Appunti per il Corso di LIP
Primo Semestre 2006-2007

Lorenzo Cioni

16 novembre 2006

Dipartimento di Informatica, Università di Pisa

e-mail: lcioni@di.unipi.it

< − >

Attenzione: versione preliminare

Sono graditi commenti e segnalazioni di errori

Indice

Elenco delle figure

1 Introduzione

Nelle presenti note si affrontano brevemente i seguenti argomenti:

1. la struttura del Sistema Operativo Linux;
2. alcuni comandi testuali in Linux;
3. la gestione attraverso l'interfaccia grafica;
4. l'utilizzo di Java;
5. l'utilizzo di Eclipse.

Nelle presenti note non si fa cenno alle problematiche relative alla istallazione, configurazione e amministrazione di un sistema di calcolo gestito dal Sistema Operativo Linux (ovvero si esamina tale sistema lato utente) e lo stesso vale per quanto riguarda Java e suo il supporto run time e il programma Eclipse. L'idea di base è quella di avere una macchina astratta Linux/Java/Eclipse cui l'utente, in possesso di una conoscenza almeno di base del linguaggio Java, accede per gestire i suoi progetti Java, dalla stesura al debugging all'esecuzione.

1.1 Linux, la struttura

Il Sistema Operativo (*SO*) è un insieme di componenti software che si occupano di gestire sia l'hardware sia i programmi applicativi, i documenti ed i dati degli utenti. Linux è un SO:

1. multiutente,
2. multitasking,

ovvero ha come primitivi i concetti di proprietà e di procedure di accesso (multiutenza) e consente ad ogni utente di eseguire più task (vedi oltre) in concorrenza fra di loro (multitasking).
Le componenti principali del SO Linux sono le seguenti [Pet96]:

1. il kernel,
2. la shell,
3. il file system,
4. le utility.

Il **kernel** rappresenta l'elemento che implementa le caratteristiche del SO e gestisce i dispositivi hardware. La **shell** è l'interfaccia fra l'utente e il kernel in quanto riceve i comandi dall'utente e li invia, se corretti, al kernel perchè siano eseguiti.
Il **file system** rappresenta l'organizzazione logica dei dati sui dispositivi fisici di memorizzazione. Ha una tipica struttura gerarchica composta da contenitori (directory in contesto testuale o folder in contesto grafico) e contenuti (tipicamente stream di bit detti file).
Le utility sono tutti i programmi, sia di sistema sia di utente, che il kernel esegue in modo concorrente.
L'accesso degli utenti ai servizi del SO può avvenire in due modi:

1. a riga di comando,
2. tramite un'interfaccia grafica o *GUI* (graphical user interface).

Nel primo modo l'utente interagisce con il SO tramite una shell che accetta i comandi sotto forma di stringhe di caratteri con una ben precisa sintassi. Tipicamente, nel caso di Linux, la sintassi è la seguente:

$$comando\ opzioni\ parametri \tag{1}$$

In tale stringa si ha:

1. *comando* individua un programma eseguibile;
2. le *opzioni* sono precedute dal carattere − e rappresentano i **modificatori** del comando;
3. i *parametri* (opzionali, esistono comandi che non prevedono parametri) sono gli elementi su cui il comando agisce.

Esistono comandi che non prevedono opzioni. Esempi sono **cd** e **pwd** (vedi oltre).
Nel secondo modo l'interazione avviene attraverso un'interfaccia grafica che costituisce un gestore di finestre e implementa, in genere, la metafora della scrivania (o desktop). In molte versioni di Linux il desktop è caratterizzato da più aree di lavoro fra le quali l'utente può switchare. In tale ambiente sono compresi:

1. il **file manager** per la gestione di folder e file,
2. il **program manager** per la gestione dei programmi in esecuzione (ovvero dei processi).

All'interno della GUI, l'utente può accedere ai cosiddetti **terminali** attraverso i quali può immettere i comandi sotto forma di stringhe, come avviene nel caso dell'interfaccia a riga di comando o testuale. La GUI interagisce, in ogni caso, con la shell la quale, a sua volta, interagisce con il kernel.

1.1.1 La shell

Nel caso di una shell a linea di comando si ha che questa svolge essenzialmente i compiti seguenti:

1. gestisce la linea di immissione dei comandi (editing della linea di comando);
2. gestisce la cronologia dei comandi (history);
3. gestisce i cosiddetti metacaratteri;
4. interpreta i comandi immessi dall'utente ovvero passa al kernel l'indicazione di quali programmi eseguire, con quali opzioni e su quali dati;
5. gestisce le cosiddette redirezioni e la concatenazione dei comandi;
6. consente di gestire i processi (ovvero i programmi in esecuzione) commutandoli fra le varie modalità di esecuzione possibili (foreground e background).

1.1.2 Il file system

Il **file system** consente la gestione dei file in una struttura gerarchica ad albero la cui radice viene indicata con il carattere / e rappresenta il "contenitore globale" di tutti i documenti contenuti nel sistema, comprese altre porzioni di file system accessibili tramite connessioni remote. Le directory, o folder, sono definibili, in modo ricorsivo, come contenere file e altre directory. Alcune directory sono dette di sistema e contengono file e programmi tipici del SO. Altre directory sono tipiche degli utenti e contengono file e programmi "privati". Ogni directory contiene sempre due riferimenti:

1. uno alla directory stessa;
2. uno alla directory di livello superiore nella struttura gerarchica del file system e detta directory padre.

Ogni utente, al momento di accedere al SO, accede, di default, ad una directory particolare, detta **home directory**, che rappresenta il suo ambiente di lavoro in cui sono memorizzati tutti i suoi dati e i suoi programmi.
Il SO mette a disposizione degli utenti un nutrito insieme di comandi di tipo testuale (o da linea di comando) per la gestione delle directory e dei file in essi contenuti.
Comandi analoghi sono accessibile tramite la GUI sia facendo uso di comandi contenuti in menù sia mediante operazioni eseguite, con l'ausilio del mouse, sulle icone rappresentative degli oggetti o mediante l'accesso a menù flottanti, richiamabili sul desktop, di solito con l'ausilio del pulsante destro del mouse.

1.1.3 Le utility

Le utility sono, essenzialmente, programmi di servizio quali editori, programmi di visualizzazione, filtri e programmi di comunicazione.
I primi consentono di creare e modificare file di testo, grafici e di molti altri tipi, i secondi consentono di visualizzare i file senza consentire di apportarvi modifiche.
I filtri accettano dati da altri comandi, da tastiera o da file, e li modificano in base a certi criteri per produrre dati da immettere in file (redirezione), inviare ad altri comandi (concatenamento) o visualizzare sullo schermo. Vedremo a breve alcune nozioni relative agli operatori di redirezione e di concatenazione, accessibili solo tramite l'interfaccia testuale e non tramite la GUI.
I programmi di comunicazione consentono l'interazione con altri computer e con dispositivi hardware collegati in locale o in remoto. La trattazione delle utility più complesso esula dall'ambito delle presenti note.

1.1.4 L'accesso al sistema

Il SO Linux è un SO multiutente per cui l'effettivo uso dei suoi servizi è subordinato alla esecuzione delle operazioni di **login** mediante le quali un utente fornisce gli elementi che lo identificano al sistema (user-id e password) e sui quali non ci soffermiamo oltre.
Una volta eseguito l'accesso, l'interazione può avvenire in due modi:

1. tramite una GUI (o interfaccia grafica),
2. tramite una interfaccia testuale (o a riga di comando).

In entrambi i casi l'accesso può avvenire sul sistema locale o su un sistema remoto, connesso al sistema locale tramite una connessione dedicata o

condivisa. Nel seguito supporremo un accesso in locale. L'interfaccia testuale, inoltre, sarà vista come una delle possibili applicazioni che l'utente può gestire, in modo concorrente, tramite la GUI.

1.1.5 La GUI

La GUI implementa la metafora del desktop (o scrivania) e consente l'accesso tramite:

1. finestre,
2. menù,
3. icone.

Le **finestre** sono aree dello schermo dedicate alla visualizzazione di dati e sono, di solito, associate ad una applicazione e corredate da menù di comandi tipici dell'applicazione e da elementi per la gestione della finestra (spostamento, ridimensionamento, chiusura e così via.
I **menù** rappresentano un metodo per raggruppare comandi e consentono la gestione del Desktop e la gestione sia delle finestre sia delle icone e degli elementi ad esse associati.
Le **icone** sono immagini associate a programmi e/o file e/o dispositivi hardware. Tramite il mouse, le icone possono essere spostate, rimosse e selezionate in modo da aprire l'elemento corrispondente o mandarlo in esecuzione, se si tratta di un programma: una icona di solito presente sul desktop è quella associata alla home directory la cui apertura consente l'accesso, tramite la GUI, ai file e alle directory contenute nella home directory dell'utente.
La GUI consente l'apertura contemporanea di più applicazioni, ognuna con una o più finestre in modo da implementare il multitasking del SO. In ogni caso solo una delle applicazioni in esecuzione ha il focus, ovvero accetta dati da tastiera e reagisce alle azioni del mouse, mentre tutte le applicazioni in esecuzione accedono in modo concorrente al video.
Vedremo come, nel caso dell'interfaccia testuale, si possano avere applicazioni o programmi in esecuzione (ovvero processi) che sono eseguiti o in **foreground** o in **background**, cosa ciò voglia dire e come sia possibile far passare un processo dall'una all'altra modalità.

1.1.6 La documentazione in linea

Si ritiene di consigliare l'uso di comandi quali:

1. man,

2. whatis,

3. apropos,

immessi tramite un'interfaccia testuale (rappresentata nella GUI dal programma **xterm** o **XTerm** e simili) per ottenere informazioni relative ad un comando e alla sua sintassi.
La sintassi di tali comandi è la seguente:

$$cmd\ comando \tag{2}$$

in cui:

$$cmd \in \{man, whatis, apropos\} \tag{3}$$

e *comando* è il comando in merito al quale si cercano informazioni. Una interessante possibilità è la seguente:

$$man\ man \tag{4}$$

1.1.7 Esempi di comandi

Vediamo ora alcuni esempi di comandi eseguiti sia tramite l'interfaccia testuale (nel seguito TI) sia tramite la GUI. Nel secondo caso i comandi sono acceduti tramite menù a tendina o flottanti: i primi sono posizionati, di solito, sotto la barra del titolo delle varie applicazioni mentre i secondi sono richiamabili, in vari contesti, usando il pulsante destro del mouse. In genere si dà prima il comando nella TI poi il comando (o le azioni corrispondenti) nella GUI:

1. creazione di una directory (**mkdir**) o di un folder (**new folder**) da menù a tendina o flottante;

2. spostamento di file (**mv**), operazioni di **drag and drop** con il mouse;

3. copia di file (**cp**), operazione di **select** con il mouse, comando di **copy** da menù flottante e comando di **paste** da menù flottante;

4. cancellazione di file (**rm**), operazioni di **drag and drop** nel "cestino" con il mouse.

Torneremo dettagliatamente su tali comandi in quanto segue.

1.2 La Shell

La **shell** rappresenta il programma di interfaccia fra l'utente e il kernel sia che questi interagisca con il SO tramite una GUI sia che interagisca tramite una TI. Nel seguito ci riferiremo al primo caso, il secondo essendo un caso speciale di questo. All'interno di una GUI un utente può avere in esecuzione un numero qualunque di TI, ciascuna associata ad un programma/processo terminale.
La shell gestisce la riga di comando accessibile tramite gli **xterm** e tramite i vari menù disponibili sul desktop e sulle finestre.
La gestione della riga di comando si traduce in:

1. gestione dei metacaratteri o caratteri speciali;
2. gestione delle redirezioni e della concatenazione dei comandi;
3. gestione dei processi di utente;
4. gestione di **alias**, della **history** e delle **variabili di ambiente e di sistema**.

Gli **alias** sono sinonimi mnemonici di comandi complessi e/o di uso frequente. Si differenziano dagli script perchè sono associati di solito a singoli comandi che la shell si occupa di mandare in esecuzione (con le opportune opzioni) mentre gli script equivalgono a programmi complessi che la shell interpreta passo passo.
La **history** contiene la cronologia, di lunghezza configurabile, dei comandi immessi nel recente passato dall'utente e che l'utente può richiamare in esecuzione, eventualmente modificandoli.
Le variabili di ambiente e di sistema consentono all'utente di personalizzare il suo ambiente di lavoro e di associare nomi mnemonici a dispositivi fisici quali le stampanti.
Nel seguito vedremo come accedere la shell da una TI. L'accesso tramite una GUI non presenta grosse difficoltà (a parte la ricerca dei corrispettivi dei comandi che vedremo nei vari menù a tendina e flottanti che la GUI mette a disposizione degli utenti).

1.2.1 La riga di comando

La shell accetta comandi sulla linea di comando e tale sua disponibilità è segnalata dalla presenza di un prompt, configurabile dall'utente (il valore corrente è contenuto nella variabile di ambiente **PROMPT**). Nel seguito

supporremo che il prompt sia il carattere # per cui la situazione seguente all'interno di un **xterm**:

$$\# \tag{5}$$

segnala che la shell correntemente in esecuzione e che ha il focus è in grado di accettare un comando dall'utente. I comandi sono immessi effettivamente solo quando l'utente premo il tasto **Enter** o **Invio** sulla tastiera, cui corrisponde il carattere terminatore di comando. Fino a quel momento l'utente può editare il comando e modificarlo secondo le sue esigenze (o correggerlo se errato in qualche suo aspetto: lessicale, sintattico o semantico).
Esempi di comandi sono i seguenti:

$$\# \, date \tag{6}$$

che stampa il valore della data corrente;

$$\# \, ls \; -la \tag{7}$$

che visualizza il contenuto della directory corrente con, per ogni elemento, tutte le informazioni che lo descrivono.
La sintassi generale dei comandi è la seguente:

$$comando \, [opzioni] \, [argomenti] \tag{8}$$

Il **comando** rappresenta il nome di un programma eseguibile noto al SO ed è l'unico elemento obbligatorio della sintassi, dato che gli altri elementi sono opzionali.
Le **opzioni** sono modificatori dei comandi e sono singoli caratteri ciascuno preceduto dal carattere $-$ o gruppo di caratteri preceduti da un sono carattere $-$. Ad esempio i seguenti comandi sono equivalenti:

$$\# \, ls \; -la \tag{9}$$

$$\# \, ls \; -l \; -a \tag{10}$$

I **parametri** rappresentano i dati su cui il comando agisce. Vedremo che si hanno casi di comandi che non accettano parametri (ad esempio il comando **pwd** che visualizza la "posizione" nel file system della directory corrente) e che i parametri, se ammessi dal comando, possono essere ottenuti per redirezione da file o per concatenazione da altri comandi.
La riga di comando fa uso di un **buffer** che contiene tutti i caratteri fino al carattere terminatore in modo che l'utente possa editare il suo contenuto utilizzando i tasti **backspace** e **canc**, i tasti di spostamento (freccia a sinistra o a destra) e combinazioni di caratteri. Ad esempio (se con **CTRL** si individua il tasto **control**):

1. **CTRL-U** cancella l'intera riga di comando;
2. **CTRL-F** sposta il cursore a sinistra di un carattere sulla riga di comando;
3. **CTRL-B** sposta il cursore a destra di un carattere sulla riga di comando;
4. **CTRL-D** cancella il carattere che precede il cursore sulla riga di comando.

La riga di comando può contenere il carattere ; come separatore di comandi che sono eseguiti in successione da sinistra a destra e può contenere il carattere \ che fa da escape del carattere terminatore di linea in modo che un comando possa essere spezzato su più linee separate (o perchè troppo lungo o per aumentarne la leggibilità).

1.2.2 I metacaratteri

Altri elementi che possono essere presenti sulla linea di comando (e che sono "risolti" dalla shell prima che questa passi un comando al kernel perchè questi lo esegua) sono i **metacaratteri** o **caratteri speciali** quali:

1. $*$
2. ?
3. []
4. $\wedge$

Per la loro interpretazione è bene tenere presente che non hanno lo stesso significato di simboli analoghi che si trovano nelle grammatiche regolari per cui si ha:

1. $*$ indica una qualunque stringa di caratteri legali, anche vuota, in modo che il comando **ls d*** elenca file e directory contenuti nella directory corrente i cui nomi iniziano per *d* seguito da qualunque insieme di caratteri;
2. ? indica un carattere singolo, in modo che il comando **ls d?.doc** elenca file e directory contenuti nella directory corrente i cui nomi iniziano per *d* seguito da un qualunque carattere seguito dalla stringa **.doc**;

3. [] individua un insieme di caratteri "contigui" in modo che $[1-3]$ individua l'insieme composto dai caratteri $1, 2, 3$ e $[a-zA-Z]$ individua tutti i caratteri alfabetici minuscoli e maiuscoli;

4. $\wedge$ usato insieme al precedente ne individua il complemento rispetto all'insieme di tutti i caratteri.

I metacaratteri sono interpretati dalla shell e sostituiti con stringhe effettive prima che i comandi siano passati al kernel per essere eseguiti.

1.2.3 Input e output standard, redirezioni e concatenazione

Le operazioni di input e output dati sono associate a dei canali logici noti come:

1. standard input o **stdin**;

2. standard output o **stdout**;

3. standard error o **stderr**.

Di default lo **stdin** è associato alla tastiera mentre lo **stdout** e lo **stderr** sono associati al video. È possibile fare in modo che:

1. lo **stdin** sia associato ad un file (redirezione) o ad un comando (concatenazione);

2. lo **stdout** e lo **stderr** siano associati ad un file (redirezione) o ad un comando (concatenazione).

La redirezione dello **stdout** su un file può essere ottenuta facendo uso degli operatori:

1. >

2. >>

La sintassi di tali operatori presuppone l'uso insieme a comandi che producono dati sullo **stdout**. Non ha senso usarli su comandi che assegnano valori alle variabili di ambiente o di sistema oppure comandi quali:

1. **cd** *nome_dir* che modifica la posizione corrente all'interno del file system (vedi oltre);

2. **chmod**, **chown** che modificano diritti e proprietà a file e directory (vedi oltre).

La sintassi è la seguente:

$$\#\, comando > nome\_file \tag{11}$$

$$\#\, comando >> nome\_file \tag{12}$$

Nel primo caso il flusso di dati prodotto da *comando* viene immesso nel file $nome\_file$ il cui contenuto precedente viene perduto (se il file preesiste al comando lo si sovrascrive). Nel secondo caso il flusso di dati prodotto da *comando* viene accodato al contenuto pre-esistente nel file $nome\_file$ il cui contenuto precedente viene mantenuto. Se il file non esiste prima della esecuzione del comando si ha un comportamento identico a quello precedente. Se si vuole impedire che il contenuto di un file esistente venga sovrascritto per errore usando il primo operatore si può settare una variabile di ambiente con il comando:

$$\#\, set\ -o\ noclobber \tag{13}$$

in modo che il comando:

$$\#\, comando > nome\_file \tag{14}$$

non sia eseguito ma produca un messaggio di errore se il file $nome\_file$ esiste già al momento della esecuzione del comando. Se si vuole forzare l'esecuzione della redirezione si può procedere come segue:

$$\#\, comando >!\ nome\_file \tag{15}$$

La redirezione dello **stdout** può essere usata per creare file contenenti i dati prodotti dalle esecuzione di comandi e utilizzabili come input da altri comandi. Ad esempio il comando:

$$\#\, ls\ -la\ *.java > src\_java \tag{16}$$

crea il file *src_java* che contiene i nomi di tutti i file sorgente in Java nella directory corrente. Lo **stdin** di default è associato alla tastiera in modo che i caratteri immessi siano rediretti al programma che il quel momento ha il focus della tastiera (e anche sullo schermo se è attiva la funzione di echo che consente all'utente di verificare la correttezza o meno di quanto immesso). È possibile fare in modo che:

1. un comando che accetta dati da un file li possa prendere dallo **stdin** fino a che l'utente non immette il carattere di fine file (o EOF) che, in ambiente Unix/Linux, è rappresentato dalla combinazione CTRL-D;

2. un comando che accetta dati sulla linea di comando li può accettare da file mediante la redirezione.

Nel primo caso si può utilizzare in associazione la redirezione dello **stdout** in modo da creare un file con quanto immesso da tastiera fino all'EOF.
Ad esempio, il comando:

$$\#\, cat\ > lettera \tag{17}$$

accetta dati da tastiera fino all'EOF ed immette tutto nel file *lettera*, con le cautele viste in precedenza.
Nel secondo caso il seguente comando (un po' artificioso):

$$\#\, cat\ < lettera \tag{18}$$

visualizza il contenuto del file *lettera* ed equivale al seguente:

$$\#\, cat\, lettera \tag{19}$$

da cui la sua artificiosità.
Oltre allo **stdin** e allo **stdout** è possibile redirigere il flusso dei messaggi di errore dei comandi, flusso che di default è visualizzato sullo **stderr** ovvero il video. Se si vuole che i messaggi di errore dei comandi non si mescolino ai dati validi da essi prodotti è possibile redirigere lo **stderr** su un file. La redirezione dello **stderr** sfrutta il fatto che ai canali standard sono associati degli identificatori numerici in modo che si abbia:

1. allo **stdin** sia associato l'identificatore 0;
2. allo **stdout** sia associato l'identificatore 1;
3. allo **stderr** sia associato l'identificatore 2.

Per redirigere il solo **stderr** è necessario fare riferimento, quindi, al canale 2 secondo la sintassi seguente:

$$\#\, comando\ parametri\ 2>\ nome\_file \tag{20}$$

oppure:

$$\#\, comando\ parametri\ 2>>\ nome\_file \tag{21}$$

(in cui *parametri* individua sia le opzioni sia gli argomenti di *comando*) se non si vuole sovrascrivere il contenuto di $nome\_file$. Volendo separare dati ed errori si può procedere come segue:

$$\#\, comando\, parametri\ 1>>\ file\_dati\, 2>>\ file\_errori \tag{22}$$

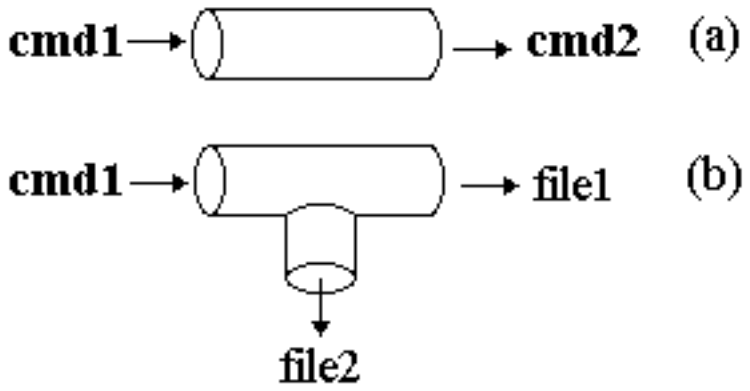

Figura 1: *pipe e tee, dettagli nel testo*

Se invece si usa la sintassi seguente:

$$\#\, comando\, parametri\ 1>\ file\ 2>\&1 \tag{23}$$

si ridirigono su $file$ sia i dati sia i messaggi di errore prodotti dal *comando*. Mediante l'uso di file creati tramite la redirezione è possibile passare dati da un comando ad un altro. Ad esempio si può avere:

$$\#\, cmd1\ >\ elenco \tag{24}$$

(in cui $cmd1$ in genere comprende parametri ed opzioni) che produce i dati in uscita nel file *elenco* e:

$$\#\, cmd2\ <\ elenco\ >\ elenco\_filtrato \tag{25}$$

(in cui $cmd2$ in genere comprende parametri ed opzioni) che elabora tali dati e ne produce una versione "filtrata" nel file $elenco\_filtrato$. Un modo più diretto, che non fa uso di file intermedi, è quello di fare ricorso all'operatore di **pipe** (vedi la figura 1 (a)) che permette di mettere in connessione, direttamente, lo **stdout** di un comando con lo **stdin** di un altro. La sintassi del comando è la seguente:

$$\#\, cmd1\ |\ cmd2 \tag{26}$$

La pipe connette lo **stdout** di **cmd1** con lo **stdin** di **cmd2** in modo da consentire la combinazione di comandi per l'esecuzione di compiti più complessi. L'uso più frequente della **pipe** è in associazione a comandi detti **filtri** quali:

1. **more**, visualizza un file una pagina (ovvero una schermata) alla volta;

2. **sort**, ordina lessicograficamente i dati in arrivo sullo **stdin** e li emette sullo **stdout**.

Ad esempio, se si vuole stampare un file il cui contenuto sia stato ordinato lessicograficamente e le cui righe sino state numerate in senso crescente si può usare la seguente combinazione di comandi:

$$\#\ sort\ elenco\ \mid\ cat - n\ \mid\ lpr - P\ printer \tag{27}$$

che fa uso di due pipe e di nessun file intermedio (**esercizio**: scrivere una combinazione di comandi che ottenga lo stesso risultato ma con file intermedi e senza usare la pipe). Un altro esempio è il seguente:

$$\#\ sort\ elenco\ \mid\ more \tag{28}$$

Se si vuole riferire lo **stdin** all'interno di un comando si può usare il carattere $-$. Usando tale riferimento si può inserire in un flusso dati l'output di un comando e inviare il tutto ad un altro comando:

$$\#\ pwd\ \mid\ cat -\ elenco \mid\ lpr - P\ printer \tag{29}$$

in modo da inserire il nome della directory corrente (in cui è contenuto il file *elenco*) in testa alla sua stampa, senza modificare in alcun modo il contenuto di tale file.
La figura 1 (b) illustra una pipe un po' più complessa, a tre vie, detta **tee**. La sintassi è la seguente:

$$\#\ cmd1\ \mid\ tee\ file2 \tag{30}$$

mentre lo **stdout** di **cmd1** rimane disponibile per essere rediretto o su un file (come $file2$) o su un altro comando.
In tal modo si riesce a redirigere l'output di un comando verso due destinazioni distinte. Se si usa:

$$\#\ cat\ file\ \mid tee\ nuovo\_file \tag{31}$$

si visualizza sullo **stdout** il $file$ e, allo stesso tempo, lo si copia in $nuovo\_file$. Se, invece, si ha:

$$\#\ sort\ file\ \mid tee\ nuovo\_file \mid\ lpr - P\ printer \tag{32}$$

si ordina lessicograficamente il contenuto di $file$ e, in parallelo, lo si salva nel file $nuovo\_file$ e lo si manda in stampa sulla stampante la cui coda di stampa si chiama $printer$. Si fa notare come la coda di default sia definibile

una volta per tutte assegnando in modo opportuno un valore alla variabile di ambiente $PRINTER$ (vedi oltre). Un esempio più complesso è il seguente:

$$\#\ sort\ file\ \mid\ cat\ -n\ \mid tee\ nuovo\_file \mid\ lpr - P\ printer \tag{33}$$

in cui si salva il contenuto ordinato lessicograficamente e numerato di $file$ in $nuovo\_file$ e lo si stampa. Cosa succede se, invece, si usa la seguente sequenza di comandi?

$$\#\ sort\ file \mid tee\ nuovo\_file\ \mid\ cat\ -n\ \mid\ lpr - P\ printer \tag{34}$$

1.2.4 Le variabili della shell $BASH$ e gli script (cenni)

La shell è caratterizzata sia da **variabili di ambiente** (cui è solo possibile assegnare valori) sia da variabili definite dall'utente e dette, pertanto, **variabili di utente**.
Le prime hanno una validità estesa tutta la sessione di lavoro e ad ogni login riassumono il loro valore di default mentre le seconde mantengono la loro validità dal momento in cui sono definite fino a quando non sono eliminate (o fino alla fine della sessione di lavoro ovvero fino al logout).
Le variabili di ambiente (detta anche di sistema) hanno nomi standard. Alcune variabili significative sono le seguenti:

1. $HOME$: contiene nome e posizione (vedi oltre) della directory principale (o home directory) dell'utente;

2. $SHELL$: contiene nome e posizione del programma che implementa la shell;

3. $PATH$ contiene separati dal carattere **;** nome e posizione delle directory al cui interno la shell ricerca i comandi di cui l'utente richiede l'esecuzione digitandone il nome;

4. $PS1$: contiene il prompt principale (o di prima linea) che indica all'utente che la shell è in attesa di un comando;

5. $PS2$: contiene il propmpt secondario (o di seconda linea) che indica che un comando si estende su più linee di input (ad esempio se si fa precedere il tasto **enter** da tasto di escape \);

6. $TERM$: contiene il tipo di terminale utilizzato come interfaccia all'interno della quale sono eseguiti i comandi, alcuni dei quali adeguano la loro esecuzione in base a tale tipo.

Altre variabili saranno introdotte e spiegate alla bisogna. In molti tipi di shell le variabili, una volta definite, devono essere **esportate** con il comando **export $VAR** in modo che il loro valore sia accessibile ai comandi che l'utente esegue all'interno della shell.
Per assegnare loro valori o per accedere ai loro valori si usano le stesse convenzioni che si usano per le variabili di utente per cui la trattazione si limiterà a queste ultime.
Le operazioni eseguibili sulle variabili sono le seguenti:

1. definizione;
2. assegnamento;
3. modifica;
4. accesso.

Una variabile viene definita semplicemente usando il suo nome. Il nome è un qualunque insieme di caratteri che non inizia per un numero e non contiene caratteri speciali (per la shell) quali !, & e lo spazio (che funge da separatore), avente un significato mnemonico. I nomi delle variabili possono contenere il carattere _. Abitualmente i nomi delle variabili sono definiti usando caratteri maiuscoli. Tale convenzione non sarà sempre rispettata nel seguito. Al momento della definizione, una variabile (di utente) assume come valore di default (ovvero in mancanza di un contemporaneo assegnamento) la stringa vuota.
Per assegnare un valore ad una variabile (operazione che può essere fatta contestualmente alla definizione) si usa l'operatore = secondo la sintassi seguente:

$$\#\ nome\_var = valore \tag{35}$$

(in cui # è il prompt $PS1$) senza nessun carattere spazio prima e/o dopo il segno di assegnamento =.
La modifica si traduce, semplicemente, nell'assegnamento di un nuovo valore ad una variabile.
L'accesso al valore di una variabile è reso possibile dall'operatore $. Ad esempio:

$$\begin{array}{l} \#\ PRINTER = lj1 \\ \#\ export\ \$PRINTER \\ \#\ echo\ \$PRINTER \\ lj1 \\ \#\ echo\ PRINTER \\ PRINTER \\ \# \end{array} \tag{36}$$

L'operatore $ consente di inserire il valore di una variabile in un comando come nell'esempio che segue:

$$\#\, lpr\ -P\ \$PRINTER\ nome\_file \tag{37}$$

Per conoscere il nome e il valore delle variabili di utente si può usare il comando **set** mentre il comando **setenv** fornisce le stesse informazioni per le variabili di ambiente. Se si vuole rimuovere una delle variabili di utente è sufficiente usare il comando:

$$\#\, unset\, nome\_var \tag{38}$$

che differisce dal comando:

$$\#\, nome\_var =''\ '' \tag{39}$$

dato che il secondo si limita ad assegnare alla variabile la stringa vuota mentre il primo la rimuove dall'ambiente della shell corrente. Si ritiene di fare notare che, al momento, ci si riferisce alla shell $BASH$ e che, quindi, molte cose non sono del tutto valide per altre shell, quali la $C-shell$. Si ricorda che è necessario eseguire l'**export** di una variabile se si vuole che sia accessibile ad i programmi che sono eseguiti all'interno della shell.
La shell vede alcuni caratteri come caratteri speciali:

1. il carattere spazio è un separatore di elementi sulla linea di comando;
2. i metacaratteri individuano insiemi di caratteri;
3. il . indica la directory corrente;
4. il $ consente di accedere al valore di una variabile;
5. i caratteri < e > rappresentano gli operatori di redirezione

Se si vuole che il valore di una variabile contenga uno o più caratteri speciali è possibile:

1. racchiudere tale valore fra una coppia di '' (in modo da "nascondere" alla interpretazione della shell tutti caratteri speciali meno il $);
2. racchiudere tale valore fra una coppia di ' (in modo da "nascondere" alla interpretazione della shell tutti caratteri speciali compreso il $);
3. far precedere ogni carattere speciale dal simbolo di escape \.

Alle variabili di utente è possibile assegnare anche il valore prodotto in uscita da un comando utilizzando la sintassi seguente:

$$\#\, nome\_var = `comando` \tag{40}$$

in cui il **comando** (comprese le opzioni e i parametri) è racchiuso in una coppia di apici inversi.
Un esempio pratico è il seguente:

$$\begin{array}{l} \#\, current\_dir = `pwd` \\ \#\, echo\ \$current\_dir \\ \backslash usr \setminus users \setminus cioni \setminus java\_project \\ \#\, list\_java = `ls * .java` \\ \#\, echo\ \$list\_java \\ main.java\ method.java \\ \# \end{array} \tag{41}$$

Si noti che:

1. gli apici inversi racchiudono un comando e consentono di assegnare il suo output ad una variabile di ambiente;
2. i singolo apici possono essere usati per:
   (a) impedire l'interpretazione dei caratteri speciali da parte della shell;
   (b) definire menmonici di comandi di largo uso.

Ad esempio il comando:

$$\#\, lsdir =' ls\ -F' \tag{42}$$

assegna un comando ad una stringa in modo che lo si possa richiamare come:

$$\#\, \$lsdir \tag{43}$$

così da ottenere l'elenco degli elementi contenuti nella directory corrente (vedi oltre) con i nomi delle directory seguiti da carattere /. In tal modo si creano sinonimi di comandi, sebbene la shell abbia metodi più eleganti per ottenere lo stesso risultato.
Modi più eleganti di creare mnemonici di comandi di uso corrente sono:

1. fare uso di **alias**;
2. fare uso di **script**.

Un **alias** è un modo in cui si associa una stringa ad un'altra così da creare un mnemonico per un comando di uso frequente (vedi oltre).
Uno script è un file che contiene comandi in successione e anche strutture di controllo e variabili. La trattazione completa degli script esula dallo scopo delle presenti note. In questa sede si sottolinea che:

1. uno script ha un nome che rappresenta il mnemonico che si vuole utilizzare;
2. perchè sia eseguibile come un comando qualunque, uno script deve essere:
   (a) "abilitato", ovvero reso eseguibile con il comando **chmod u+x nome_script** (vedi oltre);
   (b) "reso reperibile", ovvero inserito in una directory il cui nome compare fra i nomi contenuti nella variabile $PATH$ corrente.

Ad esempio:

$$\begin{array}{l} \#\ cat\ lss \\ ls * .java \\ \#\ cat\ lsc \\ ls * .class \\ \# \end{array} \tag{44}$$

sono due script per la realizzazione di mnemonici. Per eseguirli è possibile eseguire i passi suddetti oppure si possono seguire le due procedure indicate qui di seguito, utili soprattutto in fase di debugging di script molto più complessi di quelli visti:

1. $$\#\ .\ lss \tag{45}$$
   in modo da forzare l'esecuzione nella shell corrente;
2. $$\#\ sh\ lss \tag{46}$$
   in modo da forzare l'esecuzione in una shell eseguita nella shell corrente.

Gli script possono fare uso di argomenti passati sulla linea di comando. Tali parametri sono individuati all'interno degli script tramite le variabili:

1. $1 il primo,
2. $2 il secondo,

fino al nono.
Ad esempio:

$$\begin{array}{l} \#\ cat\ lst \\ ls * .\$1 \\ \# \end{array} \tag{47}$$

accetta un parametro sulla linea di comando e lo interpreta con una stringa che rappresenta l'estensione dei file contenuti nella directory corrente per cui se si ha:

$$\#\ .\ lst\ java \tag{48}$$

il comportamento è identico a quello di *lss* mentre se si ha:

$$\#\ .\ lst\ class \tag{49}$$

il comportamento è identico a quello di *lsc*. Nel primo caso la stringa *java* è associata alla variabile posizionale $1mentre nel secondo alla stessa variabile è assegnata la stringa *class*.
La variabile $0 contiene il nome dello script stesso mentre la variabile $∗ fa riferimento a tutti gli argomenti presenti sulla linea di comando e $# indica quanti sono tali argomenti.

1.3 Programmi e processi: comandi per la gestione dei processi

Un **programma** è un file che contiene comandi espressi secondo un linguaggio di programmazione. Esempi di programmi scrivibili dagli utenti sono gli script di shell. Gli script sono in genere interpretati all'interno di una shell. In modo simile, a tutti i comandi di sistema visti finora corrispondono invece programmi scritti in un qualche linguaggio di programmazione, tradotti in un formato eseguibile e resi eseguibili con il comando **chmod**.
Quando un programma viene mandato in esecuzione ad esso corrisponde, in genere, un **processo**. Ciò è vero per i comandi di sistema. Si possono avere eccezioni in cui l'esecuzione di un programma si traduce in più processi ma nelle presenti note manterremo, per semplicità, la corrispondenza un programma÷un processo. In modo analogo diremo che un processo equivale ad un **job**. Di solito si riserva il termine **job** per identificare i processi di utente come distinti dai processi di sistema.
In merito ai processi si hanno varie modalità di esecuzione che diremo:

1. in **background**;
2. in **foreground**.

Nel primo caso il processo accetta comandi e dati da tastiera e visualizza dati sul video, in molti casi all'interno di un xterm. In questo modo è possibile interrompere o sospendere l'esecuzione del processo con semplici comandi da tastiera. Con altri comandi è possibile trasformare la modalità di esecuzione di un processo da foreground a background (e viceversa).
I processi in background, viceversa, possono accedere al video ma non accettano comandi o dati da tastiera. Per evitare che i dati e i messaggi di errore prodotti dai processi in background si mescolino con quelli prodotti dai vari processi che l'utente esegue, in successione, in foreground è possibile usare gli operatori di redirezione visti per indirizzare i primi in file ad hoc che saranno esaminati solo in un secondo tempo.
In generale un utente ha in esecuzione più processi, alcuni in foreground e altri in background (in genere sono processi che lavorano su grosse moli di dati e che non necessitano di interazioni con l'utente). In più sono presenti molti processi di sistema fra cui molti girano in modalità background e sono detti **demoni**.
Per eseguire un processo in background si usa la seguente sintassi:

$$\begin{array}{l} \#\, comando\, [1 > [>]\, file\_dati\, 2 > \ file\_errori]\, \& \\ [1]\, 534 \\ \# \end{array} \qquad (50)$$

in cui *comando* include sia le opzioni sia gli eventuali parametri. A tale comando la shell risponde emettendo una coppia di valori che sono:

1. il numero di job dell'utente, [1] in questo caso;
2. il numero di processo nel sistema (o **pid** da **process identifier**), 534 in questo caso.

Il numero di job rappresenta il modo con cui l'utente può fare riferimento ad un suo processo.
Il **pid** rappresenta il modo in cui il SO identifica un processo. Il SO mette a disposizione il comando **jobs** per permettere all'utente di conoscere quanti e quali sono i suoi processi in background. Di ogni processo in background, il comando **jobs** fornisce:

1. l'identificativo numerico fra [];
2. lo stato, **Running** o **Stopped**;
3. se un processo è correntemente in esecuzione (segno +) o lo sarà (segno −);

4. il nome del comando corrispondente al processo.

Il SO comunica la conclusione di un processo in background solo al termine della esecuzione del comando in foreground corrente, senza interrompere la sua esecuzione. È possibile usare il comando **notify** con la seguente sintassi:

$$\#\, notify\ \%n \tag{51}$$

in cui n rappresenta il numero di job di cui si vuole conoscere immediatamente l'evento terminazione. Il numero di job consente di portare un processo dalla modalità background alla modalità foreground con il comando **fg**. La sintassi è la seguente:

$$\#\, fg\ \%n \tag{52}$$

Per far passare un processo dalla modalità foreground alla modalità background si deve procedere in due tempi:

1. lo si mette in stato di **stopped** (sospeso) con la combinazione di tasti **CTRL-Z**;

2. lo si manda in esecuzione in background con il comando **bg** usando la sintassi seguente:

$$\#\, bg \tag{53}$$

In tal modo si libera la shell corrente da un processo mandato in esecuzione in modalità foreground ma che non necessita di dati da tastiera.
Una volta che un processo è stato sospeso lo si può riattivare sia in foreground sia in background: nel primo caso si deve usare il comando **fg**, nel secondo si deve usare il comando **bg**. La presenza di job sospesi impedisce l'esecuzione della procedura di logout. In tal caso si devono riattivare i processi sospesi portandoli in modalità foreground in modo da terminarli prima di eseguire correttamente il logout.
Altri comandi utili per la gestione dei processi sono i seguenti:

1. **ps**,

2. **kill**.

Il primo fornisce lo stato dei processi (*p***rocess** *s***tatus**) di un utente. Di ogni processo il comando fornisce:

1. il **pid**;

2. l'identificativo del terminale da cui è stato mandato in esecuzione un processo (o l'indicazione ?? se al processo non è associato nessun terminale);

3. il tempo di CPU $TIME$ utilizzato per l'esecuzione del processo;

4. il nome del comando la cui esecuzione ha originato il processo.

Il secondo permette di terminare anticipatamente l'esecuzione di un processo in esecuzione in background oppure in foreground ma che non risponde ai comandi da tastiera e si trova in una condizione di **stallo**. Il comando ha le seguenti sintassi:

1. *kill %n*

2. *kill pid*

Nel primo caso si usa l'identificatore di job ottenibile con il comando **jobs**, nel secondo si usa il **pid** ottenibile con il comando **ps**.

1.4 Il file system e la gestione dei file

Il file system del SO Linux ha una struttura gerarchica ad albero basata sul concetto di **directory**. Una directory è un contenitore di file ed altre directory. Ogni directory contiene:

1. un riferimento alla directory stessa indicato dal mnemonico .

2. un riferimento alla directory di livello superiore, indicato dal mnemonico ..

Il fatto che la struttura sia ad albero significa che esiste un contenitore di livello più alto al cui interno sono contenute tutte le altre directory, siano esse locali oppure remote ed accedute tramite protocolli di rete. Il contenitore di livello più alto è una directory particolare detta **root** e indicata con il carattere "/". Per tale directory il riferimento alla directory di livello superiore (che non esiste) si traduce in un riferimento alla directory stessa.

1.4.1 I file in Linux

All'interno del file system si hanno file e directory. Da un punto di vista logico, file e directory hanno una struttura diversa, dal momento che i secondi possono essere considerati come elenchi di riferimenti a file e ad altre directory mentre i secondi individuano aree di memorizzazione sui supporti di memoria. Da un punto di vista fisico le due tipologie non si differenziano poi molto. Per comodità espositiva faremo riferimento alla struttura logica per cui manterremo evidente la differenziazione.
Directory particolarmente significative per gli utenti sono:

1. la **home directory**, ovvero la directory principale di un utente, al cui interno l'utente memorizza i propri dati e i propri programmi e al cui interno può creare altre directory in modo da definire una sua struttura gerarchica con radice nella propria **home directory**:

2. la **directory corrente**, su cui torneremo in seguito, e che individua la posizione corrente all'interno della struttura gerarchica del file system. Al momento del login la directory corrente di default è la home directory di ogni utente.

All'interno del file system, file e directory sono individuati da:

1. un nome;

2. una posizione, assoluta (se riferita alla root) o relativa (se riferita alla directory corrente).

Il nome di file e directory può essere lungo al più 256 caratteri comprensivi di caratteri alfabetici, numerici e caratteri speciali quali _ (underscore), . (punto, che può essere il primo carattere in casi particolari), (virgola) mentre non può iniziare con un carattere numerico. Il carattere . può essere usato per suddividere il nome di un file in due parti:

1. una prima del . che corrisponde al nome vero e proprio,

2. una dopo il . che rappresenta una sorta di estensione che permette di individuare il tipo di un file ma che non ha un reale significato per il SO Linux a differenza di quello che accade, ad esempio nel caso del SO Windows.

I file il cui nome inizia con il carattere . sono, in genere file di configurazione. Vedremo qualcosa al proposito in una delle sezioni che seguono.

1.4.2 Tipi di file in Linux

All'interno del SO Linux tutti i file hanno la stessa struttura fisica ovvero lo stesso formato fisico: una successione di byte. In tal modo si implementano tutte le componenti del SO mediante un numero limitato di tipi di file:

1. file comuni;

2. file di tipo directory che contengono informazioni relative al contenuto di una directory;

3. file per i dispositivi fisici, di due tipi:

(a) a caratteri;

(b) a blocchi.

Il SO mettere a disposizione degli utenti alcuni comandi per la caratterizzazione delle differenze fra i file in modo da discriminare fra:

1. directory;
2. file di testo;
3. file binari.

Il comando **file** ammette la seguente sintassi:

$$\#\ file\ set\_di\_nomi\_file \tag{54}$$

permette di sapere il "tipo" di ciascuno dei file di cui si specifica il nome. Fra i tipi possibili si ha:

1. **text**;
2. **directory**;
3. **executable**;
4. **C program text**;
5. **empty**;

ed altri il cui significato è facilmente intuibile. L'opzione $-f$ seguita dal nome di un file consente l'esame del tipo dei file i cui nomi sono contenuti in tale file.
L'accesso alla struttura interna di file non di testo (ovvero il cui tipo non è caratterizzato dalla parola chiave **text**) può essere fatto usando il comando **od** (**octal dump**) che consente di visualizzare sullo schermo (o redirigere ad altri comandi o file) il contenuto di un file qualunque i cui byte sono visualizzati nella rappresentazione:

1. ottale (default), (opzione $-o$);
2. esadecimale, (opzione $-x$);
3. decimale, (opzione $-d$);
4. a caratteri, (opzione $-c$).

1.4.3 La struttura del file system (fs)

La struttura del fs è molto semplice e "pulita": ogni copia del SO ha una directory **root** (radice), individuata dal carattere /, al cui interno sono contenuti:

1. file di vario tipo;
2. altre directory, dette di sistema e inaccessibili (se non indirettamente) ai normali utenti.

Da un punto di vista sintattico si ha:

$$dir := \ldots file * dir* \tag{55}$$

ovvero una directory contiene sempre i due riferimenti logici alla directory stessa e alla directory di livello superiore oltre ad un numero qualunque di altri file (anche nessuno) e directory (anche nessuna). Il carattere / compare nelle stringhe che individuano la posizione di un file o una directory all'interno della struttura del fs in modo da separare le varie directory fra di loro. Se tale stringa (nel seguito **pathname**) inizia per / si riferisce alla **root** ed è detta **pathname assoluto**, se non inizia per / si riferisce alla **directory corrente** ed è detta **pathname relativo**. Se un pathname relativo inizia per . vuol dire che si riferisce (in modo ridondante, perchè?) alla directory corrente mentre se inizia per .. si riferisce alla directory di livello superiore (o directory padre).
All'interno del fs sono contenute tutte le directory e i file locali e le directory mediante le quali, con operazioni di **mount**, è possibile accedere, ad esempio, a dispositivi removibili locali, quali **dischetti** o **CD-rom**, oppure ad altri dischi locali o a dischi remoti, acceduti tramite protocolli di rete, quali NFS e simili.
Le directory sono individuate nel fs in basa alla loro posizione relativa rispetto alla root. Si parla di **sottodirectory di primo livello** per fare riferimento alle directory direttamente contenute nella root. Tali directory sono, di solito, directory di sistema e contengono i programmi e i file propri del SO. Fra tali directory si segnalano le seguenti:

1. **dev** (pathname assoluto **/dev**), contiene i file a blocchi e a caratteri associati a tutti i dispositivi hardware del sistema di calcolo;
2. **bin** (pathname assoluto **/bin**), contiene alcuni comandi di sistema;
3. **etc** (pathname assoluto **/etc**), contiene alcuni file di configurazione del sistema;

4. **home** (pathname assoluto **/home**), contiene le home directory degli utenti.

In molti casi si usa la directory *usr* o *users* per contenere le home directory degli utenti mentre in altri si accede a home directory definite su dispositivi remoti in modo che un utente possa accedere sempre allo stesso spazio di memorizzazione, indipendentemente da quale sia il sistema fisico su cui esegue il login.
Le directory contenute in */usr* sono dette, pertanto, di secondo livello. Fra queste si segnalano:

1. **bin** (pathname assoluto **/usr/bin**, pathname relativo **bin**), contiene i programmi e le utility accedute dagli utenti;

2. **sbin** (pathname assoluto **/usr/sbin**, pathname relativo **sbin**), contiene comandi di amministrazione del sistema;

3. **lib** (pathname assoluto **/usr/lib**, pathname relativo **lib**), contiene le librerie dei linguaggi di programmazione;

4. **spool** (pathname assoluto **/usr/spool**, pathname relativo **spool**), contiene i file mandati in stampa dagli utenti.

La **home directory** (*hd*) ha un nome che coincide, di solito, con l'identificativo che si usa per eseguire il login (o **userid**). Ogni utente ha piena proprietà della sua hd e al suo interno può creare sia nuovi file sia nuove directory. La hd è la directory corrente al momento del login ed è la directory in cui si viene riposizionati, di default, ogni volta che si esegue il comando *cd* senza argomenti (vedi oltre). La directory corrente è detta anche **directory di lavoro** dato che contiene gli elementi accessibili direttamente specificando il loro solo nome. Si ricorda che si ha:

$$pathname := [.[.]][/][dir/]^{*}nome \tag{56}$$

in cui le [] individuano, come di consueto, elementi opzionali, il carattere] indica 0 o più ripetizioni dell'elemento che lo precede, *dir* individua una qualunque sottodirectory "intermedia" e *nome* indica il file o la directory a cui ci si vuole riferire.

1.4.4 Alcuni comandi per l'accesso ai file

Nella presente sezione si elencano alcuni dei comandi che il SO mette a disposizione degli utenti per la gestione dei file. Le principali operazioni di gestione includono:

1. la visualizzazione del contenuto dei file;
2. la stampa del contenuto dei file o delle directory;
3. la manipolazione (creazione, cancellazione, spostamento, copia...) di file e directory.

Per la visualizzazione del contenuto dei file si hanno i comandi **cat** e **more** con la sintassi[1]:

$$\#\ \{cat\ \mid\ more\}\ nomi\_file \tag{57}$$

Il primo visualizza il contenuto di tutti i file i cui nomi sono contenuti in $nomi\_file$ come un flusso continuo di dati sullo schermo. Se si vuole interrompere e far ripartire tale flusso si possono usare le combinazioni di tasti **CTRL-S** (stop) e **CTRL-Q** (start) oppure si può usare il secondo comando che visualizza tale flusso una pagina alla volta: l'utente può passare alla successiva premendo la barra spaziatrice e può anche interrompere la visualizzazione. Altri comandi sono **head** (visualizza le righe iniziali di un file), **tail** (visualizza di default le ultime 10 righe di un file) e **less** (simile a more ma consente di andare avanti e indietro nella visualizzazione di un file), per i quali si rimanda alla documentazione in linea.
Per la stampa dei file si ha il comando:

$$\#\ lpr\ [-P\ nome\_stampante]\ nomi\_file \tag{58}$$

che consente di stampare i file i cui nomi sono contenuti in $nomi\_file$ sulla stampante di default (ovvero il cui nome è contenuto nella variabile di ambiente $PRINTER$) o sulla stampante specificata. Il comando:

$$\#\ lpq\ [-P\ nome\_stampante] \tag{59}$$

permette di conoscere quali sono i file in attesa di essere stampati sulla stampante di default o su quella specificata. Ogni file ha associato un ID numerico oltre ad un proprietario. L'ID numerico può essere usato, se se ne hanno i diritti, per rimuovere il file dalla coda di stampa con il comando:

$$\#\ lprm\ [-P\ nome\_stampante]\ ID \tag{60}$$

I principali comandi per la gestione delle directory sono i seguenti:

1. **mkdir nome_dir**, crea la directory $nome\_dir$ dove $nome\_dir$ può contenere un pathname assoluto o relativo;

[1]Si ricorda che con $\{a \mid b\}$ si definisce una alternativa fra a e b in cui uno dei due elementi deve essere presente mentre con $[a \mid b]$ si definisce una alternativa fra a e b in cui entrambi gli elementi possono essere assenti.

2. **rmdir nome_dir**, rimuove la directory *nome_dir* (se vuota) dove *nome_dir* può contenere un pathname assoluto o relativo;

3. **pwd**, visualizza il pathname assoluto della directory corrente o di lavoro;

4. **cd [nome_dir]**, cambia la directory di lavoro in quella specificata (tramite un pathname assoluto o relativo) o nella home directory se manca il parametro;

5. **ls [opzioni] [nome_dir/][nomi_file]**, visualizza il contenuto della directory corrente o specificata (*nome_dir*) oppure solo gli elementi i cui nomi contenuti in $nomi\_file$, individuabili anche mediante l'uso di metacaratteri. Alcune opzioni significative (combinabili fra di loro) del comando ls sono le seguenti:

   (a) $-a$ visualizza anche gli elementi i cui nomi iniziano per .;

   (b) $-l$ visualizza le informazioni ausiliarie degli elementi, quali nome del proprietario e diritti di accesso (vedi oltre);

   (c) $-F$ visualizza le directory facendo seguire il nome dal carattere /.

Si ricordi che una directory che contiene solo i riferimenti alla directory stessa e alla directory di livello superiore è da intendersi vuota. Si fa notare che il carattere $\sim$ rappresenta un riferimento mnemonico al pathname assoluto della home directory e può comparire in tutti i casi in cui sia lecito usare tale pathname assoluto.
Altri comandi utili per la gestione di file sono quelli che permettono di:

1. copiare;

2. cancellare;

3. spostare;

4. assegnare nomi aggiuntivi;

ai file. I comandi sono, nell'ordine: **cp**, **rm**, **mv** e **ln**. Volendo ricercare i file all'interno del fs a partire da una ben precisa posizione di può usare il comando **find**. È ovvio come non sia possibile esaminare in queste note tutte le opzioni dei vari comandi (allo scopo sono stati predisposti il comando **man** e altri analoghi e sono stati scritti ponderosi manuali). In quanto segue ci limiteremo a presentare alcuni esempi di utilizzo.

Il comando **find** consente di eseguire ricerche all'interno del fs. La sua sintassi è piuttosto complessa, essendo il comando molto potente e versatile:

$$\#\ find\ elenco\_directory\ \ -opzioni\ \ criteri \tag{61}$$

Il parametro *elenco_directory* di solito rappresenta la radice della porzione di fs all'interno della quale si vuole effettuare la ricerca. Di solito si usa / (la root) e . (la directory corrente).
Opzioni significative sono:

1. $-name\ nome\_file$ permette di eseguire la ricerca dei file individuati per nome, come $'*.java'$ o $'*.class'$ (gli apici singoli impediscono la valutazione del metacarattere $*$ da parte della shell);

2. $-print$ fa in modo che i nomi (completi di pathname assoluto) trovati siano visualizzati;

3. $-type\,t$ permette di individuare i file in base al tipo, se $t = d$ si ricercano le directory, se $t = l$ si ricercano i link simbolici (vedi oltre), se $t = f$ si ricercano i file normali e così via;

4. $-size$ per eseguire la ricerca in base alla dimensione dei file;

5. $-mtime$ per eseguire la ricerca in base alla data di ultima modifica.

Se si fanno precedere le opzioni (combinabili fra di loro) dal carattere ! si esegue la negazione del criterio specificato. È inoltre possibile combinare i criteri con gli operatori logici **AND**, **OR**: il primo è rappresentato utilizzando una coppia \(e \). Ad esempio:

$$\backslash(-name'*.c' - mtime + 3\backslash) \tag{62}$$

permette di ricercare tutti i file di tipo *.c* modificato più di tre giorni fa. l'operatore OR è rappresentato con l'opzione $-o$. In ogni caso è bene racchiudere le espressioni che contengono gli operatori logici fra coppie di parentesi quotate, ovvero \(e \).
Per la **copia** di un file si può usare il comando **cp** che lascia l'originale inalterato. Le forme sintattiche possibili sono le seguenti:

1. **cp file_origine file_destinazione** copia il primo file nel secondo (sovrascrivendolo se esiste);

2. **cp nomi_file directory** copia i file specificati nella directory indicata, con le stesse caratteristiche del caso precedente;

3. **cp** $-r$ **vecchia_dir nuova_dir** copia una directory e tutto il suo contenuto nella directory specificata duplicando un porzione del fs.

L'opzione $-i$ rappresenta una salvaguadia per impedire la sovrascrittura di file esistenti: in tal caso il comando chiede esplicitamente all'utente di autorizzare la sovrascrittura.
Per cambiare nome ad un file o ad una directory oppure per spostare un file da una directory ad un'altra si può usare il comando **mv**. Per questo comando valgono molte delle considerazioni fatte a proposito del comando **cp**, al quale si rimanda. Le varie sintassi sono:

1. **mv file_origine file_destinazione**
2. **mv nomi_file directory**
3. **mv vecchia_dir nuova_dir**

per i quali valgono le considerazioni fatte per il comando *cp* se si sostituisce il verbo copiare con il verbo spostare. Si noti che una operazione di spostamento equivale alle seguenti operazioni in cascata:

1. copia;
2. cancellazione degli originali.

In tal modo si ha che il comando **mv a b** equivale a **cp a b; rm a** (vedi oltre). Nei comandi visti si può usare il carattere $\sim$ che rappresenta il pathname assoluto della home directory dell'utente e che può comparire in tutte le posizioni in cui è lecito specificare il pathname assoluto di una directory. Il comando **rm** consente di cancellare file e directory (anche non vuote, contrariamente al comando **rmdir**) Le sintassi possibili sono le seguenti:

1. **rm nomi_file** rimuove i file i cui nomi sono specificati (anche mediante metacaratteri);
2. **rm -r nome_dir** rimuove tutta la porzione di fs la cui root è la directory *nome_dir* (pericolosissimo).

Il comando **ln**, infine, rappresenta in modo per assegnare nomi diversi ad uno stesso file in modo che rappresenti un'area di memoria accessibile da più posizioni del fs e il cui contenuto sia sempre lo stesso da qualunque punto lo si acceda per modificarlo o per leggerlo. I nomi aggiunti sono detti **link** e, vedremo, sono di due tipi:

1. link simbolici;

2. link fisici.

La sintassi del comando è la seguente:

$$\#\, ln\ nome\_file\_esistente\ \ nome\_aggiunto \tag{63}$$

in modo che l'area di memoria associata al file *nome_file_esistente* sia accessibile anche mediante il nome *nome_aggiunto*. L'opzione $-l$ del comando *ls* elenca per ogni file o directory il numero di link che fanno ad esso riferimento. Se ad un file sono stati assegnati due nomi aggiunti (ovvero due link) il numero di link ottenibile come primo campo numerico da sinistra su ogni linea in output del comando **ls -l** dovrebbe valere 3.
I nomi aggiunti sono di solito usati per riferire uno stesso file da directory diverse facendo in modo che da qualunque parte lo si modifichi le modifiche si riflettano sugli stessi dati.
Un altra forma sintattica del comando **ln** è la seguente:

$$\#\, ln\ nome\_file\_esistente\ \ nome\_directory \tag{64}$$

in modo che si abbia un link nella directory specificata (e diversa dalla directory corrente) con lo stesso nome del file originario. Data la possibile esistenza di link ad un file per rimuovere effettivamente un file è necessario cancellare tutti i link che fanno ad esso riferimento. Il comando *rm* si limita a rimuovere uno dei link che fa riferimento ad un file in modo che il file sia effettivamente cancellato (e l'area dati ad esso allocata sia effettivamente liberata) solo quando anche l'ultimo link è stato rimosso.
I link visti finora sono detti **link fisici** o **hard link**. Linux consente di definire i cosiddetti **link simbolici**. I link fisici hanno la seguente limitazione: non possono riferire file localizzati in fs remoti o memorizzati su dispositivi fisici distinti anche se collegati, mediante una directory ad hoc, alla struttura gerarchica locale. Il fs locale ha, infatti, una root e un certo numero di directory, dette **mount point**, alle quali è possibile collegare la root di fs remoti (acceduti tramite protocolli di rete) oppure localizzati su dispositivi removibili quali CD-rom, dischetti e simili. Una volta eseguito il collegamento, l'utente vede il fs globale non come una entità composita ma come una entità omogenea, con la limitazione vista per i link.
I **link simbolici** consentono di superare questa limitazione.Un link simbolico:

1. contiene il pathname assoluto (o relativo) del file verso cui si attiva il link;

2. contiene tutte le informazioni necessarie per individuare un file e rappresenta un modo alternativo in cui si può scrivere il nome di un file.

La sintassi è la seguente:

$$\#\, ln\ nome\_riferimento\ \ nome\_file\_esistente \tag{65}$$

Se, con il comando **ls -l**, si visualizzano le informazioni relative a *nome_riferimento* e *nome_file_esistente* si ottengono informazioni diverse in merito alla dimensione (minore per il link simbolico che per il file "vero") e al tipo di file: il primo viene dichiarato di tipo link (o *l*, vedi oltre) mentre il secondo è dichiarato essere un file ordinario (o −, vedi oltre). La rimozione di un file è causata dalla rimozione dei link fisici mentre non è necessario rimuovere i link simbolici che diventano inutilizzabili a seguito della rimozione, dato che riferiscono un file non più esistente.
I link simbolici possono riferire directory, per come sono definiti: in questo caso il comando **pwd** visualizza il pathname assoluto della directory e non il relativo nome simbolico.Se si vuole conoscere il nome del link simbolico alla directory corrente (se esiste) si può visualizzare il nome della **directory corrente** contenuto nella variabile **cwd**:

$$\#\, echo\ \$cwd \tag{66}$$

A cose normali il comando *pwd* e il comando suddetto danno luogo allo stesso output che differisce solo in presenza di un link simbolico utilizzando il quale si è acceduta la directory corrente.

1.5 La gestione dei file in Linux

In questa sezione si esaminano alcuni comandi messi a disposizione degli utenti dal SO Linux per svolgere sui file i compiti seguenti:

1. conoscere informazioni dettagliate relative al tipo di file, ai diritti di accesso, al nome del proprietario, al numero di link e così via;
2. modificare i permessi di accesso a file e directory;
3. accedere a file e directory localizzati su dispositivi di memorizzazione removibili o non locali;
4. archiviare file e directory in un unico file previe operazioni di compressione che ne riducono le dimensioni.

1.5.1 Sapere tutto su file e directory

Il comando ls con l'opzione $-l$ (in combinazione con l'opzione $-a$ che consente di visualizzare anche i file il cui nome inizia per .) permette di visualizzare, per ognuno degli elementi contenuti nella directory corrente, le seguenti informazioni (visualizzate su ciascuna delle righe di output da sinistra verso destra):

1. tipo dell'elemento;
2. permessi di accesso;
3. numero di link;
4. nome del proprietario (come noto al SO);
5. nome del gruppo cui appartiene il proprietario;
6. dimensione dell'elemento in byte;
7. data e ora di ultima modifica;
8. nome dell'elemento.

Le forme sintattiche sono:

$$\#\, ls\ -l \tag{67}$$

per il contenuto della directory corrente;

$$\#\, ls\ -l\, nome\_file \tag{68}$$

per le informazioni relative ad un file (o ad un gruppo di file se il $nome\_file$ contiene dei metacaratteri);

$$\#\, ls\ -ld\, nome\_dir \tag{69}$$

per le informazioni relative ad una directory.
I tipi possibili per un elemento sono i seguenti:

1. $-$ se è un file ordinario;
2. d se è una directory;
3. l se è un link simbolico;
4. c se è un riferimento ad un dispositivo fisico a caratteri;

5. b se è un riferimento ad un dispositivo fisico a blocchi;

6. p se è un elemento di tipo pipe.

I permessi di accesso sono rappresentati con tre gruppi di tre caratteri ciascuno. Il primo gruppo di tre caratteri, da sinistra, è relativo ai permessi del proprietario (u per user), il secondo è relativo al gruppo (g per group) a cui il proprietario appartiene e il terzo a tutti gli altri utenti del sistema (o per others). Il proprietario è l'utente che crea l'elemento in oggetto.
Schematicamente si ha, quindi:

$$uuugggooo \tag{70}$$

In ogni tripletta il primo carattere si riferisce al diritto di lettura (r), il secondo al diritto di scrittura (w), il terzo al diritto di esecuzione (x). Vedremo che significato hanno tali diritti per file e per directory. Se il diritto è concesso al tipo di utente cui si riferisce la tripletta allora in quella posizione vi compare il carattere apposito, se il diritto è negato vi compare il carattere $-$.
Ad esempio, se ho, relativamente ad un file:

$$rw-r--r-- \tag{71}$$

ciò significa che:

1. il proprietario può accedere al file in lettura (ovvero visualizzarlo, ad esempio, con il comando *cat*) e in scrittura (ovvero editarlo e salvare le modifiche apportate al file);

2. gli utenti che sono nel suo gruppo e tutti gli altri (ovvero quelli non nel suo gruppo) possono accedere al file in lettura;

3. nessuno ha il diritto di eseguire il file. Se il file contiene codice eseguibile e lo si volesse eseguire è necessario aggiungere tale diritto alle categorie di utenti cui si vuole garantire tale diritto.

Qualora si avesse la situazione seguente:

$$rwxr-xr-x \tag{72}$$

relativamente ad una directory, ciò significherebbe che:

1. il proprietario può elencare il contenuto della directory (diritto di lettura), può modificarla (diritto di scrittura) creando e rimuovendo al suo interno file e directory e può usare la directory all'interno di un pathname per raggiungere altre porzioni del fs (diritto di esecuzione);

2. gli utenti del gruppo del proprietario e gli altri possono solo accedere in lettura e in esecuzione a tale directory.

I permessi di accesso hanno valori di default per file e directory che sono assegnati automaticamente ai nuovi elementi al momento della loro creazione con uno dei molti comandi disponibili.
Per modificare i diritti (o permessi) di accesso ad un file o ad una directory si può usare il comando **chmod**, mentre per cambiare il proprietario di un file o di una directory si può usare il comando **chown**. Il comando **chgrp** consente di cambiare il gruppo cui un utente appartiene, a patto che ciò sia possibile.
Il comando **chmod** permette di cambiare i permessi di accesso utilizzando la notazione simbolica, che abbiamo visto in precedenza, o la notazione ottale (o in base otto), sfruttando il fatto che si tratta di triplette di caratteri ciascuno dei quali può assumere due valori: o il carattere $-$ se il diritto corrispondente è negato o un carattere fra i seguenti r, w, x se il corrispondente diritto è accodato. Senza entrare in dettagli, dato che faremo riferimento alla notazione simbolica, si fa notare come:

$$rw-r--r-- \tag{73}$$

corrisponda in base otto a:

$$644 \tag{74}$$

mentre:

$$rwxr-xr-x \tag{75}$$

corrisponde a:

$$755 \tag{76}$$

La notazione ottale è più sintetica ma può presentare difficoltà ai novizi per cui non verrà approfondita ulteriormente.
Il comando **chmod** in pratica stabilisce quali diritti concedere (con il carattere $+$) o negare (con il carattere $-$) e a chi. Il chi viene individuato dai caratteri seguenti:

1. u indica il proprietario;
2. g individua il gruppo cui appartiene il proprietario;
3. o individua i restanti utenti.

Se nessuno di tali elementi è specificato, la modifica riguarda tutti e tre le categorie suddette ovvero tutti gli utenti indistintamente.
È ovvio che solo il proprietario di un elemento può cambiarne i diritti di

accesso.
La sintassi del comando **chmod** è abbastanza complessa per cui partiremo da alcuni esempi.

1. **chmod +x-w nome_file** aggiunge per tutti il diritto di esecuzione $(+x)$ e toglie il diritto di scrittura $(-w)$ su $nome\_file$;

2. **chmod go-rw+x nome_file** toglie, per il gruppo e gli altri, i diritti di lettura e scrittura su $nome\_file$ mentre concede il diritto di esecuzione;

3. **chmod go-rw+x nome_file; chmod u+rwx nome_file** il primo comando toglie, per il gruppo e gli altri, i diritti di lettura e scrittura su $nome\_file$ mentre concede il diritto di esecuzione mentre il secondo concede tutti i diritti al proprietario. In notazione ottale basterebbe scrivere **chmod 711 nome_file**.

La sintassi è la seguente:

$$\#\, chmod\, [ugo][+rwx][-rwx]\, nome\_file \tag{77}$$

in cui almeno uno dei due gruppi che contengono i diritti da concedere o quelli da negare deve essere presente.
Oltre ai diritti di accesso a file e directory è possibile modificare:

1. il proprietario, con il comando:

$$chown\ nuovo\_proprietario\ \ nome \tag{78}$$

   in cui *nome* può individuare più elementi (con i metacaratteri) e *nuovo_proprietario* ù il nome con cui un utente diverso dal proprietario è noto al SO, ovvero è uno **user-id** locale;

2. il gruppo, con il comando:

$$chgrp\ nuovo\_gruppo\ \ nome \tag{79}$$

   in cui *nome* può individuare più elementi (con i metacaratteri). In questo caso è necessario che l'utente appartenga al gruppo al quale vuole "transitare" nei confronti di *nome*. La modifica riguarda solo l'elemento corrente e non gli altri di proprietà dello stesso utente ed è un modo per condividere certi file o directory ma non altri con alcuni utenti.

Una volta che il comando **chown** è stato eseguito l'utente perde ogni controllo sull'elemento a cui cambia il proprietario per cui ogni modifica è prudente che sia apportata prima che tale comando sia eseguito in modo da non avere difficoltà con i diritti di accesso assegnati ai membri del gruppo o agli altri prima della modifica di proprietario. Si noti che il nuovo proprietario può appartenere a qualunque gruppo, anche diverso da quello cui appartiene il vecchio proprietario.

1.5.2 Come accedere ai file system: mount e umount

Il fs di Linux (come di tutti i vari "sapori" di Unix), sebbene si presenti come una unica struttura ad albero, è composto da fs che possono trovarsi su dispositivi fisici distinti quali:

1. dispositivi (dischi fissi) remoti;
2. dispositivi removibili (dischetti o CD-rom).

Si ha, quindi, un insieme di file system, ciascuno con la propria root. Per trasformare tale insieme in una gerarchia unica con una sola root è necessario:

1. stabilire una delle root come root del fs globale;
2. collegare le altre root alla gerarchia mediante sottodirectory dedicate (di regola vuote) della root globale e dette **mount point**.

L'operazione di collegamento è, infatti, una operazione di **mount** eseguita con il comando **mount**. L'operazione di mount permette di collegare un fs localizzato su un dispositivo di uno dei tipi visti ad una sottodirectory della root globale e presuppone che sul dispositivo cui si vuole accedere sia presente un fs almeno compatibile con quello di Linux. L'operazione di **mount**, per essere eseguita, richiede il possesso di diritti particolari (vedi oltre) e, pertanto, non è eseguibile, se non in alcuni casi particolari (accesso a dischetti e CD-rom), dai normali utenti.
La sintassi del comando è la seguente:

$$\#\ mount\ [opzioni]\ device\ mount\_point \tag{80}$$

Il *device* individua un elemento nella directory */dev* che permette la gestione a blocchi di un dispositivo fisico mentre il *mount_point* è una sottodirectory della root globale a cui si collega la root del fs che si stà collegando il modo che tale root (secondaria) e tutte le sue sottodirectory siano accessibili come sottodirectory di *mount_point*.
Le opzioni principali sono:

1. $-f$ esegue un mount fittizio e permette di verificare se un dispositivo contiene o meno un fs "montabile";
2. $-v$ esegue il comando presentando informazioni dettagliate (verbose) sulle operazioni di "basso livello" eseguite dal comando mount;
3. $-w$ collega il fs in modalità lettura e scrittura;
4. $-r$ collega il fs in modalità sola lettura;
5. $-n$, $-t$ *tipo*, $-a$, $-o$ *opzioni* che saranno descritte a breve.

L'operazione duale della **mount**, che permette di rimuovere un fs dalla gerarchia globale ed è eseguibile solo se nessun utente o processo ha come directory corrente una delle directory della gerarchia del fs su cui si esegue tale operazione.
La sintassi del comando **umount** che esegue tale operazione di "scollegamento" è la seguente:

$$\#\ umount\ device\ mount\_point \tag{81}$$

Ogni fs, al momento in cui lo si definisce, ha associata una dimensione, limitata dalla dimensione del dispositivo fisico ospite. Tale spazio sarà occupato da file e directory nel corso della "vita" del fs. Per conoscere lo "stato" di "impegno" dello spazio assegnato ad un fs si può usare il comando **df** la cui sintassi è la seguente:

$$df\ [nome\_dir] \tag{82}$$

In assenza dell directory il comando fornisce una lista del fs su cui si è eseguita una operazione di mount e per ciascuno di essi fornisce:

1. il *device*;
2. la **dimensione** in numero di blocchi di 1024 byte ciascuno;
3. la quantità di spazio occupato;
4. la quantità di spazio libero;
5. la percentuale di spazio occupato sul totale;
6. il *mount_point*.

Se invece si specifica una directory si riesce a sapere a quale fs appartiene la directory specificata. Se si vuole eseguire una verifica (eventualmente congiunta ad operazioni di riparazione) di un fs si può eseguire il comando **fsck** la cui sintassi è:

$$\#\ fsck\ [opzioni]\ device \tag{83}$$

Il *device* rappresenta il dispositivo fisico (e non il mount point) su cui è presente il fs che si vuole controllare mentre per le opzioni più significative si rimanda alla documentazione in linea o a [Pet96]. Si fa notare che in molti casi i valori di default delle opzioni siano accettabili per controlli di routine per cui non è necessario specificarli.
Si fa notare che, in presenza della GUI, le operazioni di **mount** per i supporti removibili sono eseguite automaticamente al momento in cui il dispositivo viene inserito nel lettore appropriato mentre l'operazione di **umount** sia eseguibile, sempre che il dispositivo non sia "occupato" (nel senso prima specificato), semplicemente trascinando l'icona corrispondente sull'icona cestino o eseguendo, avendo selezionato l'icona rappresentativa del dispositivo, il comando **eject** (espelli). Si ritiene che una conoscenza di "basso livello" di tali comandi sia utile per avere gli strumenti per intervenire in caso di problemi.

1.5.3 Come accedere ai dischetti

L'accesso ai dischetti avviene, di solito, tramite il device **/dev/fd0**. Se il sistema ha più di una di tali unità, queste sono individuate dai numeri 0, 1 e così via in modo che sia necessario specificare il corretto valore per individuare il dispositivo che si vuole effettivamente utilizzare. Nel caso dei dischetti l'accesso avviene utilizzando mount point dedicati. Un esempio di comando è il seguente:

$$\#\ mount\ /dev/fd0\ /mnt/diskette \tag{84}$$

Tale comando presuppone che sul dischetto sia presente un fs Linux compatibile. Se si vuole accedere a dischetti creati in ambiente Windows (non attraverso la GUI che gestisce l'accesso in modo trasparente) si deve far uso dei cosiddetti **mtools** la cui trattazione esula dalle presenti note.
L'operazione di mount collega il fs del dischetto al fs globale mediante la directory **/mnt/diskette**. Per poter rimuovere il dischetto corrente ed inserirne un'altro è necessario che il dispositivo non sia "occupato" ed eseguire il comando **umount**:

$$\#\ umount\ /dev/fd0\ /mnt/diskette \tag{85}$$

Qualora si stia utilizzando un dischetto nuovo (o predisposto per un SO diverso) può essere necessario creare su di esso un fs con un comando apposito (che esegue la cosiddetta operazione di **formattazione**). Nel caso dei dischetti il comando prevede come parametri:

1. il device che contiene il dischetto da formattare;
2. la capacità in kilobyte.

Nel nostro caso il comando da usare è il seguente:

$$\#\, mkfs\ /dev/fd0\ 1400 \tag{86}$$

Il parametro 1440 corrisponde alla capacità standard di 1.44 MByte.
La formattazione, che cancella il dischetto rimuovendo i dati su di esso contenuti, viene eseguita, dietro richiesta, in presenza di una GUI o momento in cui si inserisce il dischetto nel lettore (se l'eventuale fs presente su di esso non è fra quelli accessibili da Linux) oppure selezionando l'icona del dischetto ed utilizzando un comando da menù (a tendina o flottante a seconda dei casi).

1.5.4 Come accedere ai CD-rom e ad altri dischi fissi

I CD-rom sono dispositivi accessibili in sola lettura per cui su di essi non è possibile eseguire il comando $mkfs$ e non ha molto senso eseguire neanche il comando $fsck$ (entrambi eseguibili, viceversa sui dischetti). Sono ovviamente eseguibili, con tutte le caratteristiche e specifiche viste per i dischetti, i comandi di **mount** e **umount**. Le uniche avvertenze sono:

1. il device di solito si chiama **/dev/hdc** o **/dev/cd0** o nomi simili;
2. il mount point di solito si chiama **/mnt/cd0** o nomi simili.

Il sistema hardware può essere arricchito mediante l'aggiunta di dispositivi di memorizzazione, detti dischi fissi, in modo temporaneo o permanente senza che tali dispositivi siano sempre accessibili al momento del boot (vedi oltre). Tali dispositivi possono, in genere, contenere anche più partizioni (o fs separati) con versioni diverse di SO (ad esempio Windows o MS-DOS). Per l'accesso a tali fs è necessario utilizzare il comando **mount** utilizzando l'opzione **-t tipo_fs**. Ad esempio,il comando:

$$\#\, mount\ -t\, msdos\ /dev/hda2\ /mnt/dos \tag{87}$$

consente di accedere ad un fs MS-DOS tramite il device **/dev/hda2**. La root di tale fs viene collegata alla directory **/mnt/dos** che ne rappresenta

il punto di accesso. Per altri valori del parametro **tipo_fs** si rimanda alla documentazione in linea. Si fa solo notare che per l'accesso ai CD-rom di solito si usa il valore **iso9660**.
Altra opzioni del comando mount sono:

1. $-n$ che collega il fs senza apportare modifiche permanenti alle strutture dati del SO ovvero il file **/etc/fstab**(vedi oltre);

2. $-a$ che collega tutti i fs specificati nel file **/etc/fstab**;

3. $-o$ *opzioni* che permette di accedere ad un fs usando un elenco di opzioni. Alcuni dei valori possibili (se sono più di uno li si deve separare con il carattere ,) sono: *ro* per un accesso in sola lettura (il default per i CD-rom), *rw* per un accesso in lettura e scrittura, $defaults$ per accedere con i valori standard delle varie opzioni e altri per i quali si rimanda alla documentazione in linea.

Per l'accesso a una partizione su un disco fisso aggiuntivo è necessario eseguire le seguenti operazioni:

1. creazione della partizione con i comandi **fdisk** o **cfdisk**;

2. formattazione della partizione con il comando **mkfs**.

Una trattazione dettagliata di tali operazioni esula dall'ambito delle presenti note.
Di solito si vuole che, al momento dell'avvio del SO (ovvero del boot) tutte le partizioni locali e remote a cui si accede abitualmente siano accessibili direttamente senza che sia necessario eseguire, per ciascuna di esse una operazione di **mount**. Si noti che la partizione principale locale (root individuata da /) viene sempre montata al boot. Perchè ciò accada si può configurare il file **/etc/fstab** in modo che contenga le specifiche di tutti i fs da collegare al fs principale locale e si fa in modo che venga eseguito automaticamente il comando **mount -a**. In modo duale, al momento dello spegnimento del SO (o shutdown) si esegue in modo automatico il comando **umount -a** in modo che i vari fs siano scollegati in modo ordinato e non si abbiano problemi al successivo riavvio. Il file **/etc/fstab** ha una struttura abbastanza complessa. A questo livello si fa solo notare che ogni riga contiene, nell'ordine:

1. il nome del device;

2. il nome del mount point;

3. il tipo di fs fra quelli ammissibili;

4. le varie opzioni da usare al momento del mount;
5. se il contenuto del fs deve essere salvato /dump) automaticamente o meno;
6. l'ordine in cui viene verificato il fs durante l'esecuzione del comando **fsck**.

1.5.5 L'utente root (cenni)

Il SO Linux è un SO multiutente. Come abbiamo visto, questa caratteristica si traduce nei concetti di **proprietario**, **gruppo**, **altri** e nei relativi permessi di accesso a file e directory. Non tutti gli utenti sono uguali fra di loro. Il SO riconosce alcuni utenti predefiniti il più importante dei quali è l'utente **root** o super-user. Tale utente è in genere l'unico che ha pieno accesso a tutte le porzioni del fs e che può eseguire operazioni di manutenzione e aggiornamento del fs. In alcuni casi solo root può eseguire lo shutdown del SO e il mount di dispositivi removibili quali CD-rom. In questo caso l'accesso a tali dispositivi è ottenibile mediante script scritti ad hoc che diano ad un utente normale i diritti di super-user solo per l'esecuzione del comando mount. Un'altra caratteristica dell'utente root è quella di non essere limitato dai diritti di acceso per cui, ad esempio, se l'utente root esegue i comandi seguenti:

1. *cd /*
2. *rm**

rischia di cancellare completamente il SO (o almeno la sua quasi totalità). Per tali motivi gli utenti/persone che hanno potenzialmente accesso come root di solito si collegano usando un utente "normale" e acquisiscono i diritti di root (con il comando **su** e la relativa password) solo quando strettamente necessario.
Il comando **su** ha la seguente sintassi:

$$\#\ su\ [altro\_utente] \tag{88}$$

e consente ad un utente di acquisire l'identità di *altro_utente* a patto che ne conosca la password per l'accesso al sistema. A seguito della esecuzione di tale comando, la shell corrente crea una shell che va in esecuzione con i diritti di *altro_utente* fino a che l'utente non la termina, con il comando $CTRL-D$.

1.6 Alcune utility per l'archiviazione

In questa sezione si presentano brevemente le seguenti utility:

1. **tar** ovvero "tape archive", perchè pensato per archiviare dati su nastro magnetico;
2. **gzip** per la compressione;
3. **gunzip** il suo duale.

Il comando **tar** permette di creare archivi contenenti una gerarchia di file e directory a partire da una directory specificata. Il comando permette, inoltre, di aggiornare i file presenti in un archivio, di aggiungerne altri e di ripristinare l'intera gerarchia contenuta nell'archivio o anche solo una sua parte.
Il comando viene usato per eseguire copie di backup di file e directory oppure per raggruppare più file e directory in un unico file per una più agevole trasmissione per posta elettronica (ad esempio).
La sintassi del comando è la seguente:

$$\#\ tar\ [opzioni]\ nome\_archivio.tar\ elenco\_file \tag{89}$$

in cui *nome_archivio.tar* rappresenta l'archivio che si vuole creare e la cui estensione deve essere *tar*, *elenco_file* può contenere file e directory individuati anche mediante metacaratteri. Prima vediamo alcuni esempi poi si passa alla descrizione delle opzioni di uso più comune.

$$\#\ tar\ cf\ javasrc.tar\ *.java \tag{90}$$

crea l'archivio di nome *javasrc.tar* contenente tutti i file di estensione *java* presenti nella directory corrente.

$$\#\ tar\ cf\ project.tar\ progetto \tag{91}$$

crea l'archivio di nome *project.tar* contenente la directory *progetto* con tutti i suoi file e le sue sottodirectory.

$$\#\ tar\ xf\ project.tar \tag{92}$$

permette di estrarre dall'archivio di nome *project.tar* la gerarchia in esso contenuta. L'archivio può essere aggiornato con il comando:

$$\#\ tar\ rf\ project.tar\ new_sources \tag{93}$$

che aggiunge la directory $new_sources$ (ovvero tutta la porzione di fs che ha come sua root locale tale directory) all'archivio *project.tar*. Se si vogliono

aggiornare i file e le directory presenti in un'archivio con quelli presenti nel fs si può usare il comando seguente:

$$\#\ tar\ uf\ project.tar\ new_sources \tag{94}$$

in modo che tutti i file modificati rispetto a quelli contenuti nell'archivio oppure aggiunti alla directory $new_sources$ sono sostituiti nell'archivio oppure aggiunti all'archivio. Il comando:

$$\#\ tar\ tf\ project.tar \tag{95}$$

consente di conoscere il contenuto di un archivio senza che venga eseguita una estrazione dei dati in esso contenuti. Se al posto di un file di estensione *.tar* si usa un device fisico si ha che il comando accede al dispositivo associato a tale device. In tal modo si ha che il comando:

$$\#\ tar\ cf\ /dev/fd0\ *.java \tag{96}$$

esegue l'archiviazione su dischetto e non su un file. Se si vuole eseguire l'archiviazione di una quantità di dati superiore alla capacità del supposto si può usare il comando:

$$\#\ tar\ cfM\ /dev/fd0\ *.java \tag{97}$$

in modo che il comando richieda la sostituzione del dischetto con uno nuovo e ciò fino a che tutti i dati non sono stati archiviati. Si noti che il comando **tar** crea uno **stream di dati** e non un fs per cui non si può eseguire il mount di un dispositivo rimovibile che contiene un archivio.
Il comando che segue:

$$\#\ tar\ xf\ project.tar \tag{98}$$

esegue l'estrazione dei dati contenuti in *project.tar* compreso il ripristino di una eventuale gerarchia di directory.
Le opzioni più comuni del comando **tar** (non precedute dal carattere $-$ e raggruppabili in una stringa) sono, pertanto, le seguenti:

1. c permette di creare un nuovo archivio;
2. t elenca il contenuto di un archivio;
3. r permette di aggiungere elementi ad un archivio;
4. u permette di aggiornare o aggiungere (se non sono già presenti) elementi ad un archivio sulla base della data di ultima modifica;

5. w attende una conferma dall'utente prima di archiviare gli elementi in modo da consentire una archiviazione selettiva;

6. x estrae elementi da un archivio;

7. m non aggiorna la data di modifica agli elementi estratti da un archivio;

8. M permette di creare archivi che sono contenuti su più di un dispositivo rimovibile;

9. $f\,nome\_archivio$ o $f\,device\_fisico$ permette di creare un archivio su un file o su un dispositivo fisico (dischetto o nastro magnetico);

10. z comprime l'archivio con **gzip** o lo scomprime con **gunzip**.

Il comando **tar** di default non esegue nessuna compressione dei file archiviati a meno che non si usi l'operazione z (insieme all'opzione c) che fa in modo che il comando **tar** richiami il comando **gzip**. Se l'opzione z la si specifica insieme all'opzione x il comando richiamato è, viceversa, il comando **gunzip**. L'uso del comando **gzip** su un file di estensione *.tar* produce un file di estensione *.tar.gz*. L'opzione z del comando **tar** comprime i singoli file prima di inserirli nell'archivio e ciò impedisce l'esecuzione di operazioni di aggiunta e aggiornamento in archivi creati con l'opzione z: Il comando **gzip**, in generale, ha la sintassi seguente:

$$\#\ gzip\,[opzioni]\,lista\_nomi \tag{99}$$

Le opzioni più comuni sono:

1. $-r\,directory$ agisce ricorsivamente sulla directory specificata e tutte le sue sottodirectory;

2. $-v$ per ogni file compresso visualizza il nome e la percentuale di compressione ottenuta;

3. $-d$ si comporta come **gunzip** ovvero decomprime gli elementi i cui nomi sono contenuti in $lista\_nomi$.

La $lista\_nomi$ contiene un elenco di elementi (in genere file) da comprimere. I file sono di solito specificati usando metacaratteri.
Il comando **gunzip** accetta in input uno o più file di estensione *.gz* e li scomprime: se file nella directory corrente altrimenti ricreando la directory con tutta l'eventuale gerarchia di sottodirectory.
Se si vuole visualizzare (o stampare) un file compresso senza scomprimerlo si può usare il comando **zcat**.

1.7 I filtri

I **filtri** sono comandi che leggono dati da file, dallo standard input o dall'output di altri comandi, li elaborano eseguendo su di essi alcune operazioni e poi inviano il risultato sullo standard output ovvero sia sullo schermo, sia su un file sia verso un altro comando. I filtri, pertanto, beneficiano degli operatori di redirezione e di concatenazione che abbiamo già esaminato. I filtri hanno in comune la caratteristica di non modificare la sorgente dei dati originaria in modo permanente per cui se si usa come sorgente di dati per un filtro un file il suo contenuto non viene in alcun modo alterato dall'azione del filtro.
In genere i filtri sono classificabili come:

1. filtri per file;
2. filtri per l'editing;
3. filtri per i dati.

In quanto segue si suppone di avere un file di testo che chiameremo $file\_dati$ e che l'input dei filtri proviene dallo standard output del comando:

$$\#\, cat\ file\_dati \tag{100}$$

connesso allo standard input di un filtro mediante una pipe |. I filtri accettano i dati anche sul loro standard input in modo che i comandi:

$$\#\, filtro\ file\_dati \tag{101}$$

e:

$$\#\, cat\ file\_dati \mid\ filtro \tag{102}$$

sono equivalenti. I primi due filtri che esaminiamo sono **head** e **tail**.Le sintassi sono le seguenti:

$$\#\, head\,[opzione]\ file\_dati \tag{103}$$

$$\#\, tail\,[opzione]\ file\_dati \tag{104}$$

Il primo visualizza le righe iniziali di un $file\_dati$ mentre il secondo le righe finali. Per il primo l'opzione usata di solito è num in modo da visualizzare il numero specificato di righe. Il valore di default è 10, lo stesso che per il secondo comando. Per il secondo comando le opzioni pìcomuni sono:

1. $-num$ per visualizzare num righe del file;

2. $+num$ per visualizzare il file a partire dalla riga specificata;

3. $-r$ per visualizzare le righe del file in ordine inverso.

È ovvio che se il numero specificato è superiore al numero di righe contenute nel file questi viene visualizzato per intero.
Mentre i filtri **head** e **tail** si limitano a estrarre una porzione dei dati in ingresso altri filtri compiono su tali dati elaborazioni più o meno complesse. Nel seguito esamineremo brevemente i seguenti filtri:

1. *wc* per il conteggio;

2. *spell* per il controllo ortografico;

3. *sort* per l'ordinamento.

Va da sè che i vari filtri che descriveremo, dato che accettano dati sullo standard input e producono dati sullo standard output, possono essere collegati per mezzo dell'operatore | (con o senza l'uso di redirezioni) in una varietà di combinazioni che non esamineremo se non in minima parte. Ad esempio, il comando:

$$\#\ head\ file\_dati\ \mid\ wc \tag{105}$$

esegue un conteggio (vedi oltre) sulle prime 10 righe del file $file\_dati$.
Il filtro **wc** accetta in ingresso un flusso di dati e conta il numero di righe, parole e caratteri (compresi i caratteri di fine riga) contenuti in tale flusso. La sintassi è la seguente:

$$\#\ wc\ [-w \mid -l \mid -c]\ file\_dati \tag{106}$$

In mancanza di una opzione il comando restituisce, nell'ordine:

1. il numero di righe,

2. il numero di parole,

3. il numero di caratteri,

contenuti o in $file\_dati$ o sullo standard input. Le opzioni hanno i seguenti significati:

1. $-w$ per contare solo le parole;

2. $-l$ per contare solo le righe;

3. $-c$ per contare solo i caratteri.

Il comando **spell** permette di verificare la correttezza ortografica delle parole contenute in un file $file\_dati$ o sullo standard input utilizzando, come criterio di correttezza, le parole contenute in un file dizionario il cui nome è passato come parametro. La sintassi del comando è la seguente:

$$\#\ spell\ [+dizionario]\ file\_dati \tag{107}$$

Il filtro **sort** consente di ordinare lessicograficamente le righe di un file. La sintassi del comando è la seguente:

$$\#\ sort\ [-r \mid -n]\ file\_dati \tag{108}$$

mentre come filtro accetta dati prodotti da altri comandi collegati ad esso con una pipe. L'opzione $-r$ fa si che l'ordinamento avvenga in senso inverso (dalla z alla a) mentre l'opzione $-n$ fa si che i dati contenuti nel file siano considerati come valori numerici e non come stringhe contenenti i caratteri da 0 a 9. Il comando/filtro ha molte altre opzioni e consente di eseguire l'ordinamento dei dati di $file\_dati$ sul contenuto di un ben preciso campo di ogni riga, previa definizione di un carattere separatore di campo. Per tali opzioni si rimanda alla documentazione in linea.
Un'altra famiglia di filtri molto potenti è quella che contiene i filtri **grep** e **fgrep** che permettono di ricercare una stringa in uno o più file (o sullo standard input, caratteristica comune a tutti i filtri e che sarà omessa, in quanti sottintesa, da ora in poi) e producono sullo standard output le righe che contengono tale stringa. Se la ricerca è in più file ogni riga è preceduta dal nome del file che la contiene. La differenza fra i due filtri è che il primo consente di ricercare una sola stringa mentre il secondo può ricercare contemporaneamente più stringhe. Il filtro **grep** permette di usare anche **espressioni regolari** di cui si parlerà a breve.
Il filtro **grep** accetta due argomenti ovvero:

1. la stringa che si vuole ricercare;
2. i nomi dei file in cui la si vuole cercare.

La sintassi del comando è la seguente:

$$\#\ grep\ [opzioni]\ stringa\ lista\_nomi \tag{109}$$

Le opzioni di uso più comune sono le seguenti:

1. $-i$ fa in modo che il filtro ignori la differenza fra lettere maiuscole e minuscole;

2. $-c$ fa in modo che il filtro visualizzi solo il numero totale di righe contenenti la stringa cercata;

3. $-l$ fa in modo che il filtro visualizzi solo i nomi dei file che contengono la stringa cercata;

4. $-n$ fa in modo che il filtro visualizzi il numero di riga e il testo delle righe che contengono la stringa cercata;

5. $-v$ fa in modo che il filtro visualizzi solo le righe che non contengono la stringa cercata.

La *lista_nomi* può contenere più nomi di file, individuati anche facendo uso di metacaratteri. Qualora la stringa che si ricerca contenga spazi la si deve racchiudere fra una coppia di apici singoli.
Il filtro **fgrep** può ricercare nei file una o più parole contemporaneamente ma non può eseguire ricerche facendo uso di espressioni regolari. La sintassi del comando è la seguente:

$$\#\ fgrep\ [opzioni]\ stringhe\ lista\_nomi \tag{110}$$

L'opzione più significativa è $-f\ nome\_file$ che fa in modo che il filtro trovi le stringhe da ricercare o in *lista_nomi* o sullo standard input in $nome\_file$. In ogni caso le stringhe da ricercare devono essere separate dal carattere di fine linea. Se le stringhe da ricercare sono specificate sulla linea di comando devono essere separate dal carattere di fine linea preceduto dal carattere di escape \.
Altri filtri disponibili in ambiente Unix e Linux sono detti **filtri di editing**: tali filtri eseguono operazioni di modifica dei dati che trovano o in file o ricevono tramite lo standard input e producono in output una versione modificata di tali dati.
I filtri che appartengono a tale famiglia sono i seguenti:

1. *tr* che esegue traduzioni di caratteri;

2. $diff$ che produce informazioni derivanti dal confronto fra due file;

3. *sed* che esegue operazioni di editing sulle righe di testo presenti sul suo standard input.

Il filtro **sed** è, come dice il nome **s**tream **ed**itor, un editor di flussi di dati che riceve sullo standard input, da file o da altri comandi. Il filtro accetta come argomenti una operazione di editing e un elenco (opzionale) di nomi di file e produce sullo standard output una versione modificata dei dati originali che,

pertanto, non sono modificati. I dati prodotti in output sono una versione modificata secondo certe regole di tutti i dati in input per cui si parla di "copia completa" (con modifiche) dei dati di input al filtro.
La sintassi del comando è la seguente:

$$\#\ sed\ 'comando\_di\_editing'\ lista\_nomi \tag{111}$$

L'insieme dei comandi di editing utilizzabili (che vanno racchiusi fra una coppia di apici singoli in modo da impedire che la shell interpreti gli eventuali caratteri speciali presenti) è molto vasto e consente l'esecuzione di operazioni di editing in batch molto potenti. In quanto segue ci limiteremo a dare solo alcuni esempi di utilizzo:

1. il comando **'3 d'** permette di cancellare la terza riga di un file;
2. il comando **'2 s/vecchia_stringa/nuova_stringa** permette di sostituire la prima occorrenza della prima stringa con la seconda nella riga indicata;
3. il comando **'2 s/vecchia_stringa/nuova_stringa/g** permette di sostituire tutte le occorrenze della prima stringa con la seconda nella riga indicata.

Altri comandi utilizzabili sono:

1. **a** per aggiungere testo dopo la riga specificata;
2. **c** per sostituire testo nelle righe specificate;
3. **i** per aggiungere testo prima della riga specificata;
4. **n** per associare un numero alle righe di output.

Il comando accetta comandi di editing multipli come parametri, ciascuno preceduto dall'opzione **-e**, oppure contenuti in un file il cui nome è passato come parametro mediante l'opzione **-f nome_file_comandi**. Il comando **sed** riceve un flusso di dati da elaborare sullo standard input per cui li può leggere da un file o ricevere da un altro comando tramite una pipe ed emette i dati elaborati sullo standard output per cui tali dati possono essere rediretti su un file oppure passati ad un altro comando tramite una pipe.
Il filtro **diff** permette di confrontare due file secondo la sintassi seguente:

$$\#\ diff\ file1\ file2 \tag{112}$$

Il comando produce sullo standard output le righe diverse dei due file e mostra come il primo debba essere modificato per essere uguale al secondo: il

comando confronta i due file riga per riga (e carattere per carattere per ciascuna coppia di righe) e produce sullo standard output solo le differenze fra coppie di righe. Le righe del primo file sono prefissate dal simbolo < quelle del secondo dal simbolo >. Il comando prevede l'uso di direttive di editing per la modifica di uno dei due file in modo da renderlo uguale all'altro, direttive su cui non ci soffermiamo oltre. L'output del comando può essere rediretto sia su un file sia verso un altro comando usando le tecniche note.
Molte utility e filtri individuano e selezionano porzioni di testo contenute in uno o più file (o anche sullo standard input) usando stringhe che contengono caratteri speciali e, pertanto, dette **espressioni regolari**. I caratteri speciali sono simili ai metacaratteri della shell, sono usati per eseguire ricerche all'interno di file e sono usati da programmi quali: ed, sed, awk, grep ed egrep. Usando le espressioni si possono ricercare varianti di una stringa in determinati punti del testo (ad esempio a inizio oppure a fine riga) in modo da definire criteri di ricerca flessibili e potenti.
I caratteri speciali utilizzabili sono i seguenti:

1. ^ che permette di riferire l'inizio delle righe di un file secondo la sintassi ^*stringa* che fa si che si ricerchino le righe che iniziano per *stringa*;

2. $ che permette di riferire la fine delle righe di un file secondo la sintassi *stringa*$ che fa si che si ricerchino le righe che terminano per *stringa*;

3. $*$ che si riferisce a 0 o un numero qualunque di occorrenze di un carattere per cui $ac*$ individua il carattere a seguito da un numero qualunque di c (anche nessuna);

4. . che individua un carattere qualunque in modo che se si ha $a.a$ si individuano stringhe come ala, ada (ma anche $adagio$) e simili ovvero tutte quelle che si possono formare sostituendo al . un carattere qualunque;

5. [] che individuano una classe di caratteri. Ad esempio se si ha $[agN]$ si individuano i tre caratteri specificati ovvero si richiede che in quella posizione di una stringa compaia uno di tali caratteri. Se si ha $doc[123]$ si individuano le stringhe $doc1$, $doc2$ e $doc3$.

Le [] permettono di specificare al loro interno altri elementi sintattici che permettono di raffinare la descrizione dell'insieme di caratteri da ricercare. Si hanno i casi seguenti:

1. ^ che permette di escludere dalla ricerca i caratteri specificati secondo la sintassi [^*caratteri*];

2. $-$ che permette di individuare sottoinsiemi contigui dell'insieme dei caratteri come in $[a-zA-Z]$ che seleziona tutti i caratteri alfabetici, minuscoli e maiuscoli;

3. $*$ che permette di riferire 0 occorrenze o un numero qualunque di occorrenze dei caratteri racchiusi fra [].

Come esempio di uso delle espressioni regolari esamineremo brevemente il filtro **grep**.
Il comando **grep** usa le espressioni regolari per selezionare dati all'interno di un file in modo da selezionare le righe che contengono una certa stringa in una certa posizione su ogni riga. Anche in questo caso ci limiteremo ad alcuni esempi.

1. Il comando:
$$\#\, ls\ -l\ \mid\ grep\,\hat{}\, d \tag{113}$$
permette di elencare solo i dati relativi alle directory contenute nella directory corrente.

2. Il comando:
$$\#\, ls\ -l\ \mid\ grep\,\hat{}\, l \tag{114}$$
permette di elencare solo i dati relativi ai link simbolici contenuti nella directory corrente.

Si può usare il carattere \$ per eseguire la ricerca di dati alla fine di ogni riga oppure i caratteri $*$ per occorrenze multiple di caratteri, . per indicare un carattere qualunque e [] per individuare insiemi di caratteri. Il comando:
$$\#\, ls\ -l\ \mid\ grep\,\hat{}\ -..x \tag{115}$$
permette di elencare i file accessibili in esecuzione dall'utente e contenuti nella directory corrente mentre il comando:
$$\#\, ls\ -l\ \mid\ grep\,\hat{}\, d..x \tag{116}$$
si riferisce alle sole directory. Una opzione molto utile del comando **grep** è la **-v** che consente di emettere sullo standard output del comando tutte le righe presenti sul suo standard input e che non contengono una certa stringa. Usando tale opzione si può definire un comando come:
$$\#\, ls\ -l\ \mid\ grep\ -v\,\hat{}\, d \tag{117}$$
in modo da elencare tutti gli elementi della directory corrente tranne le sue sottodirectory.

1.8 Complementi e cenni a concetti avanzati: la personalizzazione della shell e gli alias

Tutto quanto abbiamo visto finora ha pienamente valore su l'utente accede al SO tramite la shell BASH (Bourne Again SHell, dal nome di una delle prime shell di Unix, la Bourne Shell). Linux mette a disposizione degli utenti altre shell, quali la PDKSH (Public Domain Korn SHell), la C-shell e la TCSH. Nel seguito ci riferiremo solo alla BASH, rimandando alla documentazione in linea e ad i molti manuali sia di Linux sia di Unix per una descrizione delle altre shell.
La BASH ha una linea di comando (comprendente tutti i caratteri che l'utente digita prima del tasto di fine linea (Invio o Enter) con le seguenti funzionalità (che ci limitiamo solo ad enunciare):

1. completamento di comandi e di nomi di file e directory;
2. editing della linea di comando, richiamo dei comandi precedenti.

La seconda delle due funzionalità si appoggia sul concetto di **history**. La **history** contiene una registrazione di un numero configurabile di comandi precedentemente usati dall'utente, ciascuno preceduto da un numero e detto evento. Gli eventi sono numerati in base alla successione di esecuzione e quelli con i numeri più elevati sono quelli eseguiti più di recente. I comandi (anche quelli errati o non più validi perchè riferiti ad oggetti che non esistono più) presenti nella history possono essere richiamati sulla linea di comando (editati e/o eseguiti di nuovo) in vari modi:

1. mediante i tasti **freccia in sù** o **freccia in giù**;
2. mediante un riferimento assoluto ovvero il numero d'ordine preceduto dal carattere !;
3. mediante un riferimento relativo ovvero specificando la distanza dell'evento dalla fine dell'elenco con la sintassi $!-n$ in cui n individua lo spostamento di posizioni lungo la lista degli eventi;
4. con un richiamo parziale, usando la porzione iniziale dell'evento voluto preceduta dal carattere !.

La history può essere configurata:

1. in lunghezza, ovvero nel numero degli eventi passati di cui conserva traccia, mediante la variabile di ambiente $HISTSIZE$ che, di default, assume il valore 500;

2. nel nome del file di history, il nome di default è *.bash_history*, assegnando il nuovo nome alla variabile di ambiente $HISTFILE$.

Si ricorda che l'assegnazione di un valore ad una variabile di ambiente ha la sintassi seguente:

$$\#\, VAR = valore \tag{118}$$

che deve essere seguito da una operazione di **export** in modo che il valore sia noto ai processi che l'utente crea all'interno dell shell e che l'accesso al valore di una variabile avviene usando il riferimento $\$VAR$.
Una utile funzionalità della BASH è rappresentata dagli **alias** che consentono la definizione di sinonimi di comandi complessi o di uso corrente.
La sintassi è la seguente:

$$\#\, alias\, mnemonico =' command\_list' \tag{119}$$

Gli apici singoli sono indispensabili se:

1. la *command_list* contiene un comando con opzioni;

2. la *command_list* contiene più comandi separati dal carattere ; con o senza opzioni;

Si danno, senza pretese di completezza, alcuni esempi di uso del costrutto alias. Si deve tenere presente che la shell permette agli utenti la scrittura di file di comandi contenenti strutture di controllo e parametri, detti script, che la shell interpreta ma che, per l'utente, sono equivalenti a comandi personalizzati.
Gli esempi sono elencati e commentati solo se necessario:

1. $alias\, lss =' ls\ -s'$

2. $alias\, lsa =' ls\ -aF'$

3. $alias\, rm =' rm\ -i'$ in questo modo si crea una versione del comando rm che prevede che l'utente debba rispondere y ad ogni potenziale rimozione di un file;

4. $alias\, mv =' mv\ -i'$ in questo modo si crea una versione del comando mv che prevede che l'utente debba rispondere y ad ogni potenziale spostamento di un file;

5. $alias\, cp =' cp\ -i'$ in questo modo si crea una versione del comando cp che prevede che l'utente debba rispondere y ad ogni potenziale copia di un file su un altro esistente;

6. $alias\ lsc =' ls\ *.[co]'$ in cui si usano i metacaratteri.

Si fa notare come il carattere = non deve essere nè preceduto nè seguito da caratteri di spaziatura.
Il comando **alias** senza parametri consente ad ogni utente di conoscere la lista di tutti gli alias attivi ad un certo istante. Il comando **unalias** con la sintassi:

$$\#\ unalias\ mnemonico \tag{120}$$

permette di eliminare uno degli alias fra quelli attivi.
La BASH usa, per la sua configurazione, un file *.bash_profile* presente nella home directory di ogni utente e che viene eseguito automaticamente ad ogni login dell'utente. Oltre a tale file la shell è caratterizzata da un altro file di configurazione *.bashrc* presente nella home directory di ogni utente e che viene eseguito automaticamente ogni volta che l'utente accede alla shell o genera una sotto-shell. All'interno del file *.bash_profile* si può inserire una chiamata del tipo:

$$.\sim /.bashrc \tag{121}$$

in modo che la shell di login esegua automaticamente i comandi contenuti in *.bashrc*.
Al momento del logout, infine, vengono eseguiti i comandi specificati nel file *.bash_logout*, se presente nella home directory dell'utente. Lo scopo è quello di eseguire alcune operazioni di manutenzione dell'accounting, compresa la pulizia della cache del browser e di eventuali directory che contengono file temporanei che non si desidera conservare da una sessione di lavoro alla successiva.
Il file *.bash_profile* consente il settaggio di variabili di ambiente e di utente i cui valori sono resi accessibili a tutti i processi di utente mediante operazioni dei **export**. Le variabili che di solito si inizializzano in tale file sono:

1. $PATH$ con le directory in cui la shell ricerca i comandi;
2. $HOME$ con il pathname assoluto della directory di lavoro al login (e quella che vene resa directory corrente dalla esecuzione del comando **cd** senza parametri);
3. $HISTSIZE$ in modo da settare una dimensione della history;
4. $PS1$ in modo da settare il propmpt di prima linea;

e così per altre variabili per le quali si rimanda alla documentazione in linea della BASH.

Per aggiungere nuove directory alla variabile $PATH$ predefinita si procede in questo modo:

$$PATH = \$PATH : \$HOME/bin \tag{122}$$

in modo da aggiungere la directory $HOME/bin$ all'elenco delle directory al cui interno la shell ricerca i comandi usati dall'utente.
Il file *.bashrc* contiene comandi che sono eseguiti, al momento del login, subito dopo l'esecuzione dei comandi contenuti nel file $.bash\_profile$ oppure ad ogni accesso alla shell. Tale file in genere si apre con un richiamo alla versione generale del file (ovvero */etc/bashrc*, comune a tutti gli utenti) e, in genere, contiene le definizioni degli alias di ciascun utente (quelli comuni sono, di solito, definiti in */etc/bashrc*) e di altri elementi validi per la singola shell.

1.9 Introduzione ad Emacs

Emacs è un editor che consente agevolmente la creazione di file di testo mediante immissione diretta di dati da tastiera. I comandi eseguibili sui dati sono implementati mediante i tasti $CTRL$ e ALT. L'editore è complesso e sofisticato e consente la gestione di più finestre per l'editing in contemporanea di più file, uno per ogni finestra. Nelle presenti note si darà solo una descrizione per sommi capi del programma rimandando, come di consueto, alla documentazione in linea per ulteriori dettagli.
I file su cui l'utente lavora sono aperti ciascuno in un buffer di memoria in modo che i comandi di editing siano eseguiti solo su tale copia del file fino a che l'utente non decide di salvare le modifiche in un file (che può essere anche distinto da quello aperto originariamente).

1.9.1 Accesso ai file

L'editore viene mandato in esecuzione (in genere in modalità background) con il comando seguente:

$$\#\ emacs\ [nome\_file]\ \& \tag{123}$$

In presenza del parametro $nome\_file$ si ha che l'editore apre il file corrispondente se esiste o, altrimenti, lo crea vuoto. In assenza di tale parametro l'editore apre una finestra, cui è associato un buffer, entro la quale l'utente immette del testo che poi deve salvare in un file a cui deve attribuire:

1. una posizione all'interno del fs, di solito in una directory contenuta nella gerarchia che si diparte dalla sua home directory;

2. un nome e una estensione.

In questo caso l'editor consente all'utente di aprire anche uno dei file esistenti nel fs (comando **open**).
In ogni caso il cursore (che segnala la posizione corrente in cui viene inserito il testo che l'utente immette da tastiera) è posizionato nell'angolo in alto a sinistra della finestra che presenta, in basso due righe speciali, una sopra l'altra.
La riga inferiore permette l'immissione di comandi da parte dell'utente e la visualizzazione dei messaggi prodotti dall'editor.
La riga superiore è detta riga di stato che viene usata per visualizzare informazioni relative al testo che si stà elaborando.
Ogni finestra è quindi caratterizzata (dall'alto verso il basso):

1. da un'area di editing in cui l'utente immette i dati;
2. una riga di stato;
3. una riga di comando.

L'editor si trova automaticamente in modalità input (o immissione) mentre i comandi fanno uso di combinazioni di tasti con il tasto $CTRL$ o control. Ad esempio:

1. $CTRL-x$ e $CTRL-s$ permettono di salvare il contenuto di un buffer in un file;
2. $CTRL-x$ e $CTRL-c$ permettono di terminare l'esecuzione dell'editor;
3. $CTRL-f$ e $CTRL-b$ permettono di spostare il cursore in avanti e indietro sulla stessa riga di testo;
4. $CTRL-p$ e $CTRL-n$ permettono di spostare il cursore sulla riga precedente o sulla successiva nel file che si stà editando.

1.9.2 I comandi

I comandi sono associati:

1. alla combinazione tasto $CTRL$ + tasto alfanumerico;
2. ai meta−tasti implementati o dalla successione ESC + tasto alfanumerico o dalla combinazione ALT + tasto alfanumerico.

Nel seguito faremo riferimento a comandi per indicare eventi del primo tipo e a meta−tasti per indicare, indifferentemente, eventi degli altri due tipi. È possibile immettere comandi, usando parole chiave ad hoc, tramite la riga di comando cui si accede tramite i meta−tasti $ESC - x$ o $ALT - x$: in tal modo il cursore si posiziona su tale riga e permette all'utente di immettere uno dei comandi riconosciuti da emacs.
La riga di stato visualizza informazioni relative al testo in corso di modifica: se è stata eseguita o meno una operazione di salvataggio in un file, il nome del buffer (ovvero il nome del file in corso di editing), la posizione del cursore all'interno del file e la modalità di immissione testo che emacs può variare adeguandola alla estensione del file che si stà editando, caratteristica utile nel caso dei linguaggi di programmazione che emacs conosce in quanto consente l'indentazione appropriata delle linee dei programmi e l'uso di colori per evidenziare in modo coerente i vari elementi sintattici.

1.9.3 I comandi di editing

I comandi che l'editor mette a disposizione degli utenti sono raggruppabili nelle seguenti categorie.

1. Comandi di spostamento del cursore: oltre a quelli già visti si hanno $CTRL - v$ e $CTRL - z$ permettono di spostare il cursore in avanti e indietro di una pagina. Altri comandi fanno uso del tasto ESC in modo che:

   (a) $ESC\ f$ sposti il cursore avanti di una parola;

   (b) $ESC\ b$ sposti il cursore indietro di una parola;

   (c) ESC [sposti il cursore al paragrafo precedente;

   (d) ESC] sposti il cursore al paragrafo successivo;

   Altri fanno uso del tasto $CTRL$ in modo che:

   (a) $CTRL + a$ sposti il cursore all'inizio di una riga;

   (b) $CTRL + e$ sposti il cursore alla fine di una riga;

   (c) $CTRL + l$ sposti il cursore a metà riga;

   (d) ESC] sposti il cursore al paragrafo successivo.

2. Comandi di cancellazione del testo:

   (a) $CTRL + d$ cancella il carattere a destra della posizione corrente del cursore;

   (b) $CANC$ cancella il carattere a sinistra della posizione corrente del cursore.

3. Comandi per l'uso dei buffer: questi comandi consentono la copia in un buffer di dati selezionati da un file, con o senza rimozione. In tal modo i dati contenuti nel buffer possono essere reinseriti nel file in una posizione diversa. La copia con o senza cancellazione può riguardare varie porzioni del testo. Alcuni comandi di copia e cancellazione sono:

   (a) $ESC\ CANC$ che agisce sulla parola che precede la posizione corrente del cursore;

   (b) $ESC\ d$ che agisce sulla parola che segue la posizione corrente del cursore;

   (c) $ESC\ 3\ ESC\ d$ che agisce sulle tre parole che seguono la posizione corrente del cursore;

   (d) $CTRL\ k$ cancella la riga a partire dalla posizione corrente oppure dall'inizio riga se prima si usa il comando $CTRL\ a$ tranne il comando di fine riga che va rimosso con un ulteriore $CTRL\ k$;

   (e) $ESC\,3CTRL\,k$ cancella tre righe a partire dalla posizione corrente oppure dall'inizio della prima delle tre righe se prima si usa il comando $CTRL\ a$ tranne il comando di fine ultima riga che va rimosso con un ulteriore $CTRL\ k$;

   (f) $CTRL\ y$ inserisce il contenuto del buffer alla posizione corrente del cursore;

   (g) $CTRL\ x\ u$ annulla il comando precedente;

   (h) $ESCx$ annulla tutte le modifiche apportate nella sessione corrente.

4. Comandi per l'esecuzione di ricerche: per l'esecuzione di una ricerca in un file si può usare il comando $CTRL\ s$ a seguito del quale il cursore si posiziona sull'area di comando in modo che l'utente possa immettere la stringa da ricercare. La ricerca è di tipo incrementale nel senso che l'editore si sposta nel testo in funzione dei caratteri immessi dall'utente. La stringa da ricercare è chiusa dalla pressione del tasto ESC. Il comando $CTRL\ s$ permette di eseguire ricerche dalla posizione del cursore in avanti. Se si vogliono eseguire ricerche dalla posizione del cursore verso l'inizio del file si usa il comando $CTRL\ r$.

Oltre ai comandi visti, l'editor consente:

1. di usare espressioni regolari e metacaratteri per le ricerche;

2. eseguire sostituzioni globali e condizionali;
3. di usare più finestre per visualizzare più porzioni dello stesso file o più file contemporaneamente, per ognuno dei quali viene creato un buffer opportuno.

Per ulteriori dettagli si rimanda alla documentazione in linea del programma.

2 Introduzione all'uso di Java

2.1 Note introduttive

In questa sezione si esamina come sia possibile utilizzare il linguaggio *Java* per la stesura di piccoli programmi, facendo riferimento al testo [CM04]. La conoscenza di base del linguaggio è data per scontata. Un utile riferimento sono sia i numerosi tutorial liberamente scaricabili da internet sia, per i più tradizionalisti, il testo [HC99] ed edizioni successive. Nella presente sezione ci limitaeremo ad esaminare con un certo dettaglio i passi seguenti:

1. creazione di file di tipo *.java* con un editore, nel caso presente il programma *emacs* (vedi la sezione 1.9);
2. la loro compilazione con il comando *javac* e la loro esecuzione con il comando *java*.

2.2 Il ciclo di base

Il ciclo di base secondo il quale si scrivono programmi in un qualunque linguaggio di programmazione è il seguente:

1. editing;
2. compilazione e correzione statica;
3. debugging e correzione dinamica;
4. esecuzione.

Tranne che nei casi più semplici la scrittura di un programma richiede, infatti, la correzione degli errori sintattici dei tipi più variegati che vengono rilevati in fase di compilazione e la correzione degli errori semantici che vengono rilevati facendo eseguire il programma compilato correttamente ma non corretto da un punto di vista logico e/o fattuale.
La fase di correzione statica di solito si traduce in una successione di fasi di compilazione e di editing e termina quando la compilazione viene eseguita senza che il compilatore rilevi alcun errore.
La fase di correzione dinamica prevede che il programma venga testato su insiemi di dati ad hoc per i quali produce le risposte attese. È una fase piò complessa e delicata sulla quale non ci soffermeremo oltre.
Abbiamo visto che la shell permette la stesura di programmi (o script) che sono **interpretati** o dalla shell corrente o da una shell ad hoc. Un linguaggio come il linguaggio *C* prevede una fase di compilazione (e di link di codice già

scritto e contenuto nelle cosiddette librerie) che produce il cosiddetto codice oggetto eseguibile sul sistema su cui è avvenuta la compilazione (o su un sistema in tutto e per tutto compatibile).
Il linguaggio Java prevede, invece, una compilazione dei file sorgente (i file di tipo **.java**) utilizzando il comando **javac** che produce file di identico nome ma di estensione **.class**. I file **.class** contengono il cosiddetto **bytecode** che non corrisponde al linguaggio macchina di un particolare processore ma è eseguibile da un ambiente astratto che ne cura l'interpretazione ed è detto **Java Virtual Machine** o **JVM**. Un programma scritto in Java (e che non fa uso di classi molto particolari) può, pertanto, essere eseguito su ogni sistema su cui sia presente una JVM. Il linguaggio Java, quindi, ha alcune caratteristiche dei linguaggi come il C, in quanto richiede una fase di compilazione, e delle shell dal momento che il prodotto della compilazione viene interpretato da un ambiente astratto come gli script sono interpretati, riga per riga, da una shell che ne cura l'esecuzione.

2.3 I vari tipi di file e altro

Le istruzioni scritte in linguaggio Java (o codice sorgente) devono essere contenute in file la cui estensione sia **.java**. In più:

1. un file **A.java** deve contenere la classe **A** dichiarata come **public** ovvero **accessibile**;

2. la classe **A** deve contenere il metodo (analogo ad una procedura) **main** dichiarato **static** ovvero applicabile senza dover creare oggetti per poterlo usare.

Il classico esempio con cui si inizia è un programma che si limita a visualizzare un messaggio sullo schermo ([CM04]):

```
public class CiaoATutti {
  public static void main (String[] args) {
    System.out.println(''Ciao a tutti!!!'');
  }
}
```

Il codice suddetto è contenuto nel file **CiaoATutti.java**. Si fa notare che:

1. il metodo **main** è dichiarato **public,** ovvero accessibile, e **void**, ovvero non restituisce nessun valore all'ambiente chiamante;

2. il metodo **main** accetta un certo numero di parametri in ingresso rappresentati dal parametro formale **String[] args**, ovvero un array di stringhe di nome collettivo **args**;

3. la visualizzazione avviene utilizzando il metodo **println** che accetta in input la stringa da visualizzare (in questo caso la stringa **Ciao a tutti!!!** e che appartiene alla classe **out** contenuta nella classe **System**. Il carattere . definisce una gerarchia in modo analogo alla gerarchia definita dal carattere / nel fs.

L'esecuzione del comando:

$$\#\ javac\, CiaoATutti.java \tag{124}$$

determina la compilazione del codice contenuto nel file **CiaoATutti.java** e la creazione del file **CiaoATutti.class** contenente il bytecode corrispondente. Il file **CiaoATutti.class** viene creato nella stessa directory che contiene il file sorgente **CiaoATutti.java**. L'esecuzione del comando:

$$\#\ java\, CiaoATutti \tag{125}$$

fa in modo che venga eseguito il metodo **main** presente nella classe **CiaoATutti** che causa la visualizzazione sullo schermo del messaggio **Ciao a tutti!!!**.
Riassumendo, nei casi semplici si ha la seguente successione di passi:

1. si crea con un editor un file **A.java** che contiene la classe **A** che, a sua volta, contiene il metodo **main**;

2. si compila il file **A.java** con il comando **javac**;

3. se la compilazione rileva errori li si deve correggere e ripetere la compilazione ripetendo il ciclo editing÷compilazione fino a che la compilazione non viene eseguita senza errori;

4. se la compilazione non rileva errori si può far eseguire il programma dalla JVM usando il comando **java**.

2.4 Qualcosa su oggetti e classi

Una **classe** rappresenta un modello o uno schema o uno stampo per la creazione di istanze dette **oggetti**. Il primo passo consiste nella definizione di una classe in modo che sia possibile la creazione di oggetti della classe. Il linguaggio è caratterizzato da una ampia collezione di classi predefinite

delle quali è possibile creare istanze a meno che non siano definite in modo particolare (vedi oltre).
Le classi sono caratterizzate da:

1. metodi pubblici, ovvero accessibili dall'esterno della classe, e privati, ovvero usati all'interno della classe stessa:

2. variabili e strutture dati in genere private della classe;

3. uno o più metodi particolari, detti **costruttori**, usati per la creazione di oggetti, istanze della classe.

Alcune classi possono essere usate come base per la definizione di classi dette derivate. Ad esempio si può definire una classe *GeometricShape*, che definisce tutte le figure geometriche su un piano, e usare tale classe per definire le classi dei *Polygons*, *Cyrcles* e così via: la prima è detta essere la **superclasse** mentre le altre sono le classi derivate. Tele meccanismo è detto **ereditarietà**.
Altre classi non possono essere estese dato che sono dichiarate **final**. Un esempio di classe di tale tipo è la classe **String**.
Se una classe è caratterizzata dal fatto che i suoi metodi sono dichiarati ma non implementati, ovvero ad essi non è associato alcun codice che li renda effettivamente utilizzabili, la si dice una **interfaccia**. Se, invece, tale caratteristica vale non per tutti i metodi ma per alcuni che sono definiti come **abstract** la classe si dice astratta, è caratterizzata dalla parola chiave **abstract** e deve essere estesa da una classe che Complementi tutti i metodi lasciati non specificati nella definizione della classe.
Se una classe è definita, infine, come **static** i suoi metodi possono essere usati senza che sia necessario definire oggetti della classe ma direttamente come membri della classe. Un tipico esempio di classe **static** è la classe **Math**, tipica del linguaggio. Se si ha una classe "normale" **A** con un metodo **eval()**, per usare il metodo è necessario creare un oggetto della classe a cui applicare il metodo, ovvero:

1. $A\, a = new\ A(i);$

2. $int\, t = \ a.eval();$

supponendo che il metodo **eval()** restituisca un valore di tipo *int*. Nel caso della classe **Math** non è necessario (nè corretto) definire un oggetto ma si può accedere direttamente ai suoi metodi. Ad esempio si ha:

$$double\, d = Math.sqrt(2.0); \tag{126}$$

Una classe è caratterizzata dai suoi **metodi** alcuni dei quali sono i costruttori della classe. Java non conosce il concetto di distruttore in modo che lo spazio di memoria allocato ad un oggetto viene rilasciato solo al momento in cui l'oggetto non è più utilizzato all'interno del codice.
I costruttori hanno lo stesso nome della classe per cui, ad esempio, la classe A ha il costruttore A. Se una classe non ha un costruttore esplicito, l'ambiente di esecuzione ne definisce uno di default. Ad esempio:

$$A\, a = new\, A(i); \tag{127}$$

definisce l'oggetto a come istanza della classe A. Il costruttore $A()$ ha, in questo caso, un parametro come input che fornisce un valore assegnato ad una delle variabili dell'oggetto istanza della classe.
La sezione 2.6 contiene un esempio di definizione di una classe per la gestione dell'input da tastiera.
I **metodi** sono le operazioni che gli oggetti istanze di una classe sono in grado di eseguire oppure i costruttori della classe, ovvero gli strumenti per la creazione di oggetti. Li si può classificare anche come:

1. metodi per l'accesso alle variabili interne della classe che permettono di conoscerne il valore corrente (o accessori);

2. metodi per la modifica del valore delle variabili interne (o modificatori).

Qui di seguito si riporta un esempio di classe tratto da [HC99]. La classe **EmployeeTest** (contenuta nel file **EmployeeTest.java**) fa uso di classi contenute nei **package** (vedi la sezione 2.5), su cui si eseguono le operazioni di **import**, e di una classe interna **Employee**. La prima serve come strumento per usare la seconda che ha un costruttore e un certo numero di metodi **public**. È immediato vedere come le variabili siano tutte "nascoste" (dato che sono dichiarate **private**) e accedute tramite metodi ad hoc.

```
/**
 * @version 1.00 07 Feb 1996
 * @author Cay Horstmann
 */

import java.util.*;
import corejava.*;

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

```
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry Hacker", 35000,
         new Day(1989,10,1));
      staff[1] = new Employee("Carl Cracker", 75000,
         new Day(1987,12,15));
      staff[2] = new Employee("Tony Tester", 38000,
         new Day(1990,3,15));
      int i;
      for (i = 0; i < 3; i++) staff[i].raiseSalary(5);
      for (i = 0; i < 3; i++) staff[i].print();
   }
}

class Employee
{
   public Employee(String n, double s, Day d)
   {
      name = n;
      salary = s;
      hireDay = d;
   }

   public void print()
   {
      System.out.println(name + " " + salary + " " + hireYear());
   }

   public void raiseSalary(double byPercent)
   {
      salary *= 1 + byPercent / 100;
   }

   public int hireYear()
   {
      return hireDay.getYear();
   }

   public int getName()
   {
      return name;
```

```
   }
   private String name;
   private double salary;
   private Day hireDay;
}
```

Nella classe **Employee** il metodo **hireYear()** è un accessore mentre il metodo **raiseSalary(double byPercent)** è un modificatore.
La classe *EmployeeTest* possiede un metodo *main* per il suo test. La sua compilazione con il comando:

$$javac\ EmployeeTest.java \tag{128}$$

dà origine ai file:

1. *EmployeeTest.class* corrispondente alla classe *EmployeeTest*;
2. *Employee.class* associato alla classe interna *Employee*.

L'esecuzione del codice compilato (o bytecode Java) tramite l'interprete *java* con il comando:

$$java\ EmployeeTest \tag{129}$$

fa si che venga eseguito il metodo *main* della classe *EmployeeTest* ovvero che vengano eseguite le operazioni in esso specificate quali:

1. inizializzazione di una struttura dati, un array di tre elementi di tipo *Employee*;
2. l'esecuzione, su ciascun elemento, del metodo *raiseSalary*;
3. l'esecuzione, su ciascun elemento, del metodo *print*;

La classe *EmployeeTest* non ha alcun metodo costruttore esplicitamente specificato mentre la classe *Employee* ne possiede uno ovvero:

```
public Employee(String n, double s, Day d)
{
      name = n;
      salary = s;
      hireDay = d;
}
```

che si occupa di creare oggetti istanze della classe inizializzando le variabili private della classe:

```
private String name;
private double salary;
private Day hireDay;
```

Il costruttore viene usato nella classe *EmployeeTest* per creare istanze della classe *Employee*:

```
staff[0] = new Employee("Harry Hacker", 35000,
   new Day(1989,10,1));
```

2.5 I package

Il linguaggio *Java* consente di raggruppare le classi in insiemi a ciascuno dei quali è assegnato un nome. In questo modo si raggruppano le classi sulle base di varie affinità funzionali formando i cosiddetti **package**.
I **package** consentono di organizzare il codice sviluppato da ciascun programmatore tenendolo distinto sia dall'insieme del codice fornito insieme all'ambiente run time e di sviluppo del linguaggio sia dal codice sviluppato da altri programmatori.
I **package** si distinguono in:

1. package standard;
2. package personali.

Nel codice della classe *EmployeeTest* sono inclusi, con una operazione di **import**, i package:

1. *java.util*, Java standard, di cui si importano tutte le classi (come specificato dal carattere jolly $*$);
2. *corejava*, sviluppato ad hoc dall'autore del codice ([HC99]), di cui si importano tutte le classi (come specificato dal carattere jolly $*$).

I package Java standard formano una gerarchia a vari livelli di annidamento la cui radice è rappresentata dal package *java*. In questo modo si riesce a garantire l'unicità dei nomi dei package allo stesso modo in cui si garantisce l'unicità dei nomi dei file nel fs con il **path name assoluto**.
Qui di seguito si riporta parte della gerarchia *Java* standard come visibile attraverso la documentazione *html* della *API* (Application Programming Interface) del linguaggio:

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi
java.awt.image
java.awt.image.renderable
java.awt.print
java.beans
java.beans.beancontext
java.io
java.lang
java.lang.ref
java.lang.reflect
java.math
java.net
java.nio
java.nio.channels
java.nio.channels.spi
java.nio.charset
java.nio.charset.spi
java.rmi
java.rmi.activation
java.rmi.dgc
java.rmi.registry
java.rmi.server
java.security
java.security.acl
java.security.cert
java.security.interfaces
java.security.spec
java.sql
java.text
java.util
java.util.jar
java.util.logging

```
java.util.prefs
java.util.regex
java.util.zip
```

Delle classi usate a lezione la classe *System* è caratterizzata come:

```
java.lang
Class System

java.lang.Object
  |
  +--java.lang.System
```

ovvero come estensione della classe *Object* del package *java.lang*. Per la classe *Math* si ha:

```
java.lang
Class Math

java.lang.Object
  |
  +--java.lang.Math
```

L'altra gerarchia ha come radice il package **corejava**. In questo modo si possono avere due package di nome **util**:

1. *java.util*;
2. *corejava.util*;

i cui sottopackage e metodi classi non si confondono, avendo nomi globali univocamente determinati.
A questo punto sorgono due problemi:

1. come si usano i package;
2. come il compilatore trova i package.

I package possono essere usati in due modi:

1. come prefissi dei nomi delle classi;
2. all'interno di istruzioni di **import**.

Un esempio del primo tipo è il seguente:

$$int\, i = Input.readInt(); \tag{130}$$

in cui si usa la classe *Input* (vedi la sezione 2.6) del package corrente (rappresentato dalla directory corrente o di lavoro) per identificare in modo univoco il metodo $readInt()$.
Un esempio del secondo tipo lo si ha nel codice seguente:

```
import java.util.*;
import corejava.*;
```

che ci permette di accedere a tutte le classi dei package specificati e, quindi, a tutti i loro metodi.
L'altro problema da risolvere è capire come il compilatore accede ai vari package. Il primo passo è capire che :

1. i file che implementano i package e le classi del package *corejava* sono contenuti nella sottodirectory di nome *corejava*;

2. i file che implementano i package e le classi del package *java.util* sono contenuti nella sottodirectory di nome *java*
   util;

3. considerazioni analoghe valgono per tutti gli altri package;

4. tali sottodirectory non devono necessariamente avere una directory radice comune, l'unica condizione necessaria è che i loro **path name assoluti** siano contenuti nella variabile di ambiente **CLASSPATH**.

Se, ad esempio, si ha:

$$CLASSPATH = c:\backslash jdk \backslash lib \backslash classes.zip; c:\backslash CoreJava; \tag{131}$$

Se si ha:

```
import java.util.*;
import corejava.*;
```

le classi usate nel codice *Java* sono ricercate, nell'ordine:

1. nel file di archivio $c:\backslash jdk \backslash lib \backslash classes.zip$;

2. nella directory $c:\backslash CoreJava$;

3. nel package corrente (ovvero nei file *.java* della directory corrente;

fino a che la ricerca non ha dato esito positivo (e si riesce ad utilizzare la classe e i metodi relativi) oppure non si sono esaurite tutte le possibilità e la ricerca termina con un messaggio di errore. Negli esempi visti a lezione, se nella directory corrente (ovvero nel package corrente) non è contenuto il file *Input.java*, che implementa la classe *Input*, ogni riferimento a metodi della classe dà luogo ad un messaggio di errore dal momento che il compilatore non è in grado di localizzare il codice che implementa il metodo.

2.6 La gestione dell'input da tastiera

Per la gestione dell'input da tastiera si può fare uso delle classi contenute nel package *java.io* oppure dei metodi contenuti nel file *Input.java* sviluppato dai docenti del corso di *LIP* del Dipartimento di Informatica dell'Università di Pisa. Tale file (il cui contenuto è riportato qui di seguito con alcuni commenti) deve essere copiato nella stessa directory che contiene il codice sorgente che utilizza i metodi in esso contenuti. Il file *Input.java* definisce la classe omonima con un certo numero di metodi che consentono di acquisire una stringa, un carattere, un double o un int. I metodi fanno uso di un elemento *reader* di tipo *BufferedReader* che accede allo standard input (la tastiera o *System.in*) tramite un elemento della classe *InputStreamReader*. La *API* di *Java* definisce il *BufferedReader* come:

```
public BufferedReader(Reader in)

    Create a buffering character-input stream
    that uses a default-sized input buffer.

Parameters:
    in - A Reader
```

Come *reader* si usa un elemento della classe *InputStreamReader*.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

/**
    Una semplice classe per leggere stringhe e numeri
    dallo standard input.
*/
```

```
public class Input{

    private static BufferedReader reader =
new BufferedReader(new InputStreamReader(System.in));

    /**
       Legge una linea di input. Nell'improbabile caso di una
       IOException, il programma termina.
       @return restituisce la linea di input che l'utente ha battuto.
    */
    public static String readLine(){
String inputLine = "";
try{
    inputLine = reader.readLine();
}
catch(IOException e){
    System.out.println(e);
    System.exit(1);
}
return inputLine;
    }

    /**
       Legge una linea di input e la converte in un intero.
       Eventuali spazi bianchi prima e dopo l'intero vengono ignorati.
       @return l'intero dato in input dall'utente
    */
    public static int readInt(){
String inputString = readLine();
inputString = inputString.trim();
int n = Integer.parseInt(inputString);
return n;
    }

    /**
       Legge una linea di input e la converte in un numero
       in virgola mobile.  Eventuali spazi bianchi prima e
       dopo il numero vengono ignorati.
       @return il numero dato in input dall'utente
    */
```

```
    public static double readDouble(){
String inputString = readLine();
inputString = inputString.trim();
double x = Double.parseDouble(inputString);
return x;
    }

    /**
       Legge una linea di input e ne estrae il primo carattere.
       @return il primo carattere della riga data in input dall'utente
    */
    public static char readChar(){
String inputString = readLine();
char c = inputString.charAt(0);
return c;
    }
}
```

Il file si apre con tre operazioni di *import* delle necessarie classi del package *java.io*. Nei casi in cui ciò è necessario sono gestite le condizioni anormali (o eccezioni) che si possono verificare in input. I metodi sono definiti *Static* in modo da essere utilizzati senza istanziare nessun oggetto della classe ma secondo la sintassi:

$$Input.nome\_metodo \tag{132}$$

Ad esempio, la chiamata:

$$Input.readDouble() \tag{133}$$

accetta un input da tastiera e lo restituisce come se fosse un valore di tipo *double*.

3 Introduzione a Eclipse

Eclipse ([Lia05]) è un **ambiente di sviluppo integrato** (o *IDE* da **integrated development environment**) per lo sviluppo rapido di programmi in Java. Il programma mette a disposizione:

1. editor,
2. compilatore,
3. interprete,
4. debugger,
5. help in linea,

integrati in una unica interfaccia grafica. Lo scopo di queste note, tratte da ([Lia05]), è quello di mostrare come si possono creare progetti o programmi, compilarli ed eseguirli.

3.1 Introduzione

Gli elementi di base di**Eclipse** sono:

1. il **workspace** ovvero una cartella/directory del file system che contiene tutte le risorse che vengono sviluppate da un programmatore;
2. le risorse sono essenzialmente di tre tipi:
   (a) file di codice sorgente;
   (b) cartelle o directory;
   (c) progetti che contengono cartelle e file.
3.

È possibile cambiare workspace[2] con il comando $File \longrightarrow SwitchWorkspace$. Una volta selezionato un workspace si apre una finestra di **Workbench** che rappresenta un ambiente classico di desktop con una o più **prospettive** o **perspective**. Se al momento di mandare in esecuzione **Eclipse** viene visualizzata la **Welcome view** è possibile passare al **Workbench** utilizzando il

[2]Nel seguito useremo la sintassi:

$$A \longrightarrow B \tag{134}$$

per individuare il menù B contenuto nel meù A.

pulsaste **Workbench**. La **Welcome view** è riottenibile in ogni momento usando il comando $Help \longrightarrow Welcome$. Una **perspective** definisce un insieme di viste (**view**) e di editor. Le modifiche apportate alle viste sono salvate automaticamente. L'uso degli editor prevede che gli elementi su cui si agisce debbano essere aperti, modificati, salvati e chiusi.

3.2 I primi passi

Al momento in cui **Eclipse** viene mandato in esecuzione viene visualizzata sullo schermo la finestra di **Workspace Launcher** che consente di scegliere una directory in cui verranno memorizzati i file dei progetti dell'utente, directory detta **workspace** o **spazio di lavoro**. È possibile accettare la directory proposta oppure eseguire una navigazione nel fs e sceglierne una ad hoc. La figura 2 illustra la finestra di **Workspace Launcher** in ambiente

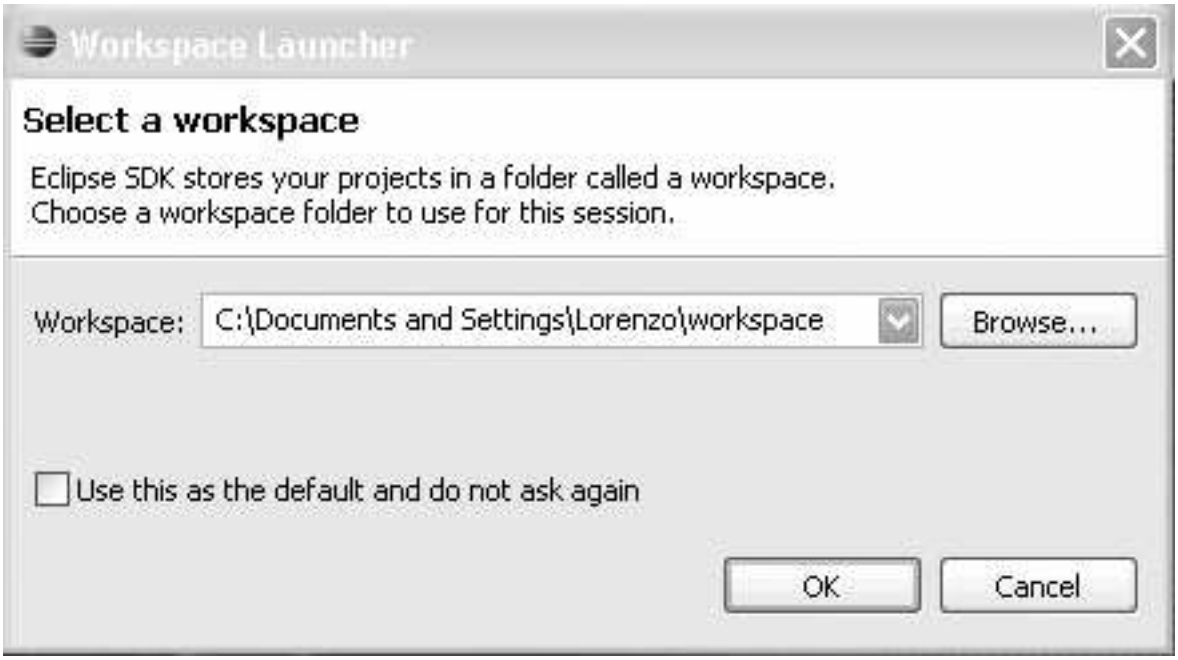

Figura 2: *Finestra di* ***Workspace Launcher***

Windows. Si vede come sia possibile:

1. accettare la directory proposta (pulsante OK),
2. terminare il programma (pulsante $Cancel$),
3. scegliere una directory diversa (pulsante $Browse \ldots$),

4. scegliere un workspace fra quelli usati in passato accedendo alla history (pulsante $\vee$).

È possibile, inoltre, fare in modo che tale finestra non sia più visualizzata spuntando la casella etichettata "Use this as the default and do not ask again". Si sconsiglia di effettuare tale scelta in modo da avere sempre la possibilità di scegliere uno spazio di lavoro diverso per i vari progetti.
Dopo aver fatto la scelta del workspace e premuto il tasto OK il programma, se è la prima volta che lo si usa, richiede che l'utente scelga la prospettiva (o *perspective*) che controlla ciò che compare nei menù e nelle barre degli strumenti. Le prospettive disponibili sono rintracciabili nel men'u

$$Window \longrightarrow OpnePerspective \tag{135}$$

Le prospettive disponibili sono:

1. Debug;
2. Java;
3. Java Browsing,
4. altre, accessibili con un pulsante $Other \ldots$.

Se si vogliono scrivere programmi Java si seleziona la seconda di tali prospettive. Ai successivi riavvii del programma viene presentata l'ultima prospettiva usata ma l'utente la può cambiare in una diversa fra quelle disponibili.

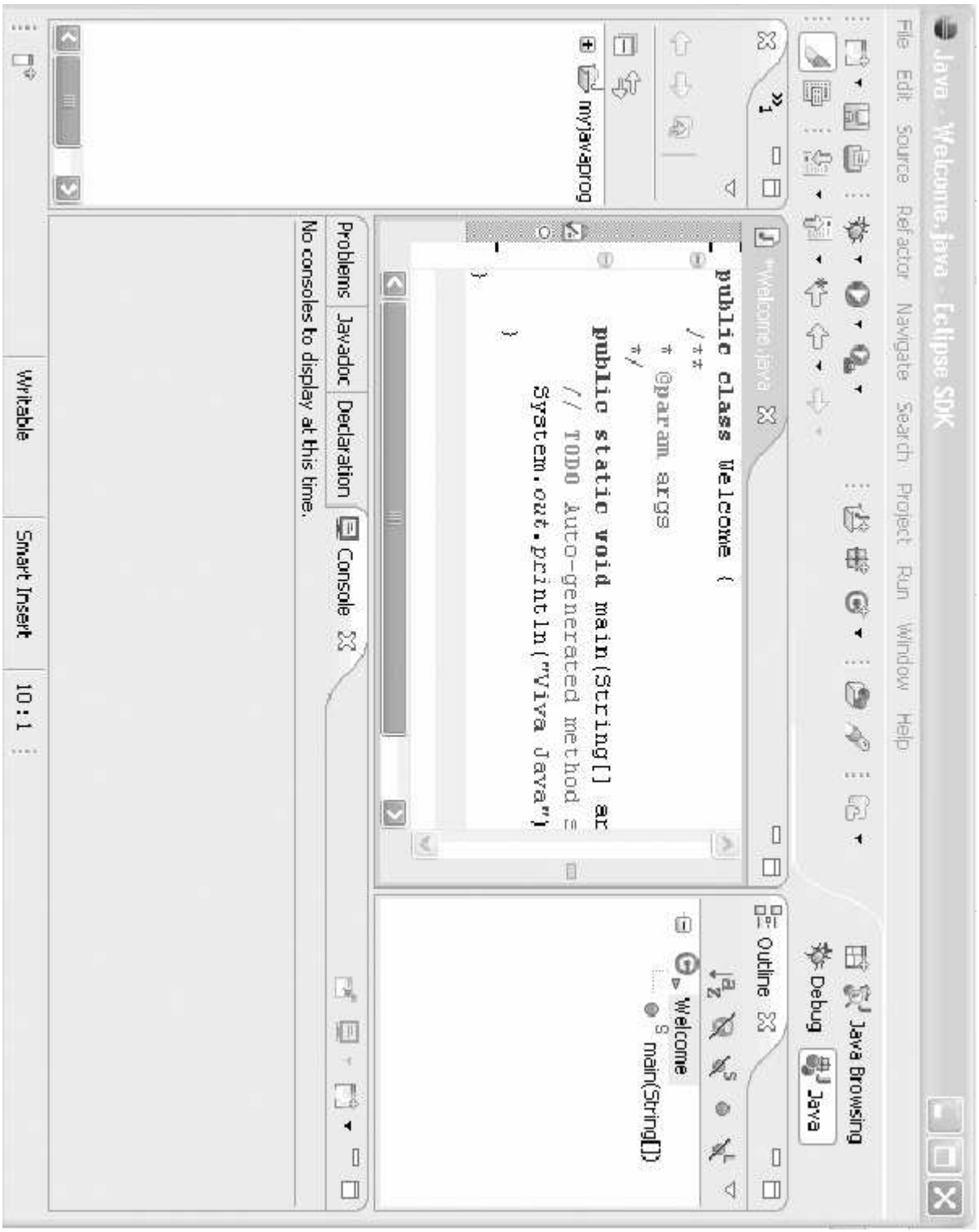

Figura 3: *Esempio di* ***prospettiva Java***

La figura 3 presenta un esempio di prospettiva Java in cui viene visualizzato un semplice programma in Java. Proseguendo l'analisi passo−passo, dopo aver scelto la prospettiva si può voler creare un **progetto**.

3.3 Creare un progetto

È possibile creare un progetto con il menù:

$$File \longrightarrow New \longrightarrow Project \tag{136}$$

che visualizza il cosiddetto **New project Wizard** che guida, passo passo, nella creazione di un nuovo progetto. Tramite il menù:

$$File \longrightarrow New \tag{137}$$

si può creare un package (sottomenù $Package$) o una classe (sottomenù $Class$) o una interfaccia (sottomenù $Interface$) o altro.
Il **New project Wizard** (vedi la figura 4) è caratterizzato da una successione di finestre in ciascuna delle quali l'utente deve immettere delle informazioni. Ogni finestra è caratterizzata, in basso, da alcuni pulsanti, non sempre attivi, quali:

1. $< Back$ per tornare alla finestra precedente,
2. $Next >$ per passare alla finestra successiva,
3. $Finish$ per terminare il $Wizard$ convalidando i dati immessi;
4. $Cancel$ per terminare il $Wizard$ senza convalidare i dati immessi.

La prima informazione da immettere, al posto di **type filter text**, è il tipo del Wizard. Si deve selezionare $Java$ e premere il pulsante $Next >$ ora accessibile. A questo punto (vedi la figura 5) si deve immettere il nome del progetto, ad esempio $myjavaprograms$, che diventa il nome della sottodirectory del workspace che conterrà i file del progetto che si stà creando e si devono selezionare le opzioni:

1. **Create new project in workspace**
2. **Use default JRE**
3. **Use project folder as root for sources and class files**

e premere il pulsante $Finish$ per creare il progetto.

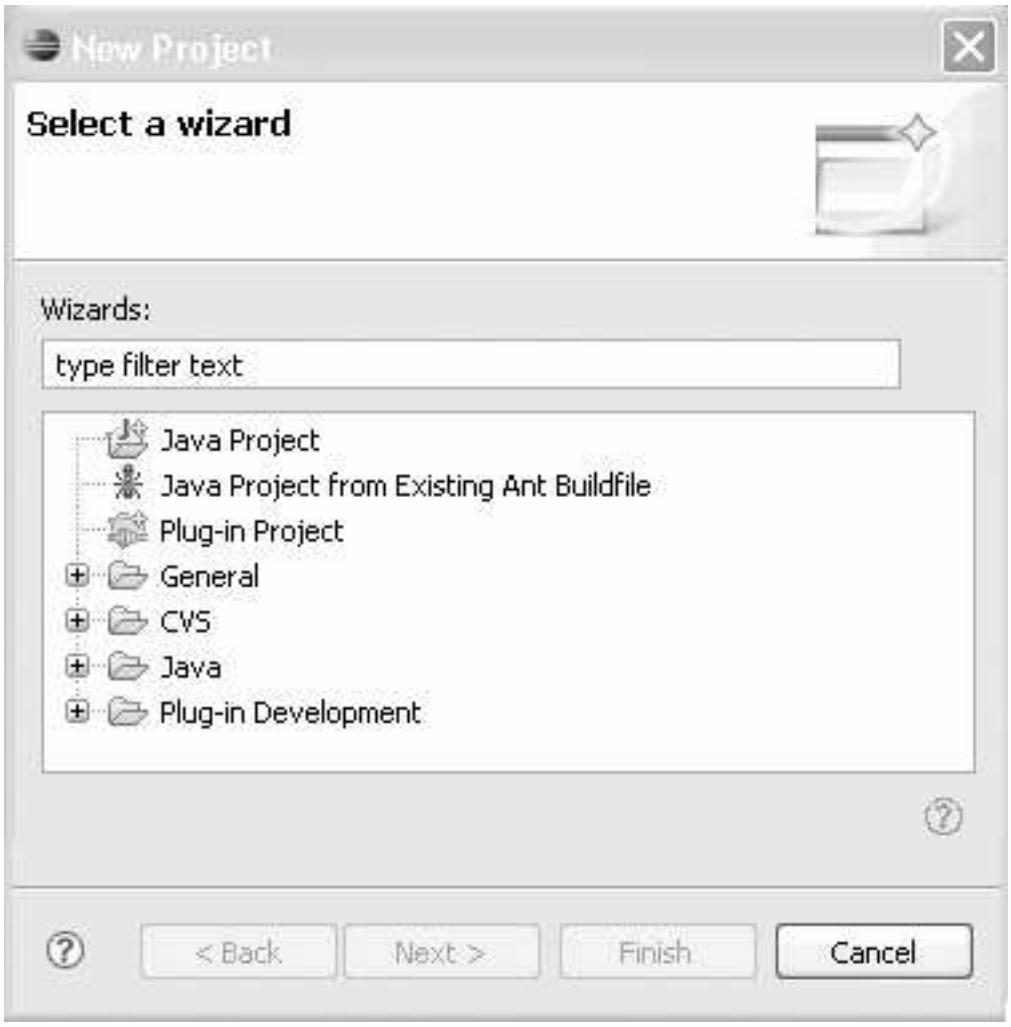

Figura 4: *Fase iniziale del* ***Project Wizard***

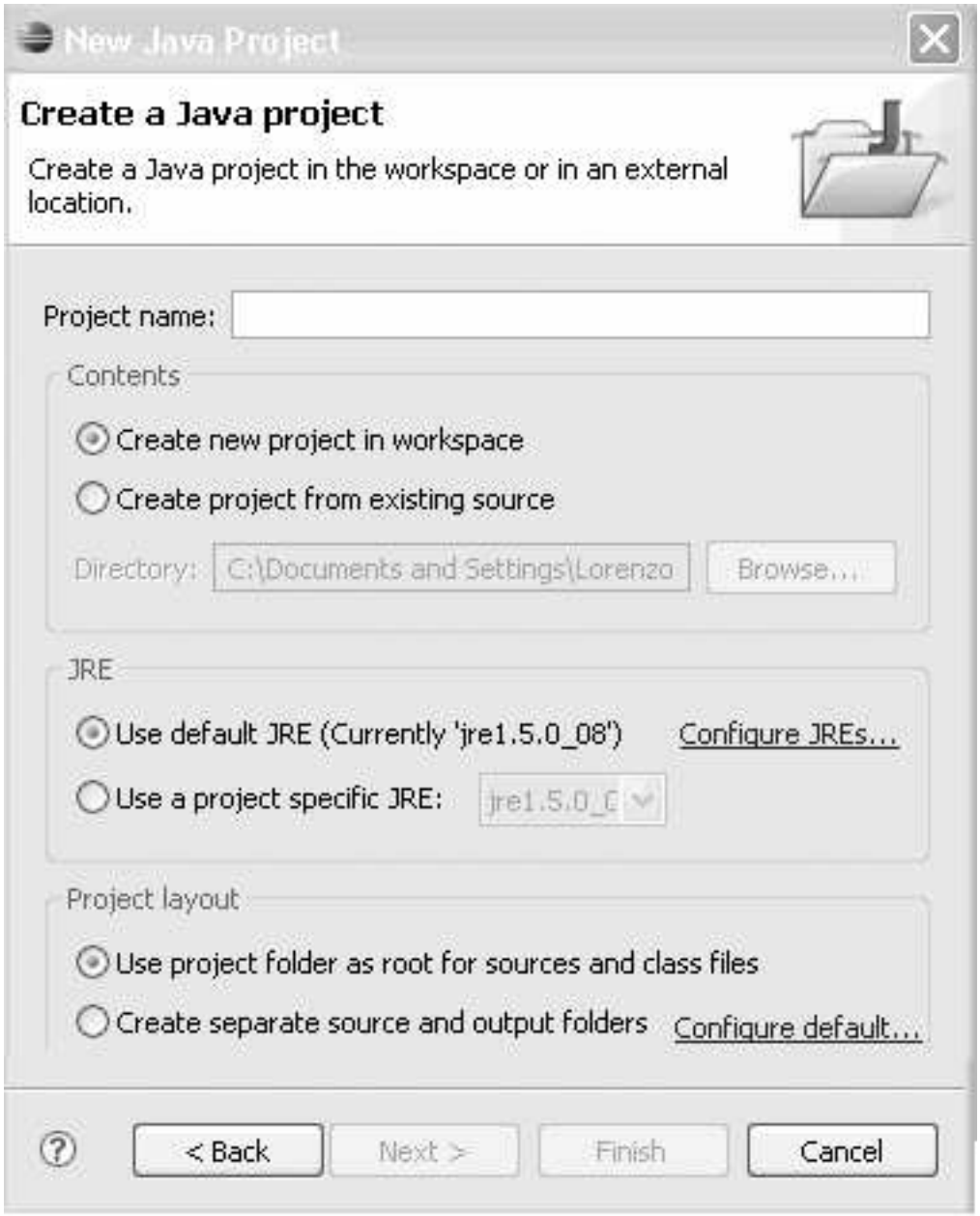

Figura 5: *Seconda fase del* ***Project Wizard***

Se il nome immesso corrisponde a quello di un progetto già esistente il Wizard segnala l'errore e non abilita il pulsante *Finish*. Le alternative possibili consentono:

1. di creare il progetto usando codice Java già scritto (e rintracciabile con il pulsante di *Browse* ...);
2. usare un altro ambiente java a tempo di esecuzione *JRE*;
3. creare directory separate per il codice sorgente e l'output.

Se invece di premere *Finish* si preme *Next* > si passa ad una ulteriore finestra del Wizard che ci permette di:

1. creare una nuova directory per il codice sorgente;
2. collegare altro codice sorgente;
3. altro:

In ogni caso, per terminare la creazione del progetto, si deve premere il pulsante *Finish*, se disponibile.

3.4 Creare un programma

A questo punto si può creare un programma all'interno del progetto usando il menù:

$$File \longrightarrow New \longrightarrow Class \tag{138}$$

in modo da visualizzare il **New Java Class Wizard**. Nella finestra di dialogo di figura 6 è necessario inserire il nome nel campo etichettato come *Name* : mentre non è necessario specificare:

1. la directory, che coincide con quella specificata a livello di progetto;
2. il **package**, che coincide con quello di default.

Se si vuole che la classe che si stà creando contenga un metodo *main* è necessario spuntare la apposita check box. Inserito il nome il pulsante *Finish* si abilita permettendoci di completare la creazione della classe. Il modificatore va mantenuto *public* se la classe che si crea è la classe principale del file. Lo stesso vale, in genere, per la superclasse che va mantenuta al valore di default a meno che la classe che si crea non estenda una casse particolare.

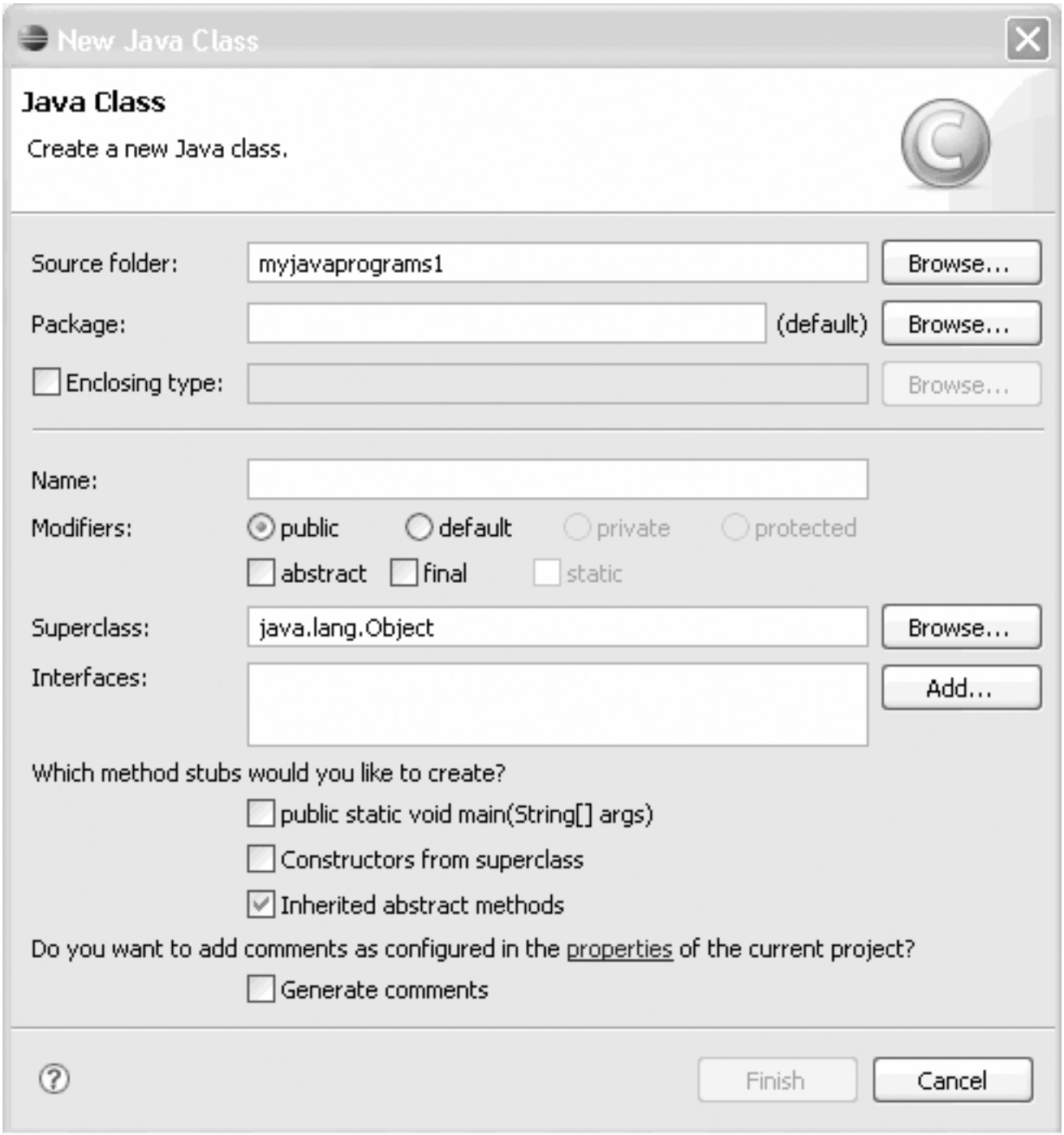

Figura 6: *Prima fase del* ***New Java Class Wizard***

Una volta che si è premuto il pulsante *Finish*, *Eclipse* crea per noi lo "scheletro" o "template" della classe *Welcome* con la struttura del metodo *main*.
La situazione finale è quella rappresentata in figura 7 A questo punto non

```
public class Welcome {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

Figura 7: *Il template della classe Welcome*

resta che scrivere il codice che svolge le funzioni che abbiamo attribuito alla classe. Per la scrittura del codice si può beneficiare della funzionalità di completamento automatico del codice fornita da Eclipse. Man mano che si scrive il codice vengono forniti suggerimenti per il suo completamento, se si fanno brevi pause dopo l'inserimento di elementi sintattici chiave come il ".".
Un modo per completare il codice dell'esempio è quello di inserire nel metodo *main* una chiamata al metodo *System.out.println* per ottenere la stampa di una stringa sul video. La funzione di completamento automatico del codice entra in gioco dopo che si è digitato il primo . mediante una finestra di dialogo che ci presenta una lista da cui possiamo scegliere e lo stesso vale dopo il secondo ".".

3.5 Compilazione ed esecuzione

Di default, il codice sorgente viene compilato dinamicamente man mano che viene scritto, in modo che eventuali errori sintattici sono rilevati subito

ed evidenziati all'interno del codice mediante l'uso di sottolineature rosse posizionate nei punti in cui sono presenti errori di sintassi.
Questa caratteristica è governata dalla voce *Buildautomatically* nel menù *Project*, voce selezionata di default. Se si vuole evitare la compilazione dinamica basta deselezionare tale voce.
Se il codice è corretto lo si può mandare in esecuzione nei modi seguenti:

1. cliccando con il pulsante destro del mouse nella finestra che contiene la classe e selezionando o $RunAs \longrightarrow JavaApplication$ oppure $Run \ldots$ e utilizzandole finestre di dialogo che seguono;

2. selezionando le voci analoghe della voce di menù *Run* nella lista di menù subito sotto la barra del titolo.

3.6 Qualcosa di più

Eclipse è un *IDE* open source distribuito in diverse modalità di packaging caratterizzato dai servizi forniti da un piccolo kernel a run time mentre ogni altra funzionalità è fornita da plug-in comunicanti. L'uso di plug-in rende il progetto facilmente estensibile dato che la piattaforma base fornisce un modo efficiente per la comunicazione fra plug-in in modo che nuove feature possano essere aggiunte senza sforzo. I plug-in sono rilevati a run time e sono caricati on demand ovvero solo se vengono utilizzati effettivamente.
L'elemento base di **Eclipse** è il **Workspace** che si occupa di gestire le risorse di utente organizzate in uno o più progetti a ciascuno dei quali è associata una cartella detta "cartella di Workspace".
L'interfaccia grafica di Eclipse è rappresentata dal **Workbench** che visualizza menù e barre degli strumenti ed 'e organizzato in **prospettive** ciascuna della quali contiene **viste** ed **editori**.
Le **viste** consentono di visualizzare, ad esempio, la struttura di un package, eventuali errori del codice, la console su cui compaiono gli output delle *System.out.println* e altre informazioni. Gli **editori** consentono di modificare il codice sorgente Java.
Come abbiamo già visto, lavorando in Java, i passi da seguire sono i seguenti:

1. creare un progetto;

2. creare una classe specificando:

   (a) il nome della classe;

   (b) il package che la contiene;

   (c) la classe che la contiene ovvero la superclasse;

(d) le interfacce implementate dalla classe;

(e) se la classe è **public**, **abstract** (ovvero non contiene tutte le implementazioni dei metodi), **final**;

(f) quali template di metodi contiene.

Nella creazione di una classe **Eclipse** fornisce suggerimenti per la creazione di classi che rispettano le specifiche sintattiche del linguaggio e visualizza eventuali errori che possono essere corretti "on the fly".

3.7 Un altro esempio di programma Java con Eclipse

Creeremo ora, passo per passo, un programma Java utilizzabile dalla linea di comando.
Si parte dalla prima versione.
Si eseguono i seguenti passi:

1. per prima cosa si manda in esecuzione Eclipse;
2. si usa la prospettiva *Java*;
3. poi si crea un nuovo progetto $File \longrightarrow New \longrightarrow Project\ldots$ selezionando **Java project** poi $Next >$, dando il nome $MagicNumberText$ al progetto e premendo $Finish$;
4. a questo punto il progetto $MagicNumberText$ compare nella vista **Package Explorer**;
5. è ora possibile aggiungere la prima classe al progetto $File \longrightarrow New \longrightarrow Class$;
6. come nome della classe si digita $MagicNumberText\_Main$ e poi si preme il pulsante $Finish$.

A questo punto la situazione del codice nella finestra di editing centrale del Workbench è la seguente:

```
 public class MagicNumberText_Main {

/**
 * @param args
 */
public static void main(String[] args) {
// TODO Auto-generated method stub
```

```
}

 }
```

Si può ora aggiungere il codice che si vuole all'interno del metodo *main*. Supponiamo di rimanere in un caso semplice e di aggiungere solo una istruzione di stampa di una stringa. Il codice completo è:

```
public class MagicNumberText_Main {

/**
 * @param args
 */
public static void main(String[] args) {
// TODO Auto-generated method stub
        System.out.println("Ciao mondo!!");
    }
}
```

A questo punto si vuole eseguire il programma e per fare ciò o si usa l'icona sotto la barra dei menù contenente un triangolino rivolto a destra oppure si seleziona uno dei comandi visti in precedenza: compare una finestra di dialogo tramite la quale selezioniamo in programma da eseguire come **java Application** e dopo poco, all'interno della Console, compare la scritta voluta.
A questo punto si vogliono aggiungere alcune funzionalità al programma in modo da aggiungere:

1. funzioni di ingresso-uscita;
2. interazioni con l'utente.

Lo scopo è quello di accettare dati dall'utente, eseguire un po' di calcoli su tali dati e restituire un risultato. I passi del programma sono i seguenti:

1. si visualizza una richiesta all'utente,
2. si accetta una stringa in input,
3. si calcola il valore numerico corrispondente ad ogni carattere (il suo codice ASCII);

4. si sommano i codici ASCII e poi si sommano le cifre del risultato fino ad arrivare ad una sola cifra sola che rappresenta il cosiddetto numero fortunato associato alla stringa.

Il codice risultante è il seguente:

```
import java.io.*;

public class MagicNumberText_Main {

/**
 * @param args
 */
public static void main(String[] args) {
// TODO Auto-generated method stub
System.out.println("Benvenuto al programma di calcolo del numero fortunato!!");
System.out.print("Immetti una stringa:");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String userName=null;
try{
userName=br.readLine();
} catch(IOException ioe) {
System.out.println("Errore!!");
System.exit(1);
}
    int i;
    int aux=0;
        for (i=0;i<userName.length();i++) {
         aux=aux+userName.charAt(i);
        }
        int result=0;
        while(aux > 0){
         result=result+(aux%10);
         aux=aux/10;
         if(aux==0 && result > 10){ aux=result; result=0;
        }
        }
        System.out.println("Numero fortunato= "+result);
        }
}
```

Il programma così scritto deve essere eseguito dalla linea di comando.
Se invece di usare l'ambiente integrato di**Eclipse** si procede a linea di comando si deve:

1. compilare il programma con il comando:

$$javacMagicNumberText_Main.java \tag{139}$$

2. eseguire il programma con il comando:

$$javaMagicNumberText_Main \tag{140}$$

Il programma visualizza la stringa **Benvenuto al programma di calcolo del numero fortunato!!** seguita, su una riga diversa, dalla stringa **Immetti una stringa:**. L'utente immette una stringa e il programma:

1. la converte in una sola cifra come specificato;

2. visualizza il risultato e termina.

Si ritengono degni di nota l'istruzione **import java.io.*;** e il blocco:

```
try{
userName=br.readLine();
} catch(IOException ioe) {
  System.out.println("Errore!!");
  System.exit(1);
}
```

con il quale il programma gestisce gli eventuali errori (eccezioni) che si possono avere a livello di metodo di input:

$$br.readLine() \tag{141}$$

3.8 Import e Export

Invece di inserire esplicitamente la riga:

$$importjava.io.*; \tag{142}$$

è possibile importare le classi necessarie con il comando:

$$File \longrightarrow Import\ldots \tag{143}$$

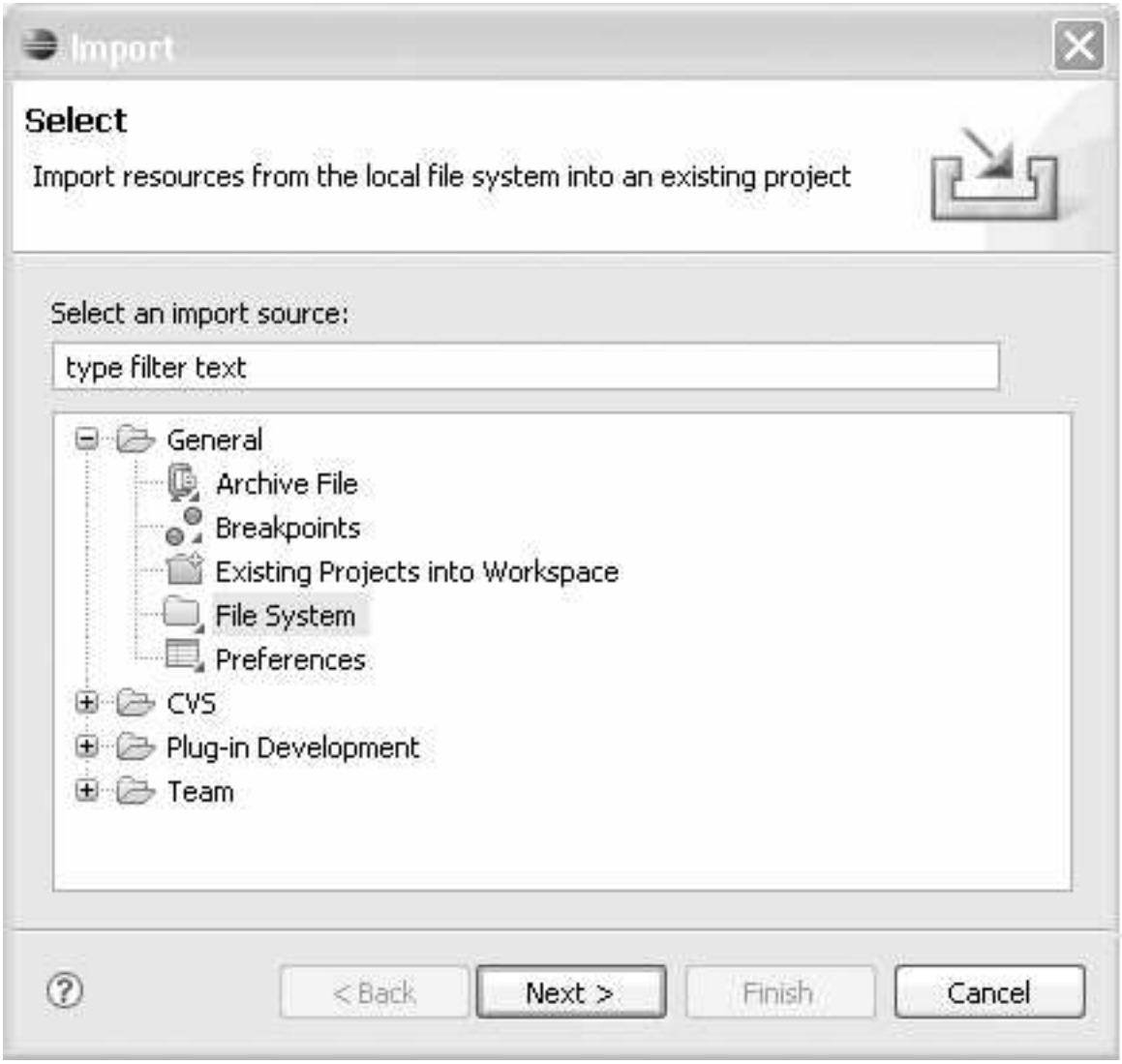

Figura 8: *Primo step per l'**Import***

Figura 9: *Secondo step per l'**Import***

L'esecuzione del comando causa l'aprirsi di una finestra di dialogo (vedi la figura 8) mediante la quale si accede al file system per localizzare il file da importare. Negli esempi visti a lezione la gestione dell'ingresso dati da tastiera avviene tramite il file *Input.java* che deve, pertanto, essere importato in ogni progetto che fa uso di comandi per gestire l'immissione di dati da tastiera.
In figura figura 8 si vede selezionata la voce **File System** e il pulsante **Next>** è accessibile in modo che si possa passare alla finestra mostrata in figura 9. Tramite la finestra di figura 9 si può esplorare il file system e localizzare il file da importare.
L'operazione duale è l'operazione di *Export* accessibile tramite il comando:

$$File \longrightarrow Export\ldots \tag{144}$$

L'operazione di *Exoprt...* (vedi le figure 10 e 11) prevede che si selezioni tipologia della destinazione dei file da esportare (in questo caso il **File System**) in modo che, premendo il pulsante **Next>** si passi alla finestra di selezione degli elementi da esportare, selezione che viene resa effettiva dalla selezione del pulsante **Finish**.

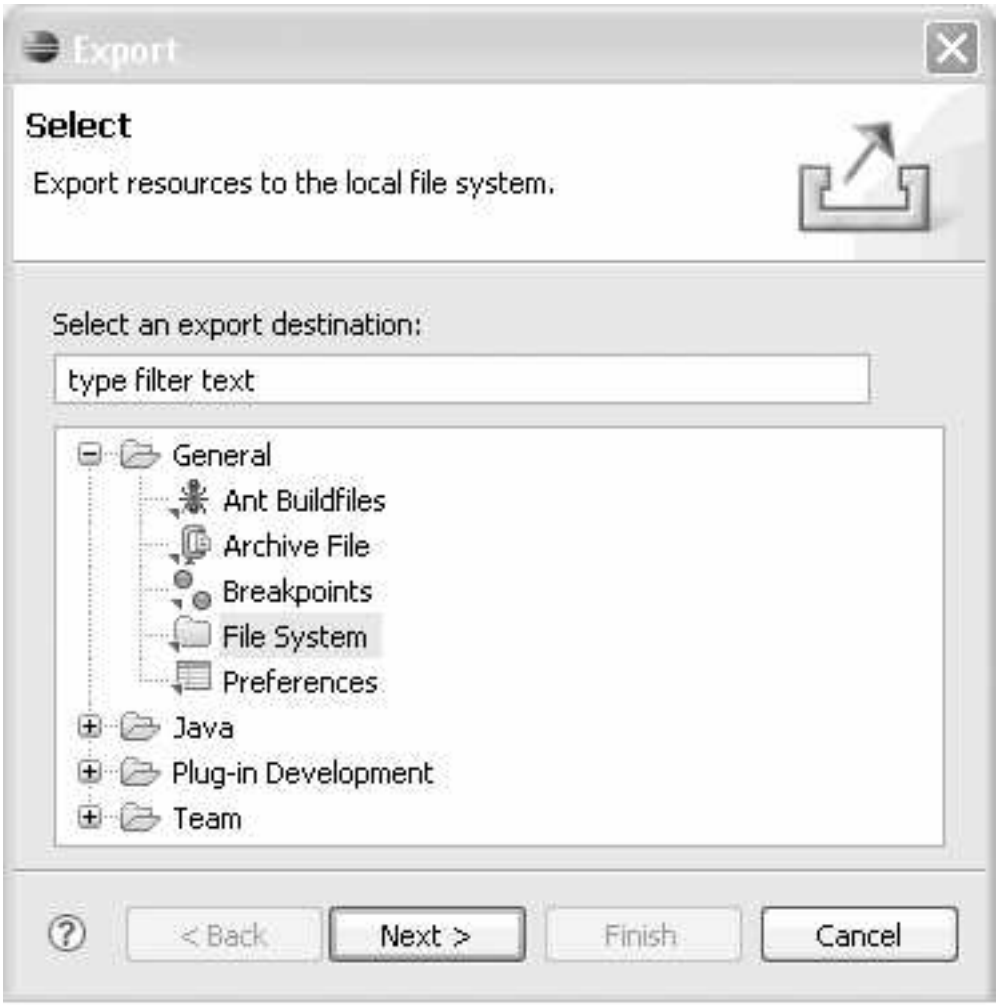

Figura 10: *Primo step per l'**Export***

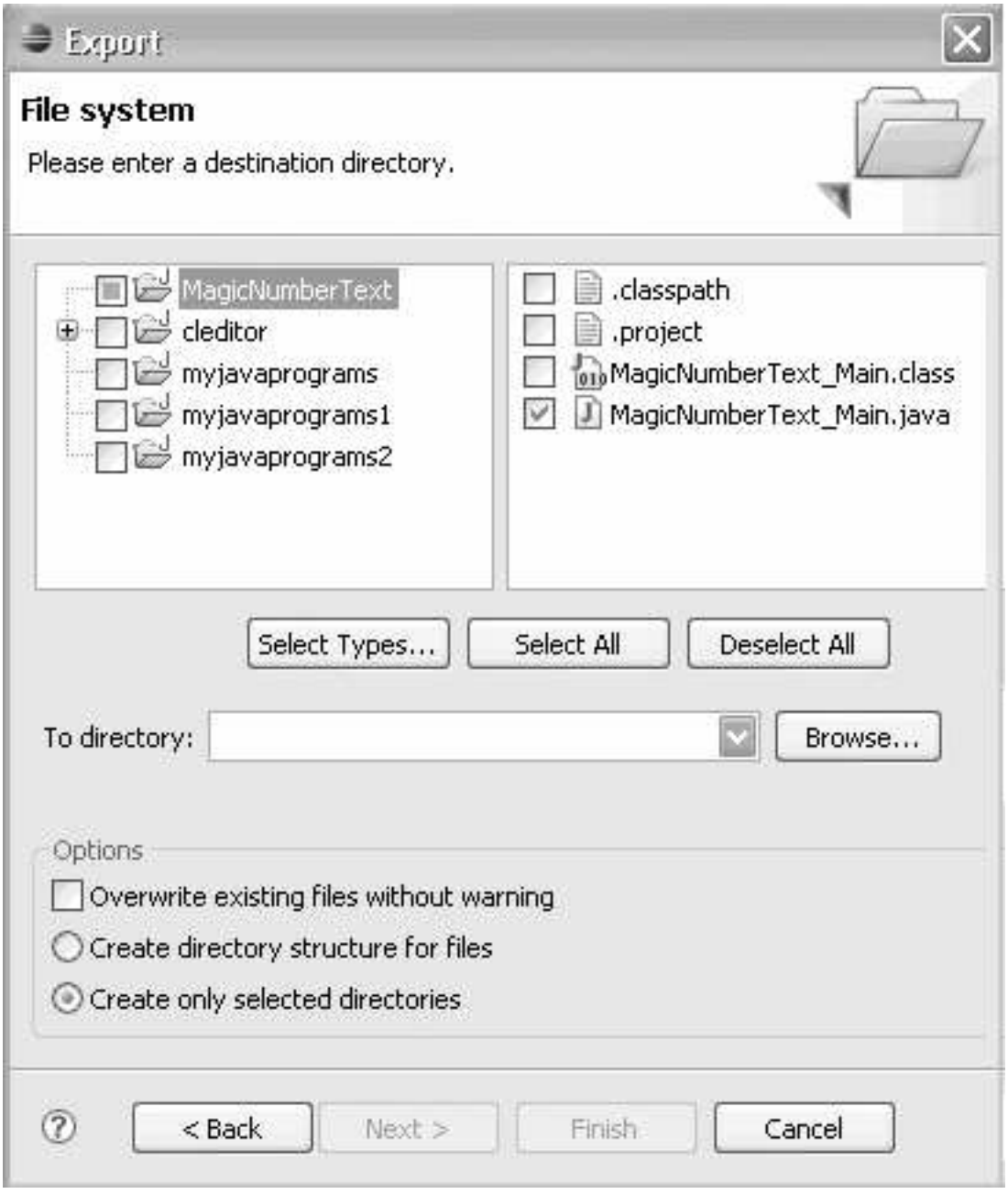

Figura 11: *Secondo step per l'**Export***

Riferimenti bibliografici

[CM04] Paolo Coppola and Stefano Mizzano. *Laboratorio di programmazione in Java.* Apogeo, 2004.

[HC99] Cay S. Horstmann and Gary Cornell. *Java 2. I fondamenti.* Mc Graw Hill, 1999.

[Lia05] Y. Daniel Liang. *Supplement J: Eclipse Tutorial.* Internet version, 2005.

[Pet96] Richard Petersen. *Linux. La grande guida.* Mc Graw Hill, 1996.