
package pack;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Due extends HttpServlet
{ public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
              throws ServletException, IOException
  { response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession ses = request.getSession();
    String testo = "";
    if(ses != null)
      testo = (String) ses.getAttribute("testo");
    out.println(testo);
    out.close();
    response.setStatus(HttpServletResponse.SC_OK);
  }
}
```

Per istallare il tutto è necessario creare dentro la cartella *webapps* di Tomcat una cartella relativa all'applicazione, che supponiamo di chiamare *myapp*, organizzata come segue:

```
myapp
  |--index.html
  |--MyApplet.class
  |--MyApplet$Ascoltatore.class
  `--WEB-INF
       |--web.xml
       `--classes
            `--pack
                 |--Uno.class
                 `--Due.class
```

Il file *web.xml* deve associare i servlet *Uno* e *Due* agli URL relativi "*uno*" e "*due*":

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <servlet>
    <servlet-name>Uno</servlet-name>
    <servlet-class>pack.Uno</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Due</servlet-name>
    <servlet-class>pack.Due</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Uno</servlet-name>
    <url-pattern>/uno</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Due</servlet-name>
    <url-pattern>/due</url-pattern>
  </servlet-mapping>
</web-app>
```

Per eseguire l'applicazione occorre lanciare un browser e digitare l'URL della pagina iniziale. Per esempio, dalla macchina locale si specifica l'URL:

http://localhost:8080/myapp/index.html

5. Strutture dati e Java collection framework

5.1. Introduzione

In questo capitolo viene affrontato lo studio sistematico delle strutture dati con due diversi scopi. Il primo, essenzialmente didattico, consiste nel far comprendere come le strutture dati "classiche", quali liste, pile, code, alberi e tabelle, possano essere realizzate usando il linguaggio di programmazione Java. Il secondo scopo, più pratico, consiste nell'illustrare l'uso del *Java collection framework*, una libreria standard che mette a disposizione del programmatore realizzazioni predefinite di strutture dati: tali implementazioni rendono notevolmente più semplice e veloce lo sviluppo di programmi in cui è necessario gestire insiemi di dati, oltre ad essere realizzazioni efficienti e largamente utilizzate nel mondo produttivo.

La trattazione delle strutture dati fatta in questo capitolo prescinde dalle nozioni date per altri scopi in alcuni capitoli di questo o del volume precedente.

5.2. Tipi lista

Una categoria di dati astratti estremamente comune è costituita dai tipi

lista. Un tipo lista è un aggregato di elementi di un certo tipo, dove ogni elemento contiene un campo informativo e un riferimento all'elemento successivo. Il numero di elementi di una lista è variabile, rendendo questo tipo di dati astratti particolarmente adatto a memorizzare dati quando la quantità di informazione varia dinamicamente.

Una lista è caratterizzata da una testa (l'elemento di testa è il primo della lista) e da un fondo (l'elemento di fondo è l'ultimo della lista). Le operazioni principali che caratterizzano un tipo lista sono l'inserimento e l'estrazione di un elemento. Entrambe le operazioni possono coinvolgere la testa, il fondo, o un punto intermedio della lista (il punto intermedio può essere determinato specificando direttamente il numero d'ordine dell'elemento o dal verificarsi di una qualche condizione, per esempio di ordinamento fra gli elementi).

Gli elementi di una lista di *Object* possono essere definiti come istanze di una classe quale la seguente (i campi *info* e *next* possono assumere valore *null*):

```
class Elem
{ private Object info;
  private Elem next;
  ...
}
```

Una lista di *Object* supporta le operazioni specificate dai metodi dichiarati in una interfaccia come la seguente (omettiamo per il momento l'inserimento ordinato):

```
public interface Lista
{ void insTesta(Object o);
  void insFondo(Object o);
  void insPos(Object o, int k);
  Object getTesta();
  Object getFondo();
  Object getPos(int k);
  Object estTesta();
  Object estFondo();
  Object estPos(int k);
  int size();
}
```

I metodi *insTesta()*, *insFondo()*, e *insPos()* costruiscono un nuovo elemento avente come informazione l'oggetto *o* passato come argomento (anche il valore *null*), e lo inseriscono, rispettivamente, in testa, in fondo, e in posizione *k* (supponiamo che gli elementi siano numerati a partire da zero). I metodi *getTesta()*, *getFondo()*, e *getPos()* restituiscono rispettivamente il valore del campo informativo dell'elemento in testa, in fondo e di quello in posizione *k*. I metodi *estTesta()*, *estFondo()*, e *estPos()* estraggono dalla lista rispettivamente l'elemento in testa, l'elemento in fondo, e quello in posizione *k*, e restituiscono il valore del campo informativo dell'elemento estratto. I metodi *insPos()*, *getTesta()*, *getFondo()*, *getPos()*, *estTesta()*, *estFondo()*, e *estPos()* lanciano eccezioni di tipo *IndexOutOfBoundsException* quando non è possibile eseguire l'operazione (per esempio, se si tenta di accedere ad un elemento specificando una posizione che non esiste). Le eccezioni di tale classe, che è la stessa a cui appartengono le eccezioni sollevate quando si accede ad un array usando un indice non valido, appartengono alla categoria delle eccezioni *unchecked*, e pertanto il programmatore non è obbligato a gestirle (per questo motivo la clausola *throws IndexOutOfBoundsException* è omessa dall'intestazione dei metodi).

La seguente classe *ListaSemplice* realizza il tipo lista di *Object*:

```
public class ListaSemplice implements Lista
{  static class Elem
   {  Object info; Elem next;
      Elem(Object o, Elem n)
      {  info = o; next = n;  }
      Object getInfo()
      {  return info;  }
      Elem getNext()
      {  return next;  }
      void setNext(Elem n)
      {  next = n;   }
   }
   Elem testa;
   int dim;
   public void insTesta(Object o)
   {  Elem e = new Elem(o, testa);
      testa = e;
      dim++;
   }
```

```
public void insFondo(Object o)
{ Elem e = new Elem(o, null);
  if(testa == null)
  { testa = e;
    dim++;
    return;
  }
  Elem p = testa, q = testa.getNext();
  while(q != null)
  { p = q; q = q.getNext();  }
  p.setNext(e);
  dim++;
}
public void insPos(Object o, int k)
    throws IndexOutOfBoundsException
{ if(k < 0 ||  k > dim)
    throw new IndexOutOfBoundsException();
  if(testa == null || k == 0)
  { testa = new Elem(o, testa);
    dim++;
    return;
  }
  Elem p = testa, q = testa.getNext();
  int j = 1;
  while(j < k && q != null)
  { p = q;
    q = q.getNext();
    j++;
  }
  p.setNext(new Elem(o, q));
  dim++;
}
public Object getTesta()
    throws IndexOutOfBoundsException
{ if(testa == null)
    throw new IndexOutOfBoundsException();
  return testa.getInfo();
}
public Object getFondo()
    throws IndexOutOfBoundsException
{ if(testa == null)
    throw new IndexOutOfBoundsException();
  Elem q = testa;
```

```
    while(q.getNext() != null)
      q = q.getNext();
    return q.getInfo();
  }
  public Object getPos(int k)
       throws IndexOutOfBoundsException
  { if(testa == null || k < 0 || k >= dim)
      throw new IndexOutOfBoundsException();
    Elem q = testa;
    while(k>0 && q.getNext() != null)
    { q = q.getNext();
      k--;
    }
    return q.getInfo();
  }
  public Object estTesta()
       throws IndexOutOfBoundsException
  { if(testa == null)
      throw new IndexOutOfBoundsException();
    Object o = testa.getInfo();
    testa = testa.getNext();
    dim--;
    return o;
  }
  public Object estFondo()
       throws IndexOutOfBoundsException
  { if(testa == null)
      throw new IndexOutOfBoundsException();
    Elem p = testa, q = testa;
    while(q.getNext() != null)
    { p = q; q = q.getNext(); }
    Object o = q.getInfo();
    if(q == testa) testa = null;
    else p.setNext(null);
    dim--;
    return o;
  }
  public Object estPos(int k)
       throws IndexOutOfBoundsException
  { if(testa == null || k < 0 || k >= dim)
      throw new IndexOutOfBoundsException();
    Elem p = testa, q = testa;
    while(k > 0 && q.getNext() != null)
```

```
    { p = q;
      q = q.getNext();
      k--;
    }
    Object o = q.getInfo();
    if(q == testa) testa = q.getNext();
    else p.setNext(q.getNext());
    dim--;
    return o;
  }
  public int size()
  { return dim;
  }
  public String toString()
  { String s;
    if(testa == null)
    { s = "<>";
      return s;
    }
    s = "<" + testa.getInfo();
    Elem p = testa.getNext();
    while(p != null)
    { s += ", " + p.getInfo();
      p = p.getNext();
    }
    return s + ">";
  }
}
```

Le variabili membro della classe *ListaSemplice* non sono dichiarate private, in modo da poter essere usate anche da eventuali sue sottoclassi.

Nella classe *ListaSemplice*, come in molte altre classi di questo capitolo, viene ridefinito il metodo *toString().*

5.2.1. Lista: il metodo *equals()*

Il metodo *equals()* della classe *ListaSemplice* è quello ereditato dalla classe *Object*: esso restituisce *true* solo se i riferimenti dell'oggetto implicito e di quello specificato come argomento sono uguali. In molti casi può essere conveniente ridefinire il comportamento di tale metodo, per

esempio facendogli restituire il valore *true* quando le informazioni contenute nella lista individuata dal riferimento implicito sono uguali e nella stessa posizione (quindi, elemento per elemento) delle informazioni contenute nella lista individuata dal riferimento specificato come argomento, *false* altrimenti.

La classe *ListaSemplice2*, sottoclasse di *ListaSemplice*, ridefinisce il metodo *equals()* secondo tale criterio:

```
public class ListaSemplice2 extends ListaSemplice
{ public boolean equals(Object l)
  { if(l == null || !(l instanceof ListaSemplice2))
      return false;
    ListaSemplice2 l2 = (ListaSemplice2) l;
    if(l2.size() != size())  return false;
    for(int i=0; i<size(); i++)
    { Object v1 = getPos(i);
      Object v2 = l2.getPos(i);
      if(v1 == null && v2 == null ||
      v1 != null && v2 != null && v1.equals(v2))
        continue;
      return false;
    }
    return true;
  }
}
```

Il metodo *equals()* della classe *ListaSemplice2* restituisce *false* anzitutto quando le liste non hanno lo stesso numero di elementi. Diversamente, le informazioni contenute nelle due liste vengono confrontate invocando il metodo *equals()* sugli oggetti memorizzati negli elementi corrispondenti (nei campi *info*) delle due liste, e non appena si trovano due oggetti diversi (secondo il metodo *equals()* della classe a cui appartengono gli oggetti), il metodo restituisce il valore *false*.

5.3. Tipi pila

Una pila è un insieme ordinato di dati di uguale tipo in cui è possibile

eseguire operazioni di inserimento e di estrazione secondo la seguente regola: l'ultimo dato inserito è il primo ad essere estratto (regola di accesso *LIFO*: *Last In First Out*).

Il tipo di dato astratto pila di *Object* supporta le operazioni specificate dalla seguente interfaccia:

```
public interface Pila
{ void push(Object elem);
  Object pop();
  boolean isEmpty();
  boolean isFull();
  int size();
}
```

Il metodo *push()* inserisce l'oggetto *e* passato come argomento nella pila (il metodo lancia un'eccezione di tipo *PilaFullException* se non è possibile eseguire l'inserimento perché la pila è piena). Il metodo *pop()* restituisce l'oggetto in cima alla pila (lancia un'eccezione di tipo *PilaEmptyException* se si tenta di estrarre un elemento quando la pila è vuota). I metodi *isEmpty()* e *isFull()* possono essere usati per determinare se la pila è vuota oppure è piena, mentre *size()* restituisce il numero di oggetti contenuti nella pila.

Le due classi di eccezione possono essere definite come segue:

```
public class PilaFullException
               extends RuntimeException {}

public class PilaEmptyException
               extends RuntimeException {}
```

Le eccezioni di tali classi appartengono alla categoria delle eccezioni *unchecked*, in quanto oggetti di sottoclassi di *RuntimeException* (e non di *Exception*), per cui le clausole *throws PilaFullException* e *throws PilaEmptyException* dei metodi *push()* e *pop()* possono essere omesse.

5.3.1. Pila realizzata con array

Una pila può essere costituita da un array (*stack*) e da un indice (*top*):

l'array memorizza i dati nelle sue componenti e l'indice *top* individua in ogni istante la posizione relativa all'ultimo inserimento (e quindi il livello di riempimento della pila), come mostrato in fig. 5.1.

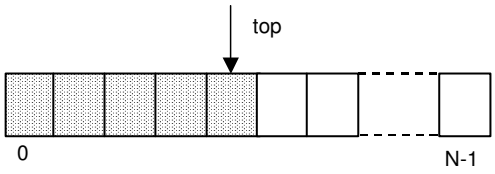


Figura 5.1. Una pila realizzata con un array di dimensione N, in cui sono contenuti cinque valori.

L'operazione di inserimento avviene nella componente dell'array successiva a quella indicata da *top*, mentre l'estrazione riguarda la componente dell'array individuata da *top* stesso. La pila è vuota quando *top* ha un valore che precede l'indice della prima componente dell'array, mentre la pila è piena quando il valore di *top* è pari all'indice dell'ultima componente dell'array.

La classe *PilaArray* realizza tali operazioni come segue:

```
public class PilaArray implements Pila
{  static int DEFAULT_SIZE = 10;
   int top;
   Object[] stack;
   public PilaArray(int s)
   {  top = -1;
      stack = new Object[s];
   }
   public PilaArray()
   {  this(DEFAULT_SIZE);  }
   public boolean isEmpty()
   {  return top == -1;  }
   public boolean isFull()
   {  return top == stack.length - 1; }
   public int size()
   {  return top + 1;  }
   public void push(Object e)
   {  if(isFull())
```

```
      throw new PilaFullException();
    top++; stack[top] = e;
  }
  public Object pop()
  { if(isEmpty())
      throw new PilaEmptyException();
    Object result = stack[top]; top--;
    return result;
  }
  public String toString()
  { String s = "---\n";
    if(!isEmpty())
      for(int i=top; i>=0; i--) s += stack[i] + "\n";
    s+="---";
    return s;
  }
}
```

5.3.2. Pila realizzata con lista

Un tipo pila può essere realizzato utilizzando una lista per la memorizzazione dei dati. In particolare, le operazioni di inserimento e di estrazione possono essere ottenute come inserimento ed estrazione in testa alla lista, in accordo alla disciplina LIFO.

Il tipo pila di *Object* può essere realizzato nel seguente modo:

```
public class PilaLista implements Pila
{ static class Elem
  { private Object o;
    private Elem next;
    Elem(Object e, Elem n)
    { o = e; next = n;  }
    Object getInfo()
    { return o;}
    Elem getNext()
    { return next;  }
  }
  Elem testa;
  int quanti;
  public boolean isEmpty()
  { return quanti == 0;  }
```

```
  public boolean isFull()
  { return false; }
  public int size()
  { return quanti;  }
  public void push(Object e)
  { Elem tmp = new Elem(e, testa);
    testa = tmp; quanti++;
  }
  public Object pop()
  { if(isEmpty())
      throw new PilaEmptyException();
    Object result = testa.getInfo();
    testa = testa.getNext(); quanti--;
    return result;
  }
  public String toString()
  { String s = "---\n";
    if(!isEmpty())
      for(Elem i = testa; i!=null; i = i.getNext())
        s += i.getInfo() + "\n";
    s+="---";
    return s;
  }
}
```

Osserviamo che, a differenza di quanto visto nel caso di pila realizzata mediante array, non esiste un costruttore che consente di specificare la dimensione della pila. Inoltre, poiché la pila non può mai essere piena, il metodo *isFull()* restituisce sempre *false*, mentre il metodo *push()* non può lanciare un'eccezione di tipo *PilaFullException*.

5.3.2. Pila: il metodo *equals()*

Anche nel caso del tipo di dato astratto pila di *Object* può essere opportuno ridefinire il comportamento del metodo *equals()* andando a confrontare il contenuto informativo dei due oggetti su cui il metodo viene invocato.

A tale scopo è utile definire una nuova interfaccia *Pila2* che estende *Pila* aggiungendole un metodo che consente di conoscere il contenuto di una pila anche in posizioni diverse dalla testa (le operazioni di inserimento

e di estrazione continuano in ogni caso a lavorare solo sulla testa della pila):

```
public interface Pila2 extends Pila
{ Object getElem(int i);
}
```

Tale metodo, atto a semplificare la realizzazione del metodo *equals()*, restituisce il riferimento dell'oggetto che si trova a *i* posizioni dalla testa della pila.

Una prima realizzazione, dove gli oggetti sono memorizzati negli elementi di un array, è costituita dalla classe *PilaArray2*, che estende *PilaArray* e implementa l'interfaccia *Pila2*:

```
public class PilaArray2 extends PilaArray
                        implements Pila2
{ public PilaArray2(int n)
  { super(n);}
  public Object getElem(int i)
  { Object result = stack[top - i];
    return result;
  }
  public boolean equals(Object o)
  { if(o == null || !(o instanceof Pila2))
      return false;
    Pila2 p = (Pila2) o;
    int n1 = size(), n2 = p.size();
    if(n1 != n2) return false;
    for(int i=0; i<n1; i++)
      if(!getElem(i).equals(p.getElem(i)))
        return false;
    return true;
  }
}
```

Una seconda realizzazione, dove gli oggetti sono memorizzati negli elementi di una lista, è costituita dalla classe *PilaLista2*, che estende la classe *PilaLista* e implementa l'interfaccia *Pila2*:

```
public class PilaLista2 extends PilaLista
                        implements Pila2
```

```
{ public Object getElem(int i)
  { Elem tmp = testa;
    for(int j=0; j<i; j++)
      tmp = tmp.getNext();
    Object result = tmp.getInfo();
    return result;
  }
  public boolean equals(Object o)
  { if(o == null || !(o instanceof Pila2))
      return false;
    Pila2 p = (Pila2) o;
    int n1 = size(), n2 = p.size();
    if(n1 != n2) return false;
    for(int i=0; i<n1; i++)
      if(!getElem(i).equals(p.getElem(i)))
        return false;
    return true;
  }
}
```

Le due classi *PilaArray2* e *PilaLista2* implementano la stesa interfaccia *Pila2*. Pertanto, è stato possibile ridefinire, nelle due classi, il metodo *equals()* (in funzione del rispettivo metodo *getElem()*) nello stesso modo, duplicando semplicemente il codice. Grazie a questo fatto, si possono confrontare pile basate su array con pile basate su liste con uno qualunque dei due metodi *equals()*, come mostrato dal seguente programma:

```
public class ProvaPila2
{ public static void main(String[] args)
  { Pila pa = new PilaArray2(5);
    Pila pl = new PilaLista2();
    pa.push("elem1");
    pa.push("elem2");
    pa.push("elem3");
    Console.scriviStringa("pa.equals(pl): " +
                          pa.equals(pl));
    pl.push("elem1");
    pl.push("elem2");
    Console.scriviStringa("pa.equals(pl): " +
                          pa.equals(pl));
    pl.push("elem3");
    Console.scriviStringa("pa.equals(pl): " +
```

```
                                pa.equals(pl));
    pa.pop();
    Console.scriviStringa("pl.equals(pa): " +
                                pl.equals(pa));
    pl.pop();
    Console.scriviStringa("pl.equals(pa): " +
                                pl.equals(pa));
  }
}
```

Eseguendo il programma si ottiene la seguente uscita:

```
pa.equals(pl): false
pa.equals(pl): false
pa.equals(pl): true
pl.equals(pa): false
pl.equals(pa): true
```

5.4. Tipi coda

Una coda è un insieme ordinato di dati di uguale tipo in cui è possibile eseguire operazioni di inserimento e di estrazione secondo la seguente regola di accesso: il primo dato inserito è anche il primo ad essere estratto (regola di accesso *FIFO*: *First In First Out*).

Il tipo di dato astratto coda di *Object* supporta le operazioni specificate dalla seguente interfaccia:

```
public interface Coda
{  void insert(Object e);
   Object extract();
   boolean isEmpty();
   boolean isFull();
   int size();
}
```

Il metodo *insert()* inserisce l'oggetto *e* passato come argomento nella coda (il metodo lancia un'eccezione di tipo *CodaFullException* se non è

possibile eseguire l'inserimento perché la coda è piena). Il metodo *extract()* estrae un elemento dalla coda e ne restituisce il riferimento (lancia un'eccezione di tipo *CodaEmptyException* quando la coda è vuota). I metodi *isEmpty()* e *isFull()* possono essere usati per determinare se la coda è vuota oppure è piena, mentre *size()* restituisce il numero di oggetti contenuti nella coda.

Le due classi di eccezione possono essere definite come segue:

```
public class CodaFullException
               extends RuntimeException {}

public class CodaEmptyException
               extends RuntimeException {}
```

Poiché si tratta di eccezioni di tipo *unchecked*, le clausole *throws CodaFullException* e *throws CodaEmptyException* dei metodi *insert()* e *extract()* possono essere omesse.

5.4.1. Coda realizzata con array

Una coda può essere costituita da un array circolare, da due indici (*front* e *back*) e da un contatore che indica quanti dati sono contenuti nella coda.

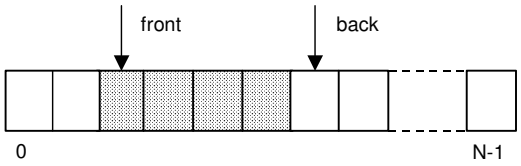


Figura 5.2. Una coda realizzata con un array di dimensione N, in cui sono contenuti quattro valori

L'array memorizza i dati nelle sue componenti, ed è circolare in quanto l'ultimo elemento è visto come adiacente al primo. L'indice *back* individua in ogni istante la posizione relativa al prossimo inserimento, mentre l'indice *front* individua la posizione relativa alla prossima estrazione (fig.

5.2). Dopo ciascuna delle due operazioni il corrispondente indice viene incrementato in modo circolare, mentre il contatore viene incrementato o decrementato a seconda che si sia aggiunto o rimosso un dato. La classe *CodaArray* realizza tali funzioni come segue:

```
public class CodaArray implements Coda
{ Object[] buffer;
  int front, back, cont;
  static int DEFAULT_SIZE = 10;
  public CodaArray(int size)
  { buffer = new Object[size]; }
  public CodaArray()
  { this(DEFAULT_SIZE); }
  public int size()
  { return cont; }
  public boolean isEmpty()
  { return cont == 0; }
  public boolean isFull()
  { return cont == buffer.length; }
  public void insert(Object o)
  { if (isFull())
      throw new CodaFullException();
    buffer[back] = o;
    back = (back + 1) % buffer.length;
    cont++;
  }
  public Object extract()
  { if (isEmpty())
      throw new CodaEmptyException();
    Object result = buffer[front];
    front = (front + 1) % buffer.length;
    cont--;
    return result;
  }
  public String toString()
  { String s = "| ";
    for (int i=0; i<cont; i++)
      s += buffer[(front + i) % buffer.length] + " ";
    s += "|";
    return s;
  }
}
```

5.4.2. Coda realizzata con lista

Una coda può anche essere realizzata utilizzando una lista per la memorizzazione dei dati. Le operazioni di inserimento nella coda e estrazione dalla coda possono essere implementate, rispettivamente, come un inserimento in fondo alla lista e una estrazione dell'elemento in testa alla lista. Per non dover scorrere tutta la lista per eseguire una operazione di inserimento, viene mantenuto un riferimento all'elemento in fondo (*back*), oltre che all'elemento in testa (*front*).

Una realizzazione che rispetta le specifiche anzidette è rappresentata dalla classe *CodaLista*:

```
public class CodaLista implements Coda
{ static class Elem
  { private Object o;
    private Elem next;
    Elem(Object e, Elem n)
    { o = e; next = n;  }
    Object getInfo()
    { return o;  }
    Elem getNext()
    { return next;  }
    void setNext(Elem n)
    { next = n;  }
  }
  Elem back, front;
  int cont;
  public boolean isEmpty()
  { return cont == 0; }
  public boolean isFull()
  { return false; }
  public int size()
  { return cont;  }
  public void insert(Object o)
  { Elem e = new Elem(o, null);
    if(back != null) back.setNext(e);
    back = e;
    if(front == null) front = e;
    cont++;
  }
  public Object extract()
  { if(isEmpty())
```

```
      throw new CodaEmptyException();
    Object o = front.getInfo();
    front = front.getNext();
    if(front == null) back = null;
    cont--;
    return o;
  }
  public String toString()
  { String s = "| ";
    Elem tmp = front;
    while(tmp != null)
    { s += tmp.getInfo() + " "; tmp = tmp.getNext(); }
    s += "|";
    return s;
  }
}
```

5.5. Tipi tabella

Una tabella è un aggregato di elementi dello stesso tipo, dove ogni elemento è costituito da un campo chiave, unico all'interno della tabella, e da un campo informazione. Su una tabella possono essere eseguite operazioni di inserimento, di ricerca e di estrazione. L'inserimento di un nuovo elemento avviene specificando sia la chiave che il contenuto informativo da memorizzare; la posizione del nuovo elemento all'interno della tabella dipende esclusivamente da proprietà della chiave. La ricerca di un elemento avviene specificando la sola chiave e l'operazione viene eseguita confrontando tale chiave con quella degli elementi presenti nella tabella: in caso di uguaglianza, il risultato è costituito dall'informazione presente nell'elemento trovato. L'estrazione consiste nella ricerca di una determinata chiave e nella rimozione dell'intero elemento corrispondente dalla tabella.

Il numero di chiavi che è possibile utilizzare nelle operazioni di inserimento, estrazione e ricerca è in genere estremamente elevato, mentre il numero di chiavi che vengono effettivamente utilizzate è sensibilmente più piccolo. Per esempio, se le chiavi sono gli identificatori di un

linguaggio di programmazione, i possibili identificatori sono molti, mentre quelli effettivamente usati in un programma sono molti di meno: se supponiamo per semplicità che un identificatore sia costituito da esattamente 8 caratteri alfabetici, il numero dei possibili identificatori è pari a 52^8 (26 lettere minuscole più 26 lettere maiuscole), mentre il numero degli identificatori usati in un programma di medie dimensioni è sensibilmente più piccolo.

Una tabella può essere realizzata usando un array che memorizza nelle sue componenti gli elementi della tabella. La dimensione *N* dell'array determina la capacità della tabella, che è in genere molto più piccola del numero di possibili chiavi. La componente dell'array da utilizzare per una operazione di inserimento, ricerca o estrazione viene determinata sfruttando una opportuna funzione *h*, detta funzione *hash*, che a partire da una chiave *k* genera un valore compreso tra *0* e *N-1*. Nel caso in cui *N* sia una potenza di due, supponiamo $N=2^r$, la funzione *h* deve generare una configurazione di *r* bit. Le funzioni hash più comuni possono essere suddivise nelle seguenti categorie (in ordine di casualità crescente).

- La funzione hash restituisce *r* bit della chiave. La scelta dei bit da restituire può influire sulla casualità del valore generato. Per esempio, nel caso in cui le chiavi siano degli identificatori, è conveniente che tali bit non siano in posizione iniziale, in modo da generare valori diversi anche quando le chiavi hanno i primi caratteri uguali. In questo modo vengono generati valori diversi anche quando vengono usati identificatori come *alfa1*, *alfa2*, *alfa3*, che differiscono solo per il carattere finale (come è abitudine comune).
- La sequenza di bit della chiave viene manipolata attraverso opportune funzioni. Per esempio, la chiave può essere divisa in gruppi di *r* bit e con tali gruppi viene eseguita una funzione logica (spesso l'OR esclusivo bit a bit).
- La sequenza di bit viene manipolata in modo più complesso, per esempio viene elevata al quadrato, quindi viene selezionato un gruppo di *r* bit in posizione opportuna.

Nel caso in cui la dimensione della tabella non sia una potenza di due, il numero ottenuto con una delle tecniche appena illustrate viene ulteriormente diviso per *N*, ottenendo un resto *m* compreso tra *0* e *N-1* che costituisce il risultato finale della funzione hash.

Quando si esegue una operazione di inserimento usando una chiave k_i tale che $h(k_i)=m$, può accadere che la componente di indice *m* dell'array

sia già occupata da un altro elemento, per esempio perché in precedenza è stato eseguito un altro inserimento con chiave k_j tale che $h(k_j)=m$ (quando si verifica questa condizione si dice che si è avuta una collisione). In questo caso è necessario inserire l'elemento con chiave k_i in una posizione libera con indice diverso da m. Notare che la componente di indice m può risultare occupata anche se non si è verificata una collisione, per esempio perché con la precedente regola vi è già stato un inserimento.

Le componenti libere dell'array vengono contrassegnate usando un particolare valore. Gli indici delle componenti da visitare nel caso in cui la posizione m-esima sia occupata possono essere generati calcolando il valore $(m + p(s)) \% N$, con $s=1, 2,$ eccetera, e fermandosi quando si incontra una componente libera o si ottiene nuovamente l'indice m. La funzione $p(s)$ dipende dalla regola di scansione: quelle più comunemente usate sono quella lineare e quella quadratica.

Scansione lineare: la quantità $p(s)$ è pari a $q*s$, dove q è una costante intera detta passo di scansione. E' opportuno scegliere q in modo che sia primo con N, così da visitare tutte le componenti dell'array prima di tornare a quella iniziale di indice m. Un caso particolare si ha quando $q=1$ (scansione unitaria). La scansione lineare può produrre la formazione di agglomerati, ossia di gruppi di elementi distanti q. A causa di essi, chiavi con valori hash diversi, ma interni ad uno stesso agglomerato producono da un certo punto in poi i medesimi indici, annullando il vantaggio della diversa posizione iniziale e allungando notevolmente i tempi di accesso.

Scansione quadratica: la quantità $p(s)$ è pari a s^2. La sequenza di indici che viene generata può non toccare tutte le componenti dell'array, e questo può portare a considerare piena una tabella anche quando non lo è. Se N è un numero primo, oltre alla posizione iniziale vengono coinvolte altre $(N-1)/2$ componenti dell'array.

Gli indici da usare per accedere all'array durante le operazioni di ricerca e di estrazione sono generati allo stesso modo. Per quanto riguarda l'operazione di estrazione è opportuno notare che non è possibile rimpiazzare l'elemento da eliminare con il valore che identifica una posizione libera. Infatti, una successiva operazione di ricerca potrebbe essere interrotta prematuramente quando viene incontrata tale componente, dando esito negativo anche se l'elemento coinvolto nella ricerca è effettivamente presente nella tabella. Pertanto il valore "elemento libero" deve essere diverso dal valore "elemento rimosso".

Un tipo di dato costituito da tabelle in cui le chiavi sono di tipo stringa, le informazioni di tipo *Object* e in cui siano possibili operazioni di inserimento, estrazione e ricerca, può essere ottenuto con la classe *TabellaHash* nel seguente modo:

```
public class InsertException extends RuntimeException
{ public InsertException(String msg)
  { super(msg); }
}

public class TabellaHash
{ static class Elem
  { private Object info;
    private String key;
    Elem(String k, Object o)
    { key = k; info = o;  }
    Object getInfo()
    { return info;  }
    String getKey()
    { return key; }
    public String toString()
    { return "k=" + key + ", o=" + info; }
  }
  static final Elem EMPTY = new Elem("EMPTY", null);
  static final Elem REMOVED = new Elem("REM", null);
  Elem[] tab;
  public TabellaHash(int n)
  { tab = new Elem[n];
    for(int i=0; i<tab.length; i++) tab[i] = EMPTY;
  }
  public int size()
  { return tab.length;  }
  int hash(String k)
  { char[] c = k.toCharArray();
    int somma = 0;
    for(int i=0; i<c.length; i++) somma += c[i]*c[i];
    return somma % tab.length;
  }
  int p(int s)
  { return s;  }
  public boolean insert(String k, Object o)
  { if(o == null)
      throw new InsertException("Inserimento " +
```

```
                                    "del valore null");
    if(find(k) != null)
      return false;
    int m = hash(k);
    int n = m, i = 1;
    while(tab[n] != EMPTY && tab[n] != REMOVED)
    { n = (m + p(i)) % tab.length;
      i++;
      if (n == m)
        throw new InsertException("Tabella piena");
    }
    tab[n] = new Elem(k, o);
    return true;
  }
  public Object find(String k)
  { int m = hash(k);
    int n = m, i = 1;
    while(tab[n] == REMOVED || tab[n] != EMPTY &&
          !tab[n].getKey().equals(k))
    { n = (m + p(i)) % tab.length;
      i++;
      if (n == m) break;
    }
    if(tab[n] != EMPTY && tab[n].getKey().equals(k))
      return tab[n].getInfo();
    else return null;
  }
  public Object extract(String k)
  { int m = hash(k);
    int n = m, i = 1;
    Object o = null;
    while(tab[n] == REMOVED || tab[n] != EMPTY &&
          !tab[n].getKey().equals(k))
    { n = (m + p(i)) % tab.length;
      i++;
      if (n == m) break;
    }
    if(tab[n] != EMPTY && tab[n].getKey().equals(k))
    { o = tab[n].getInfo(); tab[n] = REMOVED; }
    return o;
  }
  public String toString()
  { String s = "";
```

```
    for(int i=0; i<tab.length; i++)
      s+="[" + i + "] " + tab[i] + "\n";
    return s;
  }
}
```

La funzione hash è data dal resto della divisione fra la somma dei quadrati delle codifiche dei singoli caratteri della chiave e il numero degli elementi della tabella. La scansione è lineare con passo *1*. Il costruttore consente di specificare la capacità della tabella, che può essere successivamente riottenuta attraverso il metodo *size()*. Il metodo *insert()* inserisce l'informazione *o* nella tabella associandola alla chiave *k*. Tale metodo restituisce *true* se il nuovo elemento viene inserito correttamente, *false* se la chiave era già presente in tabella; inoltre, il metodo lancia un'eccezione di tipo *InsertException* nel caso in cui *o* valga *null*, oppure se la tabella è piena. Il metodo *find()* restituisce l'informazione memorizzata nella tabella se esiste un elemento che ha la chiave *k*, *null* altrimenti. Il metodo *extract()* restituisce l'informazione associata alla chiave *k* se presente nella tabella, *null* altrimenti; il metodo inoltre rimuove l'elemento corrispondente dalla tabella (nel caso in cui esista).

Per illustrare l'uso della classe *TabellaHash* riportiamo il seguente programma, che effettua inserimenti, ricerche ed estrazioni di *Integer* in una tabella di dimensione 10:

```
public class ProvaTabella
{ private static void stampaMenu()
  { Console.scriviStringa("1) Inserimento");
    Console.scriviStringa("2) Ricerca");
    Console.scriviStringa("3) Estrazione");
    Console.scriviStringa("4) Stampa");
    Console.scriviStringa("5) Esci");
  }
  public static void main(String[] args)
  { TabellaHash ht = new TabellaHash(10);
    String kk; Integer ii; int i;
    Console.scriviStringa("Creata una tabella di " +
                          "dimensione " +ht.size());
    for(;;)
    { try
      { stampaMenu();
        int x = Console.leggiIntero();
```

```
switch(x)
{ case 1:
    Console.scriviStringa("Inserisci la " +
                    "chiave (una stringa): ");
    kk = Console.leggiStringa();
    Console.scriviStringa("Inserisci " +
                          "un intero: ");
    i = Console.leggiIntero();
    if(ht.insert(kk, new Integer(i)))
      Console.scriviStringa("Elemento " +
                            "inserito");
    else
      Console.scriviStringa("Chiave gia' " +
                              "presente");
    break;
  case 2:
    Console.scriviStringa("Inserisci la " +
                    "chiave (una stringa): ");
    kk = Console.leggiStringa();
    ii = (Integer) ht.find(kk);
    if(ii != null)
      Console.scriviStringa("Informazione: "
                              + ii);
    else
      Console.scriviStringa("Valore " +
                           "non presente");
    break;
  case 3:
    Console.scriviStringa("Inserisci la " +
                    "chiave (una stringa): ");
    kk = Console.leggiStringa();
    ii = (Integer) ht.extract(kk);
    if(ii != null)
      Console.scriviStringa("Informazione: "
                              + ii);
    else
      Console.scriviStringa("Valore non " +
                              "presente");
    break;
  case 4:
    Console.scriviStringa("Tabella:\n" + ht);
    break;
  case 5:
```

```
                return;
            }
        }
        catch (InsertException e)
        { Console.scriviStringa("Errore: " +
                                    e.getMessage()); }
    }
  }
}
```

5.6. Tipi albero

Un albero è costituito da un insieme finito di *nodi* contenenti informazioni di un dato tipo, e da un insieme di *archi orientati* (ossia caratterizzati da un senso di percorrenza) con le seguenti proprietà:

- un particolare nodo costituisce la *radice*;
- esiste un percorso (una sequenza di archi) e uno solo tra la radice e un qualunque altro nodo.

Un albero è dunque un grafo aciclico in cui dalla radice è possibile raggiungere ogni altro nodo. Ogni nodo di un albero risulta essere radice di un sottoalbero, che ha a sua volta zero o più sottoalberi.

Tutti i nodi hanno un unico arco entrante, con l'esclusione del nodo radice che non ha archi entranti, mentre ogni nodo può avere zero o più archi uscenti. Un nodo privo di archi uscenti è detto *foglia* (o nodo terminale). Gli archi determinano relazioni padre-figlio: il nodo da cui l'arco parte è detto nodo *padre*, mentre il nodo raggiunto è detto nodo *figlio*. Nodi figli dello stesso padre sono detti *fratelli*. I nodi di un albero sono suddivisi in *livelli*: il livello della radice è supposto essere uguale ad uno, mentre il livello di un qualunque altro nodo è pari al livello del padre più uno.

Strutture dati di questo tipo sono comunemente impiegate nel mondo informatico. Per esempio, il file-system dei sistemi operativi della famiglia UNIX è organizzato ad albero: ogni directory corrisponde ad un nodo, e i file e i sottodirectory in esso contenuti sono rappresentati come nodi figli.

Esiste inoltre un directory, identificato dal simbolo "/", che corrisponde al nodo radice.

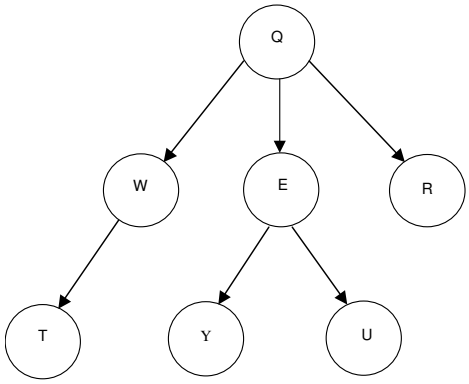


Figura 5.3. Un albero ordinato

Un albero è detto *ordinato* se esiste un ordinamento lineare per i figli di ogni nodo, in modo tale che sia possibile identificare i figli di un dato nodo come il primo, il secondo, e così via. Da un punto di vista grafico, la relazione di ordinamento tra fratelli viene spesso espressa disponendo i nodi l'uno accanto all'altro, in modo corrispondente al loro ordine. Per esempio, nell'albero di fig. 5.3, in cui l'informazione contenuta in ogni nodo è un carattere, il nodo associato al carattere *Q* ha tre figli: il primo è quello contenente il carattere *W*, il secondo quello contenente il carattere *E* e il terzo quello contenente il carattere *R*.

In un albero ordinato spesso si inseriscono o eliminano nodi con regole semplici: una tipica operazione è quella che consente di inserire un nuovo nodo con una data informazione, come ulteriore figlio di un padre con una informazione esistente.

Una operazione frequentemente eseguita su un albero ordinato è la visita, cioè l'esame (completo o meno) dei vari nodi in un ordine prestabilito. Lo scopo di questa operazione può essere la stampa delle informazioni associate ai vari nodi o la ricerca di un nodo specifico dove effettuare un inserimento o una estrazione. Le due forme di visita più spesso utilizzate sono descritte di seguito (in forma ricorsiva).

Visita (completa) in ordine anticipato

1. Esamina la radice;
2. se il numero dei sottoalberi della radice è maggiore di zero, allora:
    - visita il primo sottoalbero in ordine anticipato;
    - visita il secondo albero in ordine anticipato;
    - ...
    - visita l'ultimo sottoalbero in ordine anticipato.

Visita (completa) in ordine differito

1. Se il numero dei sottoalberi della radice è maggiore di zero, allora:
    - visita il primo sottoalbero in ordine differito;
    - visita il secondo albero in ordine differito;
    - ...
    - visita l'ultimo albero in ordine differito;
2. esamina la radice.

Con riferimento all'albero di fig. 5.3, la visita in ordine anticipato dell'albero dà come risultato QWTEYUR, mentre quella in ordine differito dà come risultato TWYUERQ.

I nodi di un albero possono essere rappresentati attraverso una classe *Nodo* dotata di due campi, contenenti rispettivamente l'informazione associata al nodo (*info* di tipo *Object*) e la lista dei nodi figli (*figli* di tipo *Lista*). Un albero è quindi identificato da un riferimento all'oggetto *Nodo* che costituisce la radice. Con questo tipo di rappresentazione, l'albero di Fig. 5.3 viene rappresentato come mostrato in Fig. 5.4.

In accordo a questa modalità realizzativa, il tipo di dato astratto albero di *Object* può essere definito dalla classe *Albero* come segue:

```
public class Albero
{  static class Nodo
   {  private Object info;
      private Lista figli;
      Nodo(Object o)
      {  info = o; figli = new ListaSemplice2();  }
      Object getInfo()
      {  return info;  }
```

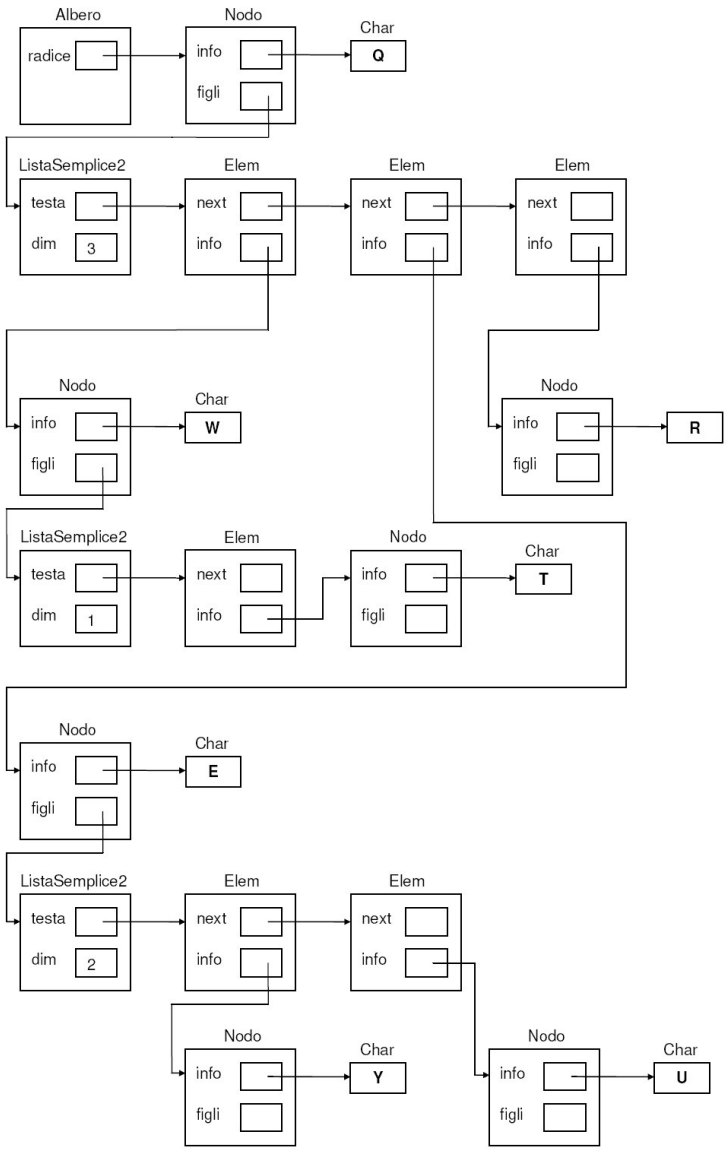


Figura 5.4. Rappresentazione dell'albero di Fig. 5.3

```
    int quantiFigli()
    { return figli.size();}
    Nodo getFiglio(int pos)
    { return (Nodo) figli.getPos(pos);  }
    Lista getFigli()
    { return figli; }
    Object getInfoFiglio(int pos)
    { return getFiglio(pos).getInfo();  }
    void nuovoFiglio(Object o)
    { figli.insFondo(new Nodo(o)); }
    public String toString()
    { return info.toString();  }
  }
  Nodo radice;
  public Albero()
  {}
  public Albero(Object o)
  { radice = new Nodo(o);  }
  public boolean insert(Object o, Object infoPadre)
  { if(radice == null)
    { radice = new Nodo(o); return true;  }
    return ins(radice, o, infoPadre);
  }
  private boolean ins(Nodo n, Object o, Object infoP)
  { if(n.getInfo().equals(infoP))
    { n.nuovoFiglio(o); return true;  }
    for(int i=0; i<n.quantiFigli(); i++)
      if(ins(n.getFiglio(i), o, infoP))
        return true;
    return false;
  }
  public void voa()
  { voa(radice); Console.nuovaLinea();  }
  private void voa(Nodo n)
  { if(n != null)
    { Console.scriviStr(n.toString());
      for(int i=0; i<n.quantiFigli(); i++)
        voa(n.getFiglio(i));
    }
  }
  public void vod()
  { vod(radice); Console.nuovaLinea();  }
  private void vod(Nodo n)
```

```
  { if(n != null)
    { for(int i=0; i<n.quantiFigli(); i++)
        vod(n.getFiglio(i));
      Console.scriviStr(n.toString());
    }
  }
}
```

Il costruttore default crea un albero vuoto, mentre il costruttore dotato di un argomento crea un albero composto da un solo nodo in cui è memorizzata l'informazione passata al costruttore. La lista dei nodi figli è realizzata usando il tipo *ListaSemplice2* definito in precedenza. Il metodo *insert()* consente di specificare, oltre all'informazione da inserire nell'albero, l'informazione del padre del nuovo nodo. I metodi *voa()* e *vod()* stampano le informazioni contenute nell'albero usando rispettivamente una strategia di visita in ordine anticipato e differito.

Come esempio di utilizzo della classe *Albero* riportiamo il seguente programma:

```
public class ProvaAlbero
{ public static void main(String[] args)
  { Console.scriviStringa(
      "Per ogni inserimento specificare " +
      "una coppia di stringhe a e b:\n" +
      "a e' l'informazione da inserire;\n" +
      "b quella del nodo padre (non significativa " +
      "per il primo inserimento).\n" +
      "Gli inserimenti terminano specificando come " +
      "prima stringa \"fine\".");
    String x, y;
    Albero a = new Albero();
    while(!(x = Console.leggiStr()).equals("fine"))
    { y = Console.leggiStr();
      if(a.insert(x, y))
        Console.scriviStringa("Inserimento eseguito");
      else
        Console.scriviStringa("Informazione " + y +
                              " non esistente");
    }
    Console.scriviStringa("Ordine anticipato:");
    a.voa();
    Console.scriviStringa("Ordine differito:");
```

```
    a.vod();
  }
}
```

Un esempio di esecuzione del programma è il seguente:

```
Per ogni inserimento specificare una coppia di stringhe
a e b:
a e' l'informazione da inserire;
b quella del nodo padre (non significativa per il primo
inserimento).
Gli inserimenti terminano specificando come prima
stringa "fine".
a a
Inserimento eseguito
b a
Inserimento eseguito
c a
Inserimento eseguito
d b
Inserimento eseguito
e b
Inserimento eseguito
fine
Ordine anticipato:
a b d e c
Ordine differito:
d e b c a
```

5.6.1. Alberi binari

Un albero binario è costituito da un insieme di nodi contenenti informazioni di un dato tipo, e da un insieme di archi orientati (aventi un senso di percorrenza), con le seguenti proprietà:

- l'albero può essere vuoto;
- un particolare nodo costituisce la radice;
- esiste un percorso (sequenza di archi) e uno solo tra la radice e un qualunque altro nodo;
- ogni nodo ha due sottoalberi (due figli), uno sinistro e uno destro (di cui uno o entrambi possono essere vuoti).

Un nodo che ha entrambi i sottoalberi vuoti è una foglia. Un albero binario è sempre ordinato, poiché risulta prestabilito l'ordine dei figli di ciascun nodo. Un albero binario è riportato in Fig. 5.5.

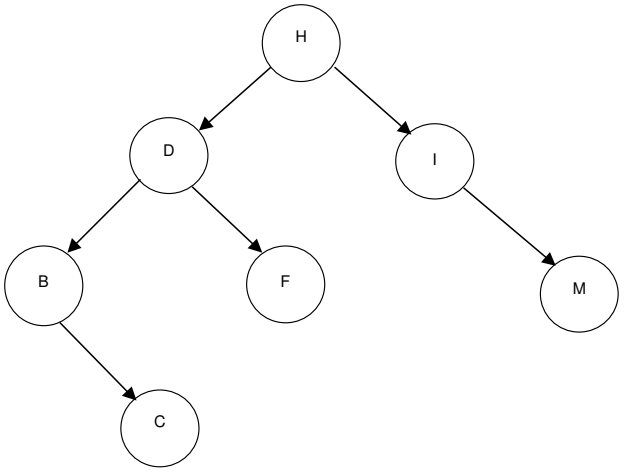


Figura 5.5. Un albero binario

La definizione di albero binario differisce da quella di albero ordinato, in quanto in un albero ordinato un padre non ha mai un figlio (in posizione iniziale o intermedia) costituito da un sottoalbero vuoto. Per esempio, l'albero binario di fig. 5.5 non corrisponde alla definizione di albero.

Un albero binario può anche essere definito ricorsivamente: esso è un insieme finito di nodi che, se non è vuoto, è costituito da un nodo che rappresenta la radice e da due sottoinsiemi disgiunti di nodi che sono a loro volta alberi binari (sottoalbero destro e sottoalbero sinistro).

In un albero binario la visita dei nodi, oltre che in ordine anticipato e differito, può essere eseguita in ordine simmetrico.

Visita (completa) in ordine anticipato

Se l'albero non è vuoto allora:

- esamina la radice;
- visita il sottoalbero sinistro in ordine anticipato;

- visita il sottoalbero destro in ordine anticipato.

Per l'albero di fig. 5.5 questa visita dà come risultato la sequenza HDBCFIM.

Visita (completa) in ordine differito

Se l'albero non è vuoto allora:
- visita il sottoalbero sinistro in ordine differito;
- visita il sottoalbero destro in ordine differito;
- esamina la radice.

Per l'albero di fig. 5.5 questa visita dà come risultato la sequenza CBFDMIH.

Visita (completa) in ordine simmetrico

Se l'albero non è vuoto allora:
- visita il sottoalbero sinistro in ordine simmetrico;
- esamina la radice;
- visita il sottoalbero destro in ordine simmetrico.

Per l'albero di fig. 5.5 questa visita dà come risultato la sequenza BCDFHIM.

Il tipo di dato astratto albero binario di *Object* può essere realizzato come segue:

```
public class AlberoBin
{ static class Nodo
  { Object info;
    Nodo sin, des;
    Nodo(Object o)
    { info = o;  }
    Object getInfo()
    { return info;  }
    boolean setSin(Nodo s)
    { if(sin == null) { sin = s; return true;  }
      return false;
    }
    boolean setDes(Nodo d)
    { if(des == null) { des = d; return true;  }
      return false;
    }
    Nodo getSin()
    { return sin;}
    Nodo getDes()
    { return des;}
    public String toString()
    { return info.toString();  }
  }
  Nodo radice;
  public AlberoBin()
  {}
  public AlberoBin(Object o)
  { radice = new Nodo(o);  }
  public boolean insertSin(Object o, Object infoPadre)
  { if(radice == null)
    { radice = new Nodo(o); return true;  }
    return insSin(radice, o, infoPadre);
  }
  private boolean insSin(Nodo n, Object o, Object ip)
  { if(n.getInfo().equals(ip))
    { if(n.getSin() != null) return false;
      n.setSin(new Nodo(o));
      return true;
    }
    if(n.getSin() != null &&
        insSin(n.getSin(), o, ip))
      return true;
    if(n.getDes() != null &&
```

```
        insSin(n.getDes(), o, ip))
      return true;
    return false;
  }
  public boolean insertDes(Object o, Object ip)
  { if(radice == null)
    { radice = new Nodo(o); return true;  }
    return insDes(radice, o, ip);
  }
  private boolean insDes(Nodo n, Object o, Object ip)
  { if(n.getInfo().equals(ip))
    { if(n.getDes() != null) return false;
      n.setDes(new Nodo(o));
      return true;
    }
    if(n.getSin() != null &&
        insDes(n.getSin(), o, ip))
      return true;
    if(n.getDes() != null &&
        insDes(n.getDes(), o, ip))
      return true;
    return false;
  }
  public void voa()
  { voa(radice); Console.nuovaLinea();  }
  private void voa(Nodo r)
  { if(r != null)
    { Console.scriviStr(r.toString());
      voa(r.getSin()); voa(r.getDes());
    }
  }
  public void vod()
  { vod(radice); Console.nuovaLinea();  }
  private void vod(Nodo r)
  { if(r != null)
    { vod(r.getSin()); vod(r.getDes());
      Console.scriviStr(r.toString());
    }
  }
  public void vos()
  { vos(radice); Console.nuovaLinea();  }
  private void vos(Nodo r)
  { if(r != null)
```

```
    { vos(r.getSin());
      Console.scriviStr(r.toString());
      vos(r.getDes());
    }
  }
}
```

Un albero binario è *ordinato per contenuto* se, preso un qualunque nodo, la sua informazione è maggiore o uguale di quella di un qualunque nodo del sottoalbero sinistro e minore di quella di un qualunque nodo del sottoalbero destro. L'albero binario di fig. 5.4 è ordinato per contenuto. In un albero binario ordinato per contenuto, la visita in ordine simmetrico dà come risultato le informazioni dei nodi secondo l'ordinamento stesso. Inoltre, è agevole inserire nuove foglie mantenendo l'ordinamento, o ricercare se è presente o meno un nodo con una certa informazione.

Un esempio di albero binario ordinato per contenuto è riportato nel sottoparagrafo 5.7.2.

5.7. Interfaccia *Comparable*

Nel caso dei tipi primitivi, la relazione di ordinamento tra due valori può essere stabilita con semplicità usando gli operatori di confronto e di uguaglianza. Nel caso di oggetti classe tali operatori non possono essere utilizzati, ed è consuetudine ricorrere ad un'interfaccia predefinita, *Comparable* (package *java.lang*), che deve essere implementata da quelle classi che vogliono definire una relazione di ordinamento tra le proprie istanze. L'interfaccia *Comparable* prevede un solo metodo:

public int compareTo (Object o)
restituisce un numero negativo, zero, o un numero positivo a seconda che l'oggetto su cui viene invocato il metodo, rispettivamente preceda, sia uguale a, o segua l'oggetto *o*.

E' in genere opportuno che il metodo *compareTo()* sia implementato in modo da essere consistente con il metodo *equals()*, ossia che

(x.compareTo(y) == 0) restituisca lo stesso valore di *x.equals(y)*. Inoltre, è buona norma che *x.compareTo(y)* restituisca un valore di segno opposto rispetto a *y.compareTo(x)* e che la realizzazione goda della proprietà transitiva, per esempio se *a.compareTo(b) > 0* e *b.compareTo(c) > 0* allora è vero che *a.compareTo(c) > 0*. L'interfaccia *Comparable* è implementata da numerose classi di sistema, tra le quali *Byte*, *Character*, *Short*, *Integer*, *Long*, *Float*, *Double*, e *String*.

A titolo di esempio riportiamo il codice della classe *MioIntero*, che implementa l'interfaccia *Comparable* come segue:

```
public class MioIntero implements Comparable
{ private int value;
  public MioIntero(int v)
  { value = v; }
  public int compareTo(Object o)
  { MioIntero i = (MioIntero) o;
    return value - i.value;
  }
  public String toString()
  { return String.valueOf(value); }
}
```

5.7.1. Esempio: lista ordinata

A questo punto, vediamo come realizzare una lista ordinata in cui l'ordine degli elementi è determinato dalla implementazione del metodo *compareTo()* degli oggetti inseriti nella lista:

```
public class ListaOrd
{ static class ElemOrd implements Comparable
  { private Comparable info;
    private ElemOrd next;
    ElemOrd(Comparable c, ElemOrd n)
    { info = c; next = n; }
    Comparable getInfo()
    { return info; }
    ElemOrd getNext()
    { return next; }
    void setNext(ElemOrd n)
    { next = n;  }
```

```
    public int compareTo(Object o)
    // metodo dichiarato nell'interfaccia Comparable
    { ElemOrd eo = (ElemOrd) o;
      return getInfo().compareTo(eo.getInfo());
    }
  }
  ElemOrd testa;
  public void insOrd(Comparable i)
  { ElemOrd e = new ElemOrd(i, null);
    if(testa == null || e.compareTo(testa) <= 0)
    { testa = e; e.setNext(testa); return;  }
    ElemOrd p = testa, q = testa.getNext();
    while(q != null && e.compareTo(q) > 0)
    { p = q; q = q.getNext();  }
    p.setNext(e); e.setNext(q);
  }
  public Comparable estVal(Comparable v)
  { ElemOrd p = testa, q = testa;
    while(q != null && q.getInfo().compareTo(v) != 0)
    { p = q; q = q.getNext();  }
    if(q == null) return null;
    Comparable c = q.getInfo();
    if(q == testa) testa = q.getNext();
    else p.setNext(q.getNext());
    return c;
  }
  public String toString()
  { String s;
    if(testa == null)
    { s = "<>"; return s;  }
    s = "<" + testa.getInfo();
    ElemOrd p = testa.getNext();
    while(p != null)
    { s += ", " + p.getInfo(); p = p.getNext();  }
    return s + ">";
  }
}
```

Nella lista possono essere inserite istanze di una qualunque classe che implementi l'interfaccia *Comparable*, a patto che i dati inseriti siano tutti dello stesso tipo o che sia esplicitamente definita una relazione di ordinamento tra oggetti appartenenti a classi diverse (per esempio, non è

possibile inserire contemporaneamente nella lista stringhe e oggetti di tipo *MioIntero*).

5.7.2. Esempio: albero binario ordinato per contenuto

Come ulteriore esempio, mostriamo come realizzare un albero binario di oggetti *Comparable* ordinato per contenuto:

```
import java.util.*;
public class AlberoBinOrd extends AlberoBin
// si ereditano i metodi voa(), vos() e vod()
{ public AlberoBinOrd()
  { super(); }
  public AlberoBinOrd(Comparable o)
  { super(o);  }
  public boolean insertSin(Object o, Object ip)
  { throw new UnsupportedOperationException(); }
  public boolean insertDes(Object o, Object ip)
  { throw new UnsupportedOperationException(); }
  public void insertOrd(Comparable o)
  { radice = insOrd(radice, o);  }
  private Nodo insOrd(Nodo n, Comparable o)
  { if(n == null) n = new Nodo(o);
    else if(o.compareTo(n.getInfo()) <= 0)
      n.setSin(insOrd(n.getSin(), o));
    else if(o.compareTo(n.getInfo()) > 0)
      n.setDes(insOrd(n.getDes(), o));
    return n;
  }
  public Nodo find(Comparable o)
  { return fnd(radice, o); }
  private Nodo fnd(Nodo n, Comparable o)
  { if(n == null) return null;
    else if(o.compareTo(n.getInfo()) == 0)
             return n;
    else if(o.compareTo(n.getInfo()) < 0)
             return fnd(n.getSin(), o);
    else return fnd(n.getDes(), o);
  }
}
```

Si noti che la classe *AlberoBinOrd* ridefinisce i metodi *insertSin()* e *insertDes()* ereditati dalla classe *AlberoBin* in modo che generino sempre un'eccezione (*unchecked*) di tipo *UnsupportedOperationException* (package *java.lang*), così che l'inserimento di nuovi valori avvenga sempre mediante il metodo *insertOrd()*, che realizza l'inserimento ordinato.

5.8. Interfaccia *Iterator*

Il package *java.util* include una interfaccia, *Iterator*, da usare per visitare, ed eventualmente rimuovere, gli elementi di una struttura dati usando alcuni metodi standard:

public boolean hasNext ()
restituisce *true* se ci sono elementi non ancora visitati, *false* altrimenti;

public Object next ()
restituisce il riferimento al prossimo elemento (lancia un'eccezione di tipo *NoSuchElementException* se non ci sono altri elementi da visitare);

public void remove ()
rimuove l'elemento che è stato restituito dall'ultima chiamata al metodo *next()* (lancia un'eccezione di tipo *IllegalStateException* se si tenta di rimuovere un elemento senza aver mai chiamato il metodo *next()* o se, dopo aver rimosso un elemento, si tenta di rimuoverne un secondo senza aver invocato *next()*); l'implementazione del metodo *remove()* è opzionale, e quando non è supportato lancia un'eccezione di tipo *UnsupportedOperationException*.

Le eccezioni di tipo *IllegalStateException* e di tipo *UnsupportedOperationException* appartengono alla categoria delle eccezioni *unchecked* e pertanto non è obbligatoria una loro esplicita gestione.

A titolo di esempio, vediamo come estendere la classe *ListaSemplice2* vista in precedenza in modo da dotarla di un metodo *iterator()* che

restituisce un oggetto di tipo *Iterator*, che consente di scorrere la lista e rimuovere elementi:

```
import java.util.*;
public class ListaIterabile extends ListaSemplice2
{  class IteratoreLista implements Iterator
   {  private Elem returned;
      private Elem prev;
      private boolean ancora, rem;
      IteratoreLista()
      {  ancora = !isEmpty();
         rem = true;
      }
      public boolean hasNext()
      {  return ancora;
      }
      public Object next()
      {  if(!ancora) throw new IllegalStateException();
         prev = returned;
         if(returned == null) returned = testa;
         else returned = returned.getNext();
         ancora = returned.getNext() != null;
         rem = false;
         return returned.getInfo();
      }
      public void remove()
      {  if(rem)
            throw new IllegalStateException();
         if(returned == testa)
            testa = testa.getNext();
         else prev.setNext(returned.getNext());
         rem = true;
      }
   }
   public Iterator iterator()
   {  return new IteratoreLista();
   }
   public boolean isEmpty()
   {  return testa == null;
   }
}
```

Le variabili membro della classe *IteratoreLista* tengono traccia dell'ultimo elemento visitato (*returned*), dell'elemento precedente (*prev*), della presenza di altri elementi non visitati (*ancora*), e dell'esecuzione di una precedente operazione di rimozione (*rem*).

A questo punto, per visitare gli elementi di un oggetto della classe *ListaIterabile* ed eventualmente rimuovere un elemento è sufficiente operare come segue:

```
import java.util.*;
public class ProvaListaIterabile
{ public static void main(String[] args)
  { ListaIterabile li = new ListaIterabile();
    li.insTesta("3");
    li.insTesta("2");
    li.insTesta("1");
    Iterator it = li.iterator();
    while(it.hasNext())
    { Object o = it.next();
      Console.scriviStringa("Elem: " + o);
    }
    it = li.iterator();
    while(it.hasNext())
    { Object o = it.next();
      if(o.equals("2"))
      { Console.scriviStringa("Rimuovo: " + o);
        it.remove();
      }
    }
    Console.scriviStringa("li: " + li);
  }
}
```

Eseguendo il programma si ottiene la seguente uscita:

```
Elem: 1
Elem: 2
Elem: 3
Rimuovo: 2
li: <1, 3>
```

5.9. Java collection framework

Il linguaggio Java dispone di una libreria standard, nota con il nome di *collection framework*, che semplifica la gestione di un gruppo di dati visti come una singola entità (detta *collezione*). Il collection framework include la definizione di tipi di dati astratti quali insiemi, liste, tabelle ed alberi.

Il collection framework si compone principalmente di:

- interfacce che rappresentano i diversi tipi di dato astratto e che definiscono l'insieme di operazioni che è possibile compiere su di essi;
- realizzazioni di carattere generale di tali interfacce;
- implementazioni di algoritmi di utilità generale, per esempio di ordinamento o di ricerca del valore massimo di un insieme.

La separazione tra interfacce e realizzazioni consente al programmatore di passare da una realizzazione ad un'altra con poco sforzo. Questo permette, per esempio, di esaminare più realizzazioni della stessa interfaccia per poi scegliere quella più efficiente per il problema in esame.

Da una collezione si può ottenere una *vista*: questa non è una nuova collezione, ma un sottoinsieme di quella originaria: tutte le modifiche apportate alla vista (per esempio la rimozione di un dato) causano una modifica della collezione originaria, e tutte le modifiche apportate alla collezione originaria si ripercuotono sulla vista.

Le interfacce e le classi che fanno parte del collection framework sono contenute nel package *java.util*.

5.10. Collection framework: interfacce

Il collection framework prevede sei interfacce che rappresentano aggregati di oggetti di diversa natura e definiscono le operazioni che è possibile eseguire su di essi: *Collection* che rappresenta un generico aggregato di elementi; *List*, *Set*, e *SortedSet* che estendono l'interfaccia

Collection e che rappresentano rispettivamente una sequenza, un insieme e un insieme ordinato di elementi; *Map* e *SortedMap* che non estendono *Collection* e che rappresentano associazioni tra oggetti.

Le relazioni di ereditarietà tra le interfacce sono sintetizzate in Fig. 5.6.

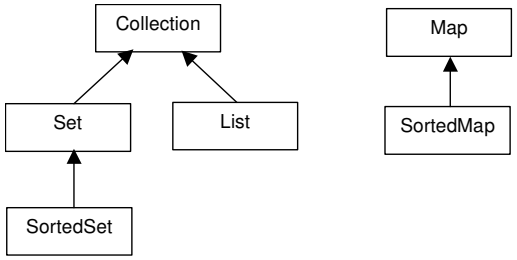


Figura 5.6. Relazioni di ereditarietà tra le interfacce del *collection framework*.

5.10.1. Interfaccia *Collection*

L'interfaccia *Collection* rappresenta un aggregato di oggetti in cui non si fa nessuna assunzione sulla presenza di elementi duplicati, sull'ordinamento degli elementi o la loro posizione.

L'interfaccia *Collection* prevede i seguenti metodi fondamentali:

public int size ()
restituisce il numero di oggetti contenuti nella collezione;

public void clear ()
rimuove tutti gli oggetti contenuti nella collezione;

public boolean isEmpty ()
restituisce *true* se la collezione è vuota, *false* altrimenti;

public Object [] toArray ()
restituisce gli oggetti contenuti nella collezione sotto forma di array;

public boolean add (Object o)
aggiunge l'oggetto *o* alla collezione;

public boolean contains (Object o)
restituisce *true* se l'oggetto *o* è contenuto nella collezione, *false* altrimenti;

public boolean remove (Object o)
rimuove, se presente, l'oggetto *o* dalla collezione

public boolean addAll (Collection c)
aggiunge tutti gli oggetti contenuti in *c* alla collezione corrente;

public boolean containsAll (Collection c)
restituisce *true* se tutti gli oggetti di *c* sono contenuti nella collezione corrente;

public boolean removeAll (Collection c)
rimuove dalla collezione corrente tutti gli oggetti contenuti in *c*;

public boolean retainAll (Collection c)
rimuove dalla collezione corrente tutti gli oggetti, con l'esclusione di quelli contenuti in *c*;

public Iterator iterator ()
restituisce un oggetto di tipo *Iterator* che consente di scorrere gli elementi della collezione.

5.10.2. Interfacce *Set* e *SortedSet*

L'interfaccia *Set* rappresenta il concetto matematico di insieme, ossia di un aggregato di elementi in cui non sono contenuti duplicati. In particolare, in un *Set* può essere contenuto al più un elemento con il valore *null* e non possono esserci due elementi *x* e *y* tali che *x.equals(y)*.

L'interfaccia *Set* presenta gli stessi metodi dell'interfaccia *Collection*, ma i metodi *add()* e *addAll()* hanno significati diversi, in quanto aggiungono all'oggetto *Set* solo gli elementi che non sono già presenti. È opportuno ridefinire il metodo *equals()* in una implementazione dell'interfaccia *Set*, in modo che restituisca il valore *true* quando i due oggetti *Set* contengono elementi uguali (secondo il metodo *equals()* definito nella classe a cui appartengono gli elementi), *false* altrimenti.

L'interfaccia *SortedSet* estende *Set* e rappresenta insiemi in cui è definita una relazione di ordinamento tra gli elementi. Gli oggetti che

vengono aggiunti ad un *SortedSet* devono implementare l'interfaccia *Comparable*, usata dal *SortedSet* per comparare gli elementi.

I metodi principali dell'interfaccia *SortedSet*, oltre a quelli ereditati da *Set*, sono:

public Object first ()
restituisce il primo elemento (il più piccolo in base alla relazione di ordinamento) dell'insieme;

public Object last ()
restituisce il più grande elemento (in base alla relazione di ordinamento) dell'insieme;

public SortedSet headSet (Object o)
restituisce come insieme ordinato una vista costituita da tutti gli elementi strettamente minori di *o*;

public SortedSet tailSet (Object o)
restituisce come insieme ordinato una vista costituita da tutti gli elementi maggiori o uguali ad *o*;

public SortedSet subset (Object x , Object y)
restituisce come insieme ordinato una vista costituita da tutti gli elementi compresi tra *x* (incluso) e *y* (escluso).

public Iterator iterator ()
restituisce un oggetto di tipo *Iterator* che consente di scorrere gli elementi del *SortedSet* .in modo ordinato.

5.10.3. Interfaccia *List*

L'interfaccia *List* rappresenta una collezione di elementi in cui ad ognuno è associato un indice che ne identifica la posizione (il primo elemento ha indice *0*). Le operazioni di inserimento e di estrazione possono specificare l'indice dell'elemento da aggiungere o rimuovere dalla collezione. Il numero di elementi contenuti in un oggetto *List* può variare nel tempo, ed è consentito che siano presenti più elementi uguali.

Oltre ai metodi dell'interfaccia *Collection*, l'interfaccia *List* prevede (tra gli altri) i seguenti metodi principali:

public Object get (int i)
restituisce l'oggetto in posizione *i*;

public Object set (int i , Object o)
rimpiazza l'oggetto in posizione *i* con quello specificato da *o*;

public void add (int i , Object o)
inserisce nella posizione specificata da *i* l'oggetto *o*;

public Object remove (int i)
rimuove l'oggetto in posizione *i*;

public int indexOf (Object o)
restituisce l'indice della prima occorrenza dell'oggetto *o*, *-1* nel caso in cui l'oggetto non sia presente;

public int lastIndexOf (Object o)
restituisce l'indice dell'ultima occorrenza dell'oggetto *o*, *-1* nel caso in cui l'oggetto non sia presente;

public List subList (int i , int j)
restituisce una vista che contiene gli elementi con indici che vanno da *i*, incluso, a *j*, escluso.

Le classi che implementano l'interfaccia *List* devono ridefinire il metodo *equals()* in modo tale che restituisca *true* quando tutti gli elementi in posizioni corrispondenti sono uguali, *false* altrimenti. Due elementi sono considerati uguali se entrambi hanno valore *null* oppure se il metodo *equals()* applicato agli elementi restituisce *true*.

5.10.4. Interfacce *Map* e *SortedMap*

L'interfaccia *Map* prevede metodi che consentono di operare su un tipo di dato astratto simile al tipo tabella visto in precedenza. In particolare, la struttura dati descritta dall'interfaccia *Map* è un aggregato di elementi costituiti da una chiave e da un valore (o campo informativo), dove sia la chiave che il valore sono di tipo *Object*. L'inserimento avviene specificando sia la chiave che il valore da memorizzare, mentre le

operazioni di ricerca e di estrazione avvengono specificando la sola chiave. Le chiavi devono essere uniche all'interno della struttura dati.

L'interfaccia prevede i seguenti metodi fondamentali:

public int size ()
restituisce il numero di elementi (coppie chiave-valore) presenti nella struttura dati;

public void clear ()
rimuove tutti gli elementi dalla struttura dati;

public boolean isEmpty ()
restituisce *true* se la struttura dati è vuota, *false* altrimenti;

public boolean containsKey (Object k)
restituisce *true* se la struttura dati contiene una chiave uguale a *k*, *false* altrimenti (il confronto avviene invocando il metodo *equals()* sulle chiavi); se *k* ha valore *null*, il metodo restituisce *true* se c'è una chiave che ha tale valore;

public boolean containsValue (Object v)
restituisce *true* se la struttura dati contiene l'oggetto *v* come campo informativo, *false* altrimenti;

public Collection values ()
restituisce una vista che comprende tutti i valori contenuti nella tabella (la vista non supporta l'aggiunta di nuovi elementi: l'invocazione dei metodi *add()* e *addAll()* sulla vista solleva un'eccezione di tipo *UnsupportedOperationException*);

public void putAll (Map m)
inserisce nella struttura dati tutti gli elementi contenuti in *m*;

public Set entrySet ()
restituisce una vista che comprende tutti gli elementi (coppie chiave-valore) della struttura dati (la vista non supporta l'aggiunta di nuovi elementi: l'invocazione dei metodi *add()* e *addAll()* sulla vista solleva un'eccezione di tipo *UnsupportedOperationException*);

public Set keySet ()

restituisce una vista delle chiavi contenute nella struttura dati sotto forma di insieme (la vista non supporta l'aggiunta di nuovi elementi: l'invocazione dei metodi *add()* e *addAll()* sulla vista solleva un'eccezione di tipo *UnsupportedOperationException*);

public Object get (Object k)
restituisce l'oggetto associato alla chiave *k*, *null* se la chiave non è presente; per distinguere il caso in cui la chiave non è presente dal caso in cui il valore associato alla chiave *k* vale *null* è necessario usare il metodo *containsKey()*;

public Object remove (Object k)
rimuove dalla struttura dati la coppia chiave-valore specificata da *k*; il valore restituito è pari al campo informativo associato alla chiave *k*, se tale chiave è presente nella tabella, *null* altrimenti (il valore *null* può essere restituito anche nel caso in cui tale valore sia associato alla chiave *k*);

public Object put (Object k , Object v)
inserisce il valore *v* associandolo alla chiave *k*; se la chiave era già presente, il vecchio valore viene sostituito dal nuovo.

Le classi che implementano l'interfaccia *Map* devono ridefinire il metodo *equals()* in modo tale che restituisca *true* se le due istanze di *Map* contengono lo stesso numero di elementi e se ogni elemento di una istanza è uguale ad un elemento dell'altra istanza. Due elementi sono considerati uguali se sono vere entrambe le seguenti condizioni: i) il metodo *equals()* applicato alle due chiavi restituisce *true* (o le due chiavi hanno entrambe il valore *null*); ii) il metodo *equals()* applicato ai due campi informativi restituisce *true* (o i due campi hanno entrambi il valore *null*).

L'interfaccia *SortedMap* comprende i metodi che consentono di operare su una struttura dati analoga a quella descritta per l'interfaccia *Map* in cui le chiavi vengono mantenute ordinate. Tutte le chiavi devono implementare l'interfaccia *Comparable*. L'interfaccia *SortedMap* estende *Map*, ed aggiunge a quest'ultima i seguenti metodi principali:

public Object firstKey ()
restituisce la prima chiave;

public Object lastKey ()

restituisce l'ultima chiave;

public SortedMap headMap (Object** k **)
restituisce come *SortedMap* una vista costituita da tutte le coppie chiave-valore in cui la chiave è strettamente minore di *k*;

public SortedMap tailMap (Object** k **)
restituisce come *SortedMap* una vista costituita da tutte le coppie chiave-valore in cui la chiave è maggiore o uguale a *k*;

public SortedMap subMap (Objecti** i **, Object** j **)
restituisce come *SortedMap* una vista costituita da tutte le coppie chiave-valore che hanno una chiave compresa tra *i*, incluso, e *j*, escluso;

L'inserimento di un nuovo dato in una vista può generare un'eccezione di tipo *IllegalArgumentException* se la chiave specificata non appartiene all'intervallo delle chiavi comprese dalla vista.

Quando i metodi *keySet()*, *values()*, e *entrySet()*, ereditati da *Map*, sono invocati su una implementazione di *SortedMap*, le collezioni e gli insiemi ottenuti come risultato riflettono l'ordinamento delle chiavi.

5.11. Collection framework: realizzazioni e loro complessità

Le classi che realizzano le interfacce del collection framework hanno nomi composti, in cui la prima parte descrive il tipo di implementazione, mentre la seconda l'interfaccia implementata. Non esiste nessuna implementazione dell'interfaccia *Collection*.

Tutte le realizzazioni del collection framework non sono sincronizzate. Nel caso in cui siano utilizzate in un ambiente multi-thread, è necessario adoperare opportune versioni sincronizzate, che possono essere ottenute come illustrato nel paragrafo 5.12.

Prima di descrivere le implementazioni, diamo alcune nozioni elementari sul concetto di complessità di un algoritmo.

5.11.1. Complessità di un algoritmo

Un algoritmo è una descrizione del procedimento di risoluzione di un problema, attraverso una sequenza di passi elementari. La complessità di un algoritmo è una misura delle risorse di calcolo impiegate per risolvere il problema in funzione del numero n dei dati di ingresso (per esempio, nel caso di un algoritmo di ordinamento di un vettore, n è il numero degli elementi del vettore stesso). Si parla di complessità in tempo (o computazionale) quando si intende quantificare il tempo di esecuzione di un algoritmo, mentre si parla di complessità in spazio quando si intende quantificare l'occupazione di memoria di un algoritmo. In questo capitolo, per semplicità, faremo riferimento solo alla complessità computazionale.

Il tempo effettivamente impiegato da un algoritmo, oltre che dalla quantità dei dati in ingresso, dipende dai tempi di esecuzione delle singole istruzioni dell'agente di calcolo su cui l'algoritmo viene eseguito. Pertanto, per poter fornire una indicazione della complessità di un algoritmo che prescinda dallo specifico agente di calcolo, ci si riferisce al numero dei passi elementari più importanti che vengono effettuati (il tempo effettivo sarà poi proporzionale a tale numero). Comunemente, si prende in considerazione l'ordine della complessità di un algoritmo, tralasciando i termini di ordine inferiore e le costanti moltiplicative. Quindi diremo che la complessità di un algoritmo è proporzionale ad n per indicare che il tempo di esecuzione cresce linearmente con il numero dei dati in ingresso (per esempio, se il numero di passi elementari è pari a $2n$ o a $7n$) o che la complessità è proporzionale ad n^2 per indicare che il tempo di esecuzione cresce come il quadrato del numero dei dati in ingresso (per esempio, se il numero di passi elementari è pari a $3n^2$ o a $4n^2 + 5n$) e così via. Spesso è opportuno distinguere tra la complessità nel caso medio e quella nel caso peggiore: la prima si ottiene, per un dato n, eseguendo la media su tutti i possibili valori dei dati di ingresso; la seconda si ottiene considerando, per un dato n, quel particolare valore dei dati di ingresso che produce il peggior comportamento dell'algoritmo. Quando non diversamente specificato faremo riferimento alla complessità media di un algoritmo.

5.11.2 Classi *ArrayList* e *LinkedList*

La classe *ArrayList* implementa l'interfaccia *List* usando un array, che

viene ridimensionato quando la sua capacità non è più sufficiente a contenere tutti gli elementi. Per tale motivo le operazioni di accesso in lettura (*get()*) e sostituzione del valore di un elemento (*set()*) sono particolarmente efficienti, mentre le operazioni di inserimento e rimozione di un elemento sono più costose in quanto possono comportare il ridimensionamento dell'array e lo spostamento di dati precedentemente inseriti.

La classe è dotata di tre costruttori: ***ArrayList()*** che crea una istanza con capacità iniziale pari a dieci; ***ArrayList(Collection c)*** che crea una istanza contenente gli elementi di *c*, con capacità iniziale pari al 110% del numero di elementi di *c*; ***ArrayList(int cap)*** che crea una istanza con la capacità iniziale specificata.

La classe *LinkedList* implementa l'interfaccia *List* attraverso una lista doppia (ogni elemento contiene un riferimento all'elemento successivo e a quello precedente). Trattandosi di una implementazione a lista, tutte le operazioni che specificano l'indice dell'elemento su cui lavorare comportano lo scorrimento della lista fino al punto opportuno. Tuttavia, a differenza dalla classe *ArrayList*, le operazioni di inserimento e di estrazione di un dato non comportano mai la copiatura dei dati precedentemente inseriti.

La classe dispone di due costruttori: ***LinkedList()*** che crea una lista vuota, e ***LinkedList(Collection c)*** che crea una lista contenente gli elementi della collezione *c*.

Sia la classe *ArrayList* che la classe *LinkedList* ridefiniscono il metodo *toString()* restituendo una rappresentazione in formato stringa del loro contenuto.

A titolo di esempio, riportiamo un programma che legge da tastiera valori interi fino a quando l'utente non digita la stringa ".". I valori vengono inseriti in fondo ad una istanza di *LinkedList l1* sotto forma di oggetti *Integer* (non è possibile inserire oggetti appartenenti ai tipi primitivi). Successivamente, gli elementi di *l1* che hanno indice pari vengono inseriti anche in *l2*, una istanza di *ArrayList*. Il programma termina con la stampa del contenuto dei due oggetti *List*.

```
import java.util.*;
public class ProvaList
{ public static void main(String[] args)
  { List l1 = new LinkedList();
```

```
    List l2 = new ArrayList();
    String s;
    while(!(s = Console.leggiStringa()).equals("."))
    { Integer j = new Integer(s);
      l1.add(j);
    }
    int d = l1.size();
    Console.scriviStringa("Dim. di l1: " + d);
        for(int x=0; x<d; x+=2)
    { Integer ii = (Integer) l1.get(x);
      l2.add(ii);
    }
    Console.scriviStringa("l1: " + l1);
    Console.scriviStringa("l2: " + l2);
  }
}
```

La seguente è una possibile esecuzione del programma:

```
7
12
11
4
56
3
9
.
Dim. di l1: 7
l1: [7, 12, 11, 4, 56, 3, 9]
l2: [7, 11, 56, 9]
```

Come ulteriore esempio, vediamo come realizzare un tipo pila di *Object* sfruttando la classe *LinkedList*, classe più opportuna di *ArrayList* in quanto gli inserimenti e le estrazioni avvengono solo in testa (per brevità tralasciamo di implementare il metodo *equals()*):

```
import java.util.*;
public class PilaLinkedList implements Pila
// interfaccia di paragrafo 5.3
{ private List l;
  public PilaLinkedList()
  { l = new LinkedList(); }
```

```
  public void push(Object elem)
  { l.add(0,elem);  }
  public Object pop()
  { if(l.size() == 0) throw new PilaEmptyException();
    return l.remove(0);
  }
  public boolean isEmpty()
  { return l.size() == 0;  }
  public boolean isFull()
  { return false; }
  public int size()
  { return l.size();  }
  public String toString()
  { return l.toString(); }
}
```

5.11.3. Classi *HashSet* e *TreeSet*

La classe *HashSet* implementa l'interfaccia *Set* sfruttando una tabella hash per la memorizzazione degli elementi dell'insieme. Ogni istanza di *HashSet* è caratterizzata (all'atto della creazione) da un limite di carico (numero compreso fra 0.0 e 1.0): quando il *fattore di carico* (rapporto tra il numero di elementi presenti nella tabella e la capacità della tabella) è superiore al limite di carico, la capacità della tabella viene incrementata automaticamente.

La classe *HashSet* dispone dei seguenti costruttori: ***HashSet()*** che crea una istanza con capacità iniziale pari a *16* e limite di carico pari a *0.75*; ***HashSet(Collection** c**)*** che crea una istanza che contiene tutti gli elementi di *c* e limite di carico *0.75* (la capacità iniziale è determinata automaticamente); ***HashSet(int** cap**)*** che crea una istanza avente la capacità iniziale specificata e limite di carico *0.75*; ***HashSet(int** cap**, float** fac**)*** che crea una istanza con la capacità iniziale e il limite di carico specificati.

Assumendo che la funzione hash disperda opportunamente gli elementi, il tempo di accesso per le operazioni di inserimento, estrazione e ricerca di un elemento è costante. Il tempo necessario a scorrere gli elementi attraverso l'interfaccia *Iterator* è proporzionale alla somma del numero di elementi più la capacità della tabella hash.

```
import java.util.*;
```

```
public class ProvaHashSet
{ public static void main(String[] args)
  { Set s = new HashSet();
    while(true)
    { Console.scriviStringa("Inserisci una stringa" +
                            " (. per terminare)");
      String x = Console.leggiStringa();
      if(x.equals(".")) break;
      s.add(x);
    }
    Console.scriviStringa("Sono presenti " +
                          s.size() + " elementi");
    while(true)
    { Console.scriviStringa("Stringa da cercare" +
                            " (. per terminare) ?");
      String x = Console.leggiStringa();
      if(x.equals(".")) break;
      if(s.contains(x))
        Console.scriviStringa("Presente");
      else Console.scriviStringa("Assente");
    }
    Console.scriviStringa("Contenuto: " + s);
  }
}
```

La classe *TreeSet* fornisce una implementazione dell'interfaccia *SortedSet* basata su una struttura dati ad albero. Gli elementi dell'insieme sono mantenuti ordinati in accordo al metodo *compareTo()* dell'interfaccia *Comparable*, che deve essere implementata dalla classe a cui appartengono gli elementi stessi. Le operazioni di inserimento, di rimozione e di ricerca di un elemento hanno complessità dell'ordine di *log(n)*, dove *n* è il numero di elementi appartenenti all'insieme.

I principali costruttori sono: ***TreeSet()*** che crea un insieme vuoto, e ***TreeSet(Collection c)*** che crea un insieme ordinato contenente tutti gli elementi della collezione *c*.

A titolo di esempio, riportiamo un programma che inserisce in una lista *l* dieci valori interi casuali compresi tra zero e cinque. Quindi, crea un insieme *ts* che contiene ordinati tutti i valori diversi presenti nella lista:

```
import java.util.*;
public class ProvaTreeSet
```

```
{ public static void main(String[] args)
  { List l = new ArrayList();
    for(int i=0; i<10; i++)
    { Integer j =
        new Integer((int)Math.round(Math.random()*5));
      l.add(j);
    }
    Set ts = new TreeSet(l);
    Console.scriviStringa("Contenuto di l:\n" + l);
    Console.scriviStringa("Contenuto di ts:\n" + ts);
  }
}
```

Eseguendo il programma si può ottenere un'uscita simile alla seguente:

```
Contenuto di l:
[2, 1, 2, 4, 2, 4, 2, 2, 2, 5]
Contenuto di ts:
[1, 2, 4, 5]
```

5.11.4. Classi *HashMap* e *TreeMap*

La classe *HashMap* implementa l'interfaccia *Map* con una struttura dati di tipo tabella hash. Ogni istanza di *HashMap* è caratterizzata (all'atto della creazione) da un limite di carico (numero compreso fra 0.0 e 1.0): quando il *fattore di carico* (rapporto tra il numero di elementi presenti nella tabella e la capacità della tabella) è superiore al limite di carico, la capacità della tabella viene incrementata automaticamente.

La classe dispone dei seguenti costruttori: ***HashMap()*** che crea una istanza con capacità iniziale pari a *16* e limite di carico pari a *0.75*; ***HashMap(Map** m**)*** che crea una istanza che contiene tutti gli elementi di *m* e limite di carico *0.75* (la capacità iniziale è determinata automaticamente); ***HashMap(int** cap**)*** che crea una istanza avente la capacità iniziale specificata e limitee di carico *0.75*; ***HashMap(int** cap**, float** fac**)*** che crea una istanza con la capacità iniziale e il limite di carico specificati.

Assumendo che la funzione hash disperda opportunamente gli elementi, il tempo di accesso per le operazioni di inserimento, di estrazione e di ricerca è costante. Il tempo necessario a scorrere gli elementi attraverso

l'interfaccia *Iterator* è proporzionale alla somma del numero di elementi più la capacità della tabella hash.

La classe *TreeMap* implementa l'interfaccia *SortedMap* memorizzando gli oggetti all'interno di una struttura ad albero. I dati sono mantenuti ordinati per chiave all'interno dell'albero. La relazione di ordinamento tra le chiavi è determinata attraverso il metodo *compareTo()* dell'interfaccia *Comparable*. Le operazioni di inserimento, di estrazione e di ricerca di un dato hanno una complessità dell'ordine di *log(n)*, dove *n* è il numero di elementi presenti.

Tra i costruttori disponibili citiamo ***TreeMap()*** che crea una istanza di *TreeMap* vuota, e ***TreeMap(Map** m**)*** che crea una istanza di *TreeMap* che contiene gli elementi di *m* ordinati.

A titolo di esempio riportiamo un programma che ciclicamente legge da tastiera un intero e una stringa e li inserisce in una istanza di *TreeMap*. L'intero, incapsulato in una istanza di *Integer*, funge da chiave, mentre la stringa costituisce il valore. Quindi, vengono ottenuti e stampati dalla istanza di *TreeMap* l'insieme delle chiavi e la collezione dei valori.

```
import java.util.*;
public class ProvaTreeMap
{ public static void main(String[] args)
  { Map m = new TreeMap();
    while (true)
    { Console.scriviStringa
          ("Inserisci un intero (0 per terminare)");
      int i = Console.leggiIntero();
      if(i == 0) break;
      Console.scriviStringa("Inserisci una stringa");
      String x = Console.leggiStringa();
      m.put(new Integer(i), x);
    }
    int j = 0;
    Console.scriviStringa("Chiavi:");
    Iterator it = m.keySet().iterator();
    while(it.hasNext())
    { Integer k = (Integer) it.next();
      Console.scriviStringa("Chiave " + j++ +
                            " :" + k);
    }
    j = 0;
    Console.scriviStringa("Valori: ");
```

```
    it = m.values().iterator();
    while(it.hasNext())
    { String s = (String) it.next();
      Console.scriviStringa("Valore " + j++ +
                            " :" + s);
    }
    Console.scriviStringa("Contenuto: " + m);
  }
}
```

La seguente è una possibile esecuzione del programma:

```
Inserisci un intero (0 per terminare)
3
Inserisci una stringa
aaa
Inserisci un intero (0 per terminare)
4
Inserisci una stringa
bbb
Inserisci un intero (0 per terminare)
1
Inserisci una stringa
ccc
Inserisci un intero (0 per terminare)
0
Chiavi:
Chiave 0 :1
Chiave 1 :3
Chiave 2 :4
Valori:
Valore 0 :ccc
Valore 1 :aaa
Valore 2 :bbb
Contenuto: {1=ccc, 3=aaa, 4=bbb}
```

Come è possibile notare, quando si scorre l'insieme delle chiavi attraverso un iteratore, la sequenza dei valori è ordinata in senso crescente, indipendentemente dall'ordine di inserimento delle chiavi. Quando si scorre l'insieme dei valori, la sequenza è determinata dall'ordinamento tra le chiavi. Si noti infine che anche la rappresentazione in formato stringa è ordinata per valori crescenti delle chiavi.

5.12. Collection framework: algoritmi di utilità

Il collection framework comprende la classe *Collecions* (package *java.util*), composta esclusivamente da metodi statici che operano su collezioni e che realizzano algoritmi di utilità generale.

Riportiamo alcuni dei metodi più importanti:

public Object max (Collection c)
restituisce il più grande degli oggetti contenuti nella collezione *c*;

public Object min (Collection c)
restituisce il più piccolo degli oggetti della collezione *c*;

public void sort (List l)
ordina gli elementi di *l* in senso crescente;

public void reverse (List l)
inverte la posizione degli elementi di *l*;

public void rotate (List l , int d)
ruota di *d* posizioni gli elementi di *l*;

public Collection synchronizedCollection (Collection c)
public Set synchronizedSet (Set s)
public SortedSet synchronizedSortedSet (SortedSet s)
public List synchronizedList (List l)
public Map synchronizedMap (Map m)
public SortedMap synchronizedSortedMap (SortedMap m)
restituiscono una vista dell'oggetto argomento i cui metodi sono sincronizzati sulla vista (vista sincronizzata).

Per assicurarsi che tutti i thread accedano alla collezione con metodi sincronizzati, si può creare la collezione stessa come se fosse una vista, per esempio con la seguente riga di codice:

```
List l = Collections.synchronizedList(new ArrayList());
```

In questo modo l'unico riferimento disponibile è quello alla vista.

Gli iteratori e le viste ottenuti a partire da una vista sincronizzata non sono sincronizzati, ma tale compito è demandato al programmatore, che deve effettuare una sincronizzazione esplicita, per esempio come indicato dal seguente frammento di codice:

```
Set s = Collections.synchronizedSet(new HashSet());
// ...
synchronized(s)
{  Iterator it = s.iterator();
   while(it.hasNext())
   {  unMetodo(it.next());  }
}
```

La sincronizzazione deve avvenire sempre sulla collezione da cui l'iteratore o la vista sono stati generati.

A. Appendice I

A.1. Protocollo HTTP

Il sistema *World Wide Web* (WWW) è essenzialmente composto da tre parti: i programmi clienti, detti *Web browser*, che sono usati dagli utenti per navigare all'interno dell'insieme di documenti ipertestuali, i programmi servitori, detti *Web server*, che contengono informazioni e le dispensano ai programmi clienti, e Internet, che costituisce l'infrastruttura di comunicazione che permette ai programmi clienti e ai programmi servitori di dialogare fra loro.

Il paradigma di comunicazione tra clienti e servitori è del tipo richiesta/risposta: il browser invia una richiesta al servitore che risponde inviando al cliente i dati relativi alla risorsa richiesta (il servitore comunica con il cliente solo quando stimolato). Il protocollo usato nelle comunicazione tra cliente e servitore è l'HTTP (*Hypertext Transfer Protocol*), un protocollo di livello applicativo costruito sopra i livelli TCP/IP. La porta predefinita su cui un servitore HTTP si pone in attesa è la numero 80.

Il protocollo HTTP è quindi usato per trasferire risorse, dove con il termine risorsa si intende un insieme di dati identificabile attraverso un URL. Nel caso più comune, una risorsa è un file che risiede sul servitore, per esempio una pagina HTML, un file contenente un'immagine o un file audio, ma niente vieta che i dati della risorsa siano generati dinamicamente dal servitore, come per esempio accade usando i servlet.

A.1.1. Richiesta

Una richiesta HTTP è espressa come un messaggio testuale composto da una riga iniziale, zero o più righe di intestazione (dette *header*), una riga vuota e un eventuale corpo del messaggio. Tutte le righe, sia del messaggio di richiesta che di risposta, terminano con i caratteri CR e LF (CRLF). Il formato del messaggio di richiesta è quindi il seguente:

```
request
    request-line  header-sequence|opt  CRLF  message-body|opt
```

La riga iniziale (*request-line*) contiene l'indicazione del tipo della richiesta (detto *metodo*), l'identificatore della risorsa (*request-URI*), e la versione del protocollo (nella forma "*HTTP/x.y*") secondo il seguente formato:

```
request-line
    method  request-URI  HTTP-version  CRLF
```

I possibili metodi sono:

```
method
    one of
     GET  HEAD  POST
```

Il significato dei metodi è il seguente:

- *GET* è il più comune dei metodi, indica che il cliente vuole la risorsa specificata;
- *HEAD* è simile al metodo *GET*, ma il servitore deve rispondere al cliente inviando come risposta solo gli header (senza includere la risorsa vera e propria); è utile per avere alcune informazioni sulla risorsa senza doverla scaricare;
- *POST* usato per inviare dei dati, come corpo della richiesta, al programma servitore; in questo caso l'URI della risorsa identifica il programma che deve processare i dati.

Riportiamo un esempio di riga iniziale dove il metodo è *GET* e la versione del protocollo è *HTTP/1.0*:

```
GET  /pub/WWW/TheProject.html  HTTP/1.0
```

Gli header forniscono informazioni aggiuntive sulla richiesta o sull'oggetto allegato come corpo del messaggio (*message-body*). Sono costituiti da coppie nome/valore separate dal carattere ':'. Per gli header e le sequenze di header vale quanto segue:

header-sequence
    header
    header header-sequence

header
    header-name **:** *header-value* ***CRLF***

Il protocollo definisce un certo insieme di nomi di header tra cui:

- *User-agent* identifica il programma che ha eseguito la richiesta; un possibile valore è, per esempio, "*Mozilla/5.0*";
- *Content-type* definisce il tipo *MIME* dell'oggetto allegato alla richiesta; possibili valori sono, per esempio, "*text/plain*", "*text/html*", e "*image/gif*";
- *Content-length* indica il numero di byte che costituiscono il corpo (*message-body*) della richiesta;
- *Host* un Web server può ospitare le pagine di siti relativi a domini diversi, per esempio *www.uno.com* e *www.due.com*; quando l'URI identifica la risorsa in modo relativo, ovvero non include l'indicazione del sito a cui appartiene la risorsa, il valore specificato dall'header *host* viene usato per capire a quale dei domini sia riferito.

A.1.2. Risposta

Il messaggio di risposta è simile a quello di richiesta ed è composto da una riga di stato (*status-line*), zero o più header, una riga vuota e un eventuale corpo della risposta:

response
status-line header-sequence|**opt** ***CRLF*** *message-body*|**opt**

La riga di stato contiene la versione del protocollo (*HTTP-version*), il codice della risposta (*status-code*) e una descrizione testuale del codice della risposta (*reason-phrase*):

status-line
HTTP-version status-code reason-phrase ***CRLF***

Il codice di stato è composto da tre cifre, la prima delle quali identifica il tipo di risposta:

- *1xx – Informativo:* la richiesta è stata ricevuta, si prosegue;
- *2xx – Successo:* l'operazione è stata completata con successo;
- *3xx – Redirezione*: il cliente deve essere rediretto verso un altro URL;
- *4xx – Errore del cliente*: la richiesta non è ben formata o non può essere soddisfatta;
- *5xx – Errore del servitore*: la richiesta era valida ma il servitore non è in grado di soddisfarla.

Riportiamo i due codici di stato (e relativa descrizione testuale) più comuni:

- **`200 OK`**
  l'operazione è stata terminata con successo e la risorsa richiesta è contenuta nel corpo della risposta;
- **`404 Not found`**
  la risorsa richiesta non è stata trovata.

Gli header del messaggio di risposta hanno la forma vista in precedenza.

A.1.3. Esempio

A titolo di esempio riportiamo il messaggio di richiesta generato da un browser visitando la pagina *http://www.ing.unipi.it* e la risposta del server:

```
GET / HTTP/1.0
Host: www.ing.unipi.it
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1;
en-US; rv:1.7) Gecko/20040614 Firefox/0.9


HTTP/1.1 200 OK
Date: Mon, 19 Jul 2004 12:43:16 GMT
Server: Apache/1.3.27 (Unix) mod_perl/1.26
Connection: close
Content-Type: text/html

<html>
...
</html>
```

A.2. HTTP/1.1

La versione 1.1 del protocollo HTTP migliora la versione 1.0 con alcune estensioni, tra cui la persistenza delle connessioni ed il trasferimento a blocchi.

La versione 1.0 del protocollo HTTP usa una connessione TCP separata per ogni transazione: la connessione viene aperta per inviare il messaggio di richiesta, quindi una volta giunta la risposta la connessione viene immediatamente chiusa. Questo causa un carico maggiore per i servitori e tempi di risposta più lunghi (si pensi per esempio al caso di una pagina HTML che contiene un numero elevato di immagini ognuna delle quali richiede una connessione TCP separata).

La versione 1.1 del protocollo consente di usare la stessa connessione TCP per eseguire più transazioni: il cliente invia più richieste al server, una di seguito all'altra, sulla stessa connessione e il server risponde con una sequenza di risposte che ha lo stesso ordine. Se una richiesta include l'header "*Connection: close*", la connessione deve essere chiusa dopo che il servitore ha inviato la risposta corrispondente. In maniera analoga, il servitore può informare il cliente della propria volontà di chiudere la connessione includendo lo stesso header nel messaggio di risposta.

Un servitore può cominciare ad inviare i dati della risposta anche quando non ne conosce ancora la dimensione totale usando il meccanismo del trasferimento a blocchi. Questa opzione risulta particolarmente utile quando la risorsa è generata dinamicamente dal processo servitore. Una risposta in cui i dati sono trasferiti a blocchi contiene l'header "*Transfer-Encoding: chunked*" (l'header "*Content-length*" è assente). Il corpo della risposta è composto da blocchi, dove ogni blocco è costituito da una riga contenente la dimensione del blocco seguita dai dati del blocco. L'ultimo blocco è seguito da una riga contenente il carattere '0' e da una riga vuota.

Infine, le specifiche della versione 1.1 includono quattro nuovi metodi (usati non molto spesso e non sempre implementati):

- *OPTIONS* restituisce al cliente l'elenco delle opzioni disponibili per comunicare con il servitore o con una sua risorsa;
- *PUT* chiede al servitore di memorizzare l'oggetto incluso come corpo della richiesta con il nome specificato attraverso l'URI;
- *DELETE* chiede al servitore di cancellare la risorsa specificata;
- *TRACE* il servitore risponde inviando un messaggio di risposta che contiene una copia del messaggio di richiesta del cliente (metodo usato a scopo diagnostico).