

# 1. Caratteristiche del linguaggio

## 1.1. Considerazioni preliminari

Il linguaggio Java, sviluppato dalla *Sun microsystem*, ha caratteristiche linguistiche e implementative che lo rendono adatto a essere utilizzato in ambiti applicativi attualmente rilevanti, tipicamente in quello delle reti informatiche.

Un programma è in genere costituito da parti separate (*processi*) distribuite su vari nodi di una rete, ciascun nodo con caratteristiche architetturelle proprie. In questo volume, tuttavia, ci limiteremo a considerare programmi semplici costituiti da un solo processo (e quindi destinati a essere eseguiti su un singolo nodo), rimandando al volume II l'esame del caso più generale.

L'utente abitualmente scrive un programma utilizzando un linguaggio di programmazione ad alto livello (programma *sorgente*): questo deve essere preliminarmente tradotto (per mezzo di un *compilatore*) nel linguaggio dell'elaboratore su cui si opera (programma *oggetto*, espresso in linguaggio *nativo*) prima di poter essere eseguito. Un programma è portabile da un'architettura a un'altra se il linguaggio in cui è scritto è indipendente dall'architettura stessa: i programmi sorgente hanno questa caratteristica in modo quasi completo.

La proprietà basilare di Java è quella di assicurare la completa portabilità, sia a livello di programmi sorgente, ma soprattutto a livello di programmi oggetto.

Per quanto riguarda il linguaggio sorgente, Java specifica che devono essere utilizzati sempre i caratteri *unicode* (codificati ciascuno con 16 bit), con la possibilità di usare simboli delle più importanti lingue del mondo. Inoltre, Java definisce le modalità di rappresentazione dei numeri (per esempio, il tipo *int* è rappresentato sempre in complemento a 2 e con 32 bit). Infine Java utilizza le cosiddette Java API (*Application Programming Interface*), un insieme di librerie di uso generale o orientate a particolari settori (programmazione distribuita, interfacce grafiche, interazioni con le basi di dati, eccetera), che devono essere uguali per ogni architettura.

Per quanto riguarda il linguaggio oggetto, i vari compilatori Java (uno per ogni architettura) generano programmi scritti nello stesso codice macchina intermedio (detto *byte code*): un programma così tradotto viene poi eseguito da un interprete, detto JVM (*Java Virtual Machine*, una per ogni architettura). Il byte-code costituisce quindi un linguaggio oggetto portatile, che può essere trasferito da un'architettura a un'altra, purché entrambe siano provviste di interprete.

Una piattaforma Java (implementazione del linguaggio per una data architettura) comprende pertanto tre componenti, ciascuna delle quali utilizza il linguaggio nativo: il compilatore, la JVM e le Java API. La piattaforma specifica per il sistema che si utilizza può essere prelevata dal sito:

<http://www.java.com>

Dal punto di vista linguistico, un programma è composto da *classi* (e anche da *interfacce*) organizzate in gerarchie ed eventualmente raggruppate logicamente in *package*: tipicamente, un package viene racchiuso in una cartella che ha il nome del package stesso. Fra le varie classi, ne esiste una che contiene la funzione *main()* (classe principale), funzione che viene eseguita per prima.

Un gruppo di classi funzionalmente omogeneo (costituenti uno o più package) rappresenta una *componente software*.

## 1.2. Ulteriori proprietà di Java

Java è un linguaggio orientato agli oggetti, ma con una struttura

semplice che non prevede alcuni costrutti linguistici, tipici di altri linguaggi di programmazione, che possono dar luogo ad errori insidiosi. Esso possiede però tutte le caratteristiche basilari per questo genere di programmazione, come l'incapsulamento dei dati, l'ereditarietà e il polimorfismo.

Alcune proprietà di Java rendono questo linguaggio robusto, come il controllo stretto sui tipi, la verifica dei limiti di un array, i tipi enumerazione sicuri, la possibilità di avere riferimenti (puntatori) solo per la memoria dinamica, la gestione delle eccezioni.

Il linguaggio Java ha caratteristiche di dinamicità, in quanto i moduli di una applicazione vengono collegati fra loro solo a tempo di esecuzione, con possibilità di cambiare alcuni moduli o inserire nuovi moduli senza dover ricompilare i moduli rimanenti.

Per poter essere utilizzato nell'ambito delle reti informatiche, il linguaggio Java possiede spiccate caratteristiche di sicurezza. Infatti quando un programma viene eseguito, anche con l'utilizzo di classi che vengono prelevate da un altro nodo della rete, entra in esecuzione un *bytecode verifier* per controllare che non vi sia codice che viola la sicurezza del sistema, ossia che non vengano forzati i valori dei riferimenti, che vengano rispettati i diritti di accesso agli oggetti, che non vi siano accessi non autorizzati alle risorse locali.

Infine, Java è un linguaggio *multithread*, nel senso che per ogni processo eseguito da un nodo vi possono essere più flussi di esecuzione paralleli. I *thread* sono *processi leggeri* che condividono lo stesso spazio di memoria, in modo da non avere troppo *overhead* per il cambio di contesto.

Rispetto al linguaggio C++, in Java non si ritrovano alcuni costrutti linguistici, spesso introdotti per rendere il C++ compatibile col C: le funzionalità derivanti da tali costrutti si possono comunque ottenere in Java (indirettamente) utilizzando altri costrutti disponibili, come vedremo nei capitoli successivi.

### 1.3. Paradigmi di programmazione

L'evoluzione dei linguaggi di programmazione è stata caratterizzata da

una continua tendenza verso un'astrazione sempre maggiore, al fine di permettere al programmatore da un lato di prescindere dai dettagli implementativi dei linguaggi, e dall'altro di esprimere nella maniera più naturale possibile le caratteristiche rilevanti della realtà da rappresentare.

I linguaggi attuali consentono di realizzare diversi livelli di astrazione, che danno luogo a differenti paradigmi di programmazione: si può avere una programmazione *procedurale*, una programmazione basata sull'*astrazione sui dati* e una programmazione orientata agli *oggetti*.

La programmazione procedurale, che è stata la prima ad essere utilizzata, consiste nel definire un'astrazione per una sequenza di azioni che realizzano un compito unitario. Per esempio, per il calcolo della radice quadrata si definisce un'apposita funzione *sqrt()*, che viene adoperata ogni volta che si debba eseguire questa operazione.

La programmazione basata sull'astrazione sui dati consiste nel definire un tipo di dato, invece che mediante la sua struttura interna, in termini di modalità di utilizzo, ossia specificando, oltre ai valori possibili, le operazioni che sono significative per quel tipo (mediante *funzioni membro*). Per esempio, per trattare pile di interi, si definisce un tipo di dato *pila\_int*, che prevede le operazioni di immissione (*push()*) e di estrazione (*pop()*).

Infine, la programmazione orientata agli oggetti estende quella basata sui tipi di dato, introducendo i concetti di ereditarietà e di polimorfismo. Con l'ereditarietà, i tipi non sono considerati indipendenti tra loro, ma vengono correlati mediante relazioni del genere *tipo-sottotipo*. Si ottengono in tal modo delle gerarchie, con un tipo base e tanti sottotipi (di vario livello): un oggetto appartenente a un sottotipo appartiene anche al tipo base (e non viceversa). Con il polimorfismo, all'interno di una gerarchia, mediante un unico comando, si può invocare una funzione membro la cui intestazione è la stessa per tutti i sottotipi, ma la cui definizione dipende dal particolare sottotipo: la determinazione della funzione membro effettivamente invocata avviene a tempo di esecuzione in dipendenza dal particolare sottotipo coinvolto. Per esempio, se consideriamo la geometria del piano, si può definire una gerarchia, con un tipo base *poligono*, dei sottotipi *triangolo* e *rettangolo* derivati dal tipo *poligono*, e un sottotipo *quadrato* derivato dal tipo *rettangolo*. In tal modo, un quadrato è anche un rettangolo e un poligono, mentre un poligono non è un rettangolo o un quadrato (in termini linguistici, una variabile di tipo poligono può assumere anche valori di tipo triangolo, rettangolo o

quadrato). L'area dei poligoni può essere calcolata con funzioni membro che hanno tutte la stessa intestazione, la cui definizione varia in relazione al particolare poligono. Con l'utilizzo di una variabile di tipo *poligono*, il comando per il calcolo dell'area può essere unico per tutta la gerarchia (esso consiste nella individuazione, tramite la variabile, della funzione membro che calcola l'area), ed essere di volta in volta utilizzato dopo aver assegnato a tale variabile il valore che individua uno specifico poligono: la funzione che viene di volta in volta eseguita non è sempre la stessa, ma dipende dal particolare oggetto individuato in quel momento dalla variabile, ossia dal particolare poligono attuale.

Il linguaggio Java, fondato sulle classi e sulle loro gerarchie, è particolarmente orientato alla programmazione a oggetti. Tuttavia, come caso particolare, si può realizzare anche la programmazione basata sull'astrazione sui dati, e con semplici artifici, perfino la più semplice programmazione procedurale.

### **1.3.1. Programmazione basata sull'astrazione sui dati**

Volendo realizzare programmi basati semplicemente sull'astrazione sui dati, si utilizzano classi singole e non gerarchie. Una classe viene utilizzata per rappresentare un tipo di dato, con i possibili valori contenuti nelle variabili membro definite nella classe, e con le possibili operazioni rappresentate dalle funzioni membro definite nella classe stessa. Le classi possono essere istanziate, dando luogo a oggetti di quella classe, ognuno avente la propria copia di variabili membro e la possibilità di utilizzare le funzioni membro. Gli oggetti vengono allocati nella cosiddetta *memoria dinamica*.

### **1.3.2. Programmazione procedurale e ambiente statico**

Come sarà chiarito nel Capitolo 9, per ottenere la programmazione procedurale si utilizza il cosiddetto *ambiente statico* di una classe, che consiste in:

- variabili membro statiche, che esistono nella classe senza che sia richiesta l'esistenza di alcun oggetto (sono comuni a tutti gli eventuali oggetti di quella classe);

- funzioni membro statiche, che possono essere eseguite senza che sia richiesta l'esistenza di alcun oggetto classe e che utilizzano l'ambiente statico della classe (sono indipendenti dal particolare oggetto di quella classe al quale si applicano);
- classi membro statiche, corrispondenti a tipi definiti nella classe (esterna), indipendentemente dallo specifico oggetto della classe (esterna) stessa.

Come caso particolarmente semplice, i) un programma può essere costituito da una sola classe (con la funzione *main()*), e ii) la classe può possedere solamente l'ambiente statico (questo è possibile, in quanto la funzione *main()* deve essere anch'essa necessariamente statica). In tal caso ci si riconduce alla semplice programmazione procedurale (o funzionale), che può essere utilmente adoperata per illustrare i concetti fondamentali della programmazione: infatti, nella classe si hanno funzioni che possono essere richiamate da qualunque altra funzione, in particolare dalla funzione *main()*, e variabili che sono condivise da tutte le funzioni.

## 1.4. Sviluppo di un programma

Nel descrivere le fasi di sviluppo di un programma Java, facciamo riferimento al sistema operativo Windows e al “prompt dei comandi” (non facciamo uso di nessun ambiente di programmazione per concentrare l'attenzione del lettore sui costrutti linguistici che man mano vengono introdotti). Supponiamo altresì che un programma sia localizzato su un unico computer e che sia compilato ed eseguito da un'unica JVM.

Le classi che costituiscono un programma possono essere memorizzate su un unico file, ovvero su file separati racchiusi in una o più cartelle: al momento facciamo riferimento al caso tipico (abbastanza diffuso) in cui ogni classe viene memorizzata in un proprio file, e tutti i file si trovano in una medesima cartella. I nomi dei file e delle classi che essi memorizzano sono fra loro legati, e nel caso tipico di una classe per file, ogni file ha lo stesso nome della rispettiva classe (ed estensione *.java*). Una classe, quella

da cui comincia il programma e che riferisce a catena tutte le altre classi, costituisce la *classe principale* e le altre le *classi secondarie*.

Supponiamo anzitutto che i file di un programma siano contenuti nella cartella che costituisce il direttorio corrente (dal quale vengono emessi i comandi di compilazione e di esecuzione).

Ogni file che memorizza una classe è detto *file sorgente*. Il compilatore si invoca con il comando *javac*, specificando (nel caso più semplice) il nome del file contenente la classe principale. Per esempio, se la classe principale è memorizzata nel file *Prog.java*, la compilazione si ottiene con il comando:

```
javac Prog.java
```

Il compilatore produce, nella stessa cartella in cui sono contenuti i file sorgente, tanti file quante sono le classi che costituiscono il programma: ciascun file (detto file *oggetto*) ha il nome di una classe ed estensione *class* e contiene il byte-code della classe stessa. (Se nel comando di compilazione viene specificato un file diverso da quello contenente la classe principale, viene compilata la classe in esso contenuta e quelle eventualmente riferite).

Per eseguire (interpretare) un programma, occorre attivare la JVM con il comando *java*, specificando il nome della classe principale: l'interprete la individua prendendo il contenuto del file oggetto contenente la traduzione della classe stessa, file che ha il nome della classe ed estensione *class*. Con riferimento all'esempio precedente, il comando di esecuzione è il seguente:

```
java Prog
```

Supponiamo adesso che i file di un programma siano contenuti in cartelle diverse da quella corrente. Nel comando di compilazione occorre indicare il percorso (assoluto o relativo rispetto al direttorio corrente) e il nome con estensione del file contenente la classe principale. La ricerca dei file contenenti le classi secondarie viene effettuata o nel direttorio corrente o nel direttorio specificato con l'eventuale opzione *classpath* (preceduta dal carattere '-') che specifica un determinato percorso (assoluto o relativo rispetto al direttorio corrente). Nel comando di esecuzione occorre specificare il nome della classe principale, e la ricerca dei file contenenti le

varie classi viene effettuata o nel direttorio corrente o nel direttorio specificato con l'eventuale opzione *classpath*.

Per esempio, se i file di un programma (sia *Prog* il nome del file contenente la classe principale) sono memorizzati in un'unica cartella *es*, contenuta nella cartella *esempio*, e quest'ultima costituisce il direttorio corrente, i comandi di compilazione e di esecuzione che fanno uso di percorsi relativi sono i seguenti:

```
javac -classpath es es\Prog.java  
java -classpath es Prog
```

Nell'ipotesi che la cartella *esempio* si trovi nella radice dell'unità *C*, i comandi di compilazione e di esecuzione che fanno uso di percorsi assoluti sono i seguenti:

```
javac -classpath C:\esempio\es C:\esempio\es\Prog.java  
java -classpath C:\esempio\es Prog
```

Se non tutti i file di un programma sono memorizzati nella stessa cartella, occorre utilizzare differenti opzioni *classpath* per indicare differenti percorsi, con un opportuno carattere di separazione (',' in Windows).

## 1.5. Inclusione di package

I programmi Java possono utilizzare classi facenti parte di un package, tipicamente dei package che costituiscono le Java API, attraverso la loro *importazione*. È possibile importare una classe di un package oppure tutte le classi del package stesso (in realtà, solo quelle *pubbliche*), rispettivamente, con le istruzioni:

```
import nome-package.nome-classe;  
import nome-package.*;
```

Le classi importate possono essere utilizzate nel programma, attraverso i loro nomi semplici (senza menzionare il package a cui appartengono).

I package relativi alle Java API hanno un nome composto, la cui prima parte è costituita dal nome *java* e dal carattere '.'. Le classi del package



*java.lang* vengono importate per default: ogni altra classe appartenente a un altro package va importata esplicitamente.

Per esempio, se un programma è costituito da una classe *Prog*, memorizzata nel file *Prog.java*, che utilizza le classi del package *java.util*, occorre inserire in testa al file *Prog.java* l'istruzione:

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

I comandi di compilazione e di esecuzione vengono emessi prescindendo da tale importazione.

## 1.6. Codifica *unicode*

I caratteri *unicode*, codificati con 16 bit, sono tali per cui i primi 128 caratteri (codificati con i numeri 0-127, aventi al più 7 bit significativi) rappresentano i caratteri *ascii*, e i primi 256 caratteri (codificati con i numeri 0-255, aventi al più 8 bit significativi) rappresentano i caratteri *latin-1* (ovviamente, per uno stesso carattere, il numero di bit è diverso nella codifica *unicode* e nella codifica *ascii* o *latin-1*).

Un carattere *unicode* può anche essere espresso con la sua codifica (fatta di soli caratteri *ascii*), che consiste nella sequenza `\u` seguita da quattro cifre esadecimali (per esempio, `\u03a9`).

In Java gli identificatori, i commenti, i letterali carattere e i letterali stringa utilizzano caratteri *unicode*, eventualmente espressi con la loro codifica. Gli altri elementi lessicali sono composti di soli caratteri *ascii*. Quando un programma viene compilato, i caratteri non *ascii* vengono sostituiti con la loro codifica, in modo che possano essere utilizzati strumenti di sviluppo basati sull'utilizzo di soli caratteri *ascii*.

## 1.7. Dichiarazioni e definizioni

In Java non vengono effettuate distinzioni fra dichiarazioni e

definizioni. Tuttavia, in questo testo useremo abitualmente il termine definizione per i tipi, per le variabili, e per le funzioni con intestazione e corpo. Invece, useremo il termine dichiarazione unicamente nei casi in cui vengono specificate solo le intestazioni delle funzioni e non il loro corpo.

## 1.8. Convenzioni sui nomi

Ovviamente, i nomi devono rispettare le relative regole grammaticali (Capitolo 2), e solo queste. Tuttavia, per convenzione, i nomi delle classi iniziano per lettera maiuscola, mentre i nomi dei membri delle classi per lettera minuscola. Quando si utilizza un nome composto, le singole componenti non vengono separate da caratteri particolari, ma quelle successive alla prima iniziano tutte con lettera maiuscola (esempio: *MioNome*).

## 1.9. Argomenti trattati

In questo volume, a parte i Capitoli 2 e 3, in cui viene presentata la grammatica del linguaggio e illustrati i concetti fondamentali su tipi, variabili ed espressioni, i Capitoli 4, 5, e 6 sono orientati verso la programmazione procedurale, i Capitoli 7, 8 e 9 verso la programmazione basata sull'astrazione sui dati, e i Capitoli 10 e 11 verso la programmazione orientata agli oggetti. I Capitoli 12 e 13 trattano il problema dell'ingresso/uscita, il Capitolo 14 descrive classi e metodi generici, e il Capitolo 15 affronta le problematiche connesse con la programmazione a thread.

## 2. Grammatica

### 2.1. Metalinguaggio

La struttura di un linguaggio di programmazione viene descritta da un'apposita *grammatica*, costituita da un insieme di regole (dette *produzioni*) scritte in un determinato *metalinguaggio*. Ogni produzione descrive una *categoria grammaticale*, in termini di altre categorie grammaticali e di *simboli terminali* (simboli del linguaggio che non vengono ulteriormente analizzati).

Il metalinguaggio utilizzato per descrivere la grammatica del linguaggio di programmazione Java deriva dal classico formalismo di Backus e Naur (*BNF*, *Backus-Naur Form*). Per la sua comprensione è necessario chiarire solo alcune convenzioni, e questo può essere fatto con il linguaggio naturale. Riferiamoci, come esempio, a un piccolissimo sottoinsieme dell'italiano scritto, per il quale definiamo la seguente grammatica:

```
frase
    soggetto lista-verbi .
soggetto
    articolo aggettivo|opt nome
articolo
    il
aggettivo
    mio
    tuo
nome
    one-of
    cane gatto cavallo
```

```

lista-verbi
  verbo
  verbo , lista-verbi
verbo
  one of
  mangia dorme cammina corre

```

Da queste regole si deduce che una *frase* del linguaggio è formata (obbligatoriamente) da un *soggetto* (categoria grammaticale) seguito da una *lista-verbi* (categoria grammaticale) e dal simbolo terminale ‘.’. Un *soggetto* è formato da un *articolo*, opzionalmente (simbolo di metalinguaggio **|opt**) da un *aggettivo* e da un *nome*. Un *articolo* è costituito dal simbolo terminale ‘*il*’. Un *aggettivo* e un *nome* possono assumere diverse forme alternative, ciascuna costituita da un simbolo terminale: le alternative possono essere scritte ciascuna su una riga separata, oppure elencate dopo il simbolo di metalinguaggio **one-of**. La categoria grammaticale *lista-verbi* è definita ricorsivamente: essa può essere costituita da un solo verbo, o da più verbi separati dal simbolo terminale ‘,’.

Con questa grammatica si possono comporre, tra l’altro, le seguenti frasi del linguaggio:

```

il mio cane mangia.
il gatto mangia, dorme.
il tuo cavallo cammina, corre, mangia.

```

Notare che in realtà una frase del linguaggio viene scritta a partire da un certo insieme di caratteri. Occorre pertanto definire ulteriori regole grammaticali che consentano di effettuare una preliminare analisi del testo (*analisi lessicale*), eliminando gli spazi e isolando le singole parole del linguaggio costituenti la frase (‘,’, ‘.’, ‘il’, ‘mio’, ‘tuo’, ...).

## 2.2. Analisi lessicale e analisi sintattica

Con riferimento a un qualunque linguaggio di programmazione, quindi, le regole grammaticali devono consentire di effettuare su un programma:

- una preliminare *analisi lessicale* che, a partire da una sequenza di caratteri, conduca alla individuazione delle *parole* (i caratteri costituiscono i simboli terminali);
- una susseguente *analisi sintattica* che, a partire dalla sequenza di parole ottenuta dall'analisi lessicale, consenta di esaminare i costrutti linguistici utilizzati (le parole costituiscono i simboli terminali).

Nel resto di questo capitolo saranno descritte le regole grammaticali relative alla struttura lessicale di Java, mentre nei capitoli seguenti verranno introdotte gradualmente le regole grammaticali relative alla sintassi del linguaggio. Verrà usata la terminologia inglese, per uniformità con la grammatica “ufficiale” di Java. Poiché tale grammatica è piuttosto complessa, per facilitare l'apprendimento dovremo discostarcene, usando forme semplificate e organizzando diversamente le categorie grammaticali.

Nel seguito di questo testo, verranno sempre omesse le regole grammaticali che definiscono una *sequenza* o una *lista* di un qualunque genere di elementi (regole analoghe a quella relativa alla *lista-verbi* precedente), che avranno la seguente forma:

```
some-element-seq
  some-element
  some-element some-element-seq
some-element-list
  some-element
  some-element , some-element-list
```

## 2.3. Analisi lessicale di Java

Java utilizza i *caratteri unicode* codificati con 16 bit. I primi 128 caratteri *unicode* coincidono con i caratteri *ascii*, e i primi 256 caratteri *unicode* con i caratteri *latin-1*. Le codifiche *unicode* dei caratteri *ascii* (oppure *latin-1*) hanno i 9 (oppure gli 8) bit più significativi uguali a 0, mentre i 7 (oppure 8) bit meno significativi hanno la configurazione

prevista dalla codifica *ascii* (oppure *latin-1*). I caratteri *ascii* e la loro codifica sono riportati in Appendice 1.

Nome	Abbreviazione	Sequenza di escape
nuova riga	NL (LF)	\n
tabulazione orizz.	HT	\t
spazio indietro	BS	\b
ritorno carrello	CR	\r
avanzam. modulo	FF	\f
segnale acustico	BEL	\a
barra invertita	\	\\
apice	'	\'
virgolette	"	\"

Tabella 2.1. Sequenze di *escape simboliche*

La codifica *unicode* di un carattere può essere espressa da una *sequenza di escape unicode*, costituita dai caratteri ‘\u’ seguiti da quattro cifre esadecimali (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) che rappresentano un numero esadecimale da 0 a FFFF (65535 in base dieci). La codifica *unicode* di un carattere *latin-1* può anche essere espressa da una *sequenza di escape ottale*, costituita dal carattere ‘\’ seguito da cifre ottali (0-7) che rappresentino un numero (ottale) inferiore a 377 (255 in base dieci). Alcuni caratteri di controllo e alcuni caratteri speciali *unicode* possono essere rappresentati da *sequenze di escape simboliche* (Tab. 2.1), ciascuna costituita dal carattere ‘\’ seguito da uno dei nove caratteri elencati nella tabella stessa.

Un programma Java subisce anzitutto un’analisi lessicale, che parte da una sequenza di caratteri e giunge ad una sequenza di parole (*token*). I token costituiscono i simboli terminali per la successiva analisi sintattica. L’analisi lessicale avviene mediante le seguenti tre fasi:

- 1) vengono eliminate le sequenze di *escape unicode* e sostituite con i corrispondenti caratteri *unicode* (una sequenza di caratteri che è potenzialmente una sequenza di *escape*

*unicode*, ma che è preceduta da un ulteriore carattere ‘\’ viene trattata come una sequenza di caratteri *unicode*);

- 2) il programma viene trasformato in sequenze di linee, eliminando da queste i *terminatori di linea*;
- 3) in base a una certa grammatica, ogni linea viene trasformata in una sequenza di *elementi*: di questi vengono eliminati gli *spazi bianchi* e i *commenti*, lasciando soltanto i cosiddetti *token*, e i *token* vengono quindi classificati.

Le regole grammaticali che consentono di individuare gli elementi e di classificare i *token* sono riportate di seguito.

### 2.3.1. Elementi lessicali

Un programma Java è costituito da una sequenza caratteri (che sono simboli terminali): questi vengono raggruppati in *elementi* (*input-element*), e un elemento può essere uno *spazio bianco* (*white-space*), un commento (*comment*) o una *parola* (*token*):

*input-element*  
*white-space*  
*comment*  
*token*

Uno spazio bianco può essere uno fra tre particolari caratteri *ascii* o un terminatore di linea:

*white-space*  
**Ascii-Space**  
**Ascii-HorizontalTab**  
**Ascii-FormFeed**  
*line-terminator*

Un terminatore di linea è costituito da uno dei caratteri *ascii* CR (Carriage Return) o LF (Line Feed), o dalla loro sequenza:

*line-terminator*  
**Ascii-CarriageReturn**  
**Ascii-LineFeed**

***Ascii-CarriageReturn Ascii-LineFeed***

Un commento può essere costituito da una sequenza di caratteri racchiusa tra ‘/\*’ e ‘\*/’ (commento tradizionale), oppure tra ‘//’ e un terminatore di linea (commento di fine linea):

*comment*  
*traditional-comment*  
*end-of-line-comment*

Un token è una specifica sequenza di caratteri, avente caratteristiche grammaticali che consentono di individuarne l’inizio e la fine (che non sono necessariamente spazi bianchi o commenti, ma che possono essere la fine o l’inizio di un altro token). Un token può essere:

*token*  
*identifier*  
*keyword*  
*literal*  
*separator*  
*operator*

I vari token verranno descritti singolarmente nei paragrafi successivi.

## **2.4. Identificatori**

Un identificatore è una parola (diversa da una parola chiave o da un letterale booleano o dal letterale nullo, vedi paragrafi successivi) formata da una *lettera* seguita da una sequenza di caratteri appositi, ossia:

*identifier*  
*letter*  
*letter identifier-char-seq*  
*identifier-char*  
*letter*  
*digit*



Una lettera Java è una lettera *ascii* minuscola o maiuscola (a-z, ossia \u0061-\u007a, e A-Z, ossia \u0041-\u005a), una lettera di qualunque altro alfabeto del mondo, o uno dei caratteri ‘\_’ (\u005f) e ‘\$’ (\u0014). Una cifra Java è una cifra *ascii* (0-9, ossia \u0030-\u0039) o una cifra di qualunque altro alfabeto del mondo.

Un identificatore viene utilizzato per individuare un’entità definita nel programma, e può essere liberamente scelto dal programmatore. Esempi di identificatori sono:

x          x1          alfa45    \_xx

**Osservazione.** In realtà, un’entità viene in genere individuata da un *nome*, che può coincidere con un *identificatore* come caso particolare, ma può anche avere una forma più complessa.

## 2.5. Parole chiave

Le parole chiave (*keyword*) sono un insieme di simboli, ciascuno dei quali è costituito da una parola inglese (formata da una sequenza di lettere), il cui significato è stabilito dal linguaggio e non può essere modificato dal programmatore:

*keyword*  
**one-of**  
**abstract boolean ... volatile while**

Le parole chiave più utilizzate saranno esaminate quando verranno trattati i costrutti grammaticali che le utilizzano.

## 2.6. Letterali

I *letterali* servono a denotare dei valori costanti indicati dal

programmatore, e possono essere classificati nel seguente modo:

```
literal
  integer-literal
  floating-point-literal
  boolean-literal
  character-literal
  string-literal
  null-literal
```

A differenza degli identificatori, che rappresentano nomi, i letterali rappresentano *operandi* definiti in fase di scrittura del programma, e vengono elaborati dal programma stesso.

### 2.6.1. Letterale intero

Un letterale *intero* viene rappresentato da una o più cifre decimali, ottali o esadecimali, a seconda della base di numerazione prescelta. Più precisamente:

- ) numero in base dieci: cifra ‘0’ oppure sequenza di cifre scelte nell’intervallo ‘0’-‘9’ che comincia con una cifra diversa da ‘0’;
- ) numero in base otto: sequenza di cifre scelte nell’intervallo ‘0’-‘7’ che comincia con ‘0’;
- ) numero in base sedici: sequenza di cifre scelte negli intervalli ‘0’-‘9’ e ‘A’-‘F’ (oppure ‘a’-‘f’) che inizia con i caratteri ‘0x’ (o ‘0X’).

Per esempio, sono letterali interi in base dieci, otto e sedici, rispettivamente:

0	45	473	5980
00	045	0473	05310
0x0	0x45	0x473	0x3AF9

**Osservazione.** Un letterale intero può essere di tipo *int* o di tipo *long*: in quest’ultimo caso deve avere il suffisso ‘l’ oppure ‘L’. Poiché i numeri interi sono rappresentati in complemento a 2, un letterale intero di tipo *int* (eventualmente preceduto dall’operatore unario ‘+’) può avere un valore (in base dieci) che va da +0 a  $+2^{31}-1$  (+1147483647), oppure, se preceduto dall’operatore unario ‘-’, da -1 a  $-2^{31}$  (-1147483648). Analogamente, un letterale intero di tipo *long* (eventualmente preceduto dall’operatore unario ‘+’) può avere un valore (in base dieci) che va da +0 a  $+2^{63}-1$ , oppure, se preceduto dall’operatore unario ‘-’, da -1 a  $-2^{63}$ .

### 2.6.2. Letterale reale

Un letterale *reale* (o letterale *floating-point*) viene rappresentato sempre in base dieci, con una parte in *virgola fissa* e un *esponente* preceduto dalla lettera ‘e’ oppure ‘E’. La parte in virgola fissa è costituita a sua volta da una parte intera e una parte frazionaria separate dal punto decimale, mentre l’esponente è un numero intero.

**Osservazione.** Un letterale reale può essere di tipo *float* (suffisso ‘f’ oppure ‘F’) o di tipo *double* (suffisso ‘d’ oppure ‘D’, ovvero nessun suffisso). Un letterale positivo di tipo *float* può avere un valore che va da  $+1.40139846e^{-45}$  a  $+3.40181347e^{+38}$ , e un letterale positivo di tipo *double* può avere un valore che va da  $+4.94065645841146544e^{-314}$  a  $+1.79769313486131570e^{+308}$ . Durante la compilazione, un letterale positivo con valore più piccolo di quello di valore minimo produce un errore per un non numero (*NaN*: Not a Number), mentre un letterale positivo di valore più grande del massimo produce un errore per un numero infinito (vedi lo standard IEEE 754).

Alcune componenti di un numero reale possono anche mancare, ma deve essere comunque presente almeno una cifra (nella parte intera o in quella frazionaria) e uno fra punto decimale, esponente o suffisso. Per esempio, sono numeri reali:

113.0	1.13e1	113e0	
.13	1.	0.0	115d

### 2.6.3. Letterale booleano

Un letterale *booleano* è costituito da uno dei valori *false* o *true*.

### 2.6.4. Letterale carattere

Un letterale *carattere* può essere costituito da un carattere (diverso da ‘” e da ‘\’) racchiuso fra apici, da una sequenza di *escape unicode* (escluse \u000A e \u000D) racchiusa fra apici o da una sequenza di *escape ottale* o *simbolica* racchiusa tra apici. Per esempio, sono letterali carattere:

'a'        '\u0031'   '\67'        '\n'

Un letterale carattere viene sempre a corrispondere (a livello di linguaggio oggetto) ad un codice *unicode*.

**Osservazione.** Un letterale carattere può anche contenere il carattere “”, ma deve essere scritto ricorrendo a una sequenza di *escape* (per esempio, scrivendo “\”, dove dopo il carattere barra invertita compare due volte il carattere apice, e non il carattere doppio apice); analogamente, il letterale carattere costituito dal carattere ‘\’ può essere scritto come “\\”. Il letterale *nuova riga* non può essere scritto utilizzando una sequenza di *escape unicode* (“\u000A”), perché tale sequenza verrebbe eliminata nelle fasi 1) e 2) (paragrafo 2.3), ma può essere scritto ricorrendo a una sequenza di *escape simbolica* (“\n”). Per la stessa ragione, il letterale *ritorno carrello* non deve essere scritto ricorrendo sequenza di *escape unicode* (“\u000D”), ma può essere scritto ricorrendo a una sequenza di *escape simbolica* (“\r”).

### 2.6.5. Letterale stringa

Un letterale *stringa* è costituito da zero o più caratteri (esclusi il carattere “” e il carattere ‘\’) o sequenze di *escape* (escluse le sequenze di *escape unicode* \u0000A e \u000D), contenuti in un’unica linea e racchiusi tra doppi apici.

**Osservazione.** I caratteri “” e ‘\’ possono essere inseriti in un letterale stringa scrivendo, rispettivamente, le sequenze di *escape simboliche* “\” e “\\”. I caratteri di controllo *nuova riga* e *ritorno carrello*, per considerazioni analoghe a quelle fatte per i letterali carattere, non possono essere inseriti in un letterale stringa mediante le loro sequenze di *escape unicode*, ma possono essere espressi con le sequenze di *escape simboliche* “\n” e “\r”.

Per esempio, sono letterali stringa:

""        "a"        "abc"        "a\"u003b"        "\\n"

### 2.6.6. Letterale nullo

Il letterale nullo è costituito dal valore *null*.

## 2.7. Separatori

Un separatore è uno dei seguenti caratteri *ascii*:

*separator*

**one of**

(   )   {   }   [   ]   ;   ,   .

I separatori costituiscono segni di interpunzione nella scrittura di un programma.

## 2.8. Operatori

Alcuni caratteri speciali o loro combinazioni sono usati come *operatori* (il loro numero è 37, di cui 26 sono effettivi e 11 rappresentano forme abbreviate di assegnamento), e servono a denotare operazioni nel calcolo delle espressioni. Il loro elenco è il seguente:

*operator*

**one of**

+   -   \*   /   %   &   |   ^

=   >   <   !   ~   ?   :

==   <=   >=   !=   &&   //   ++   --

<<   >>   >>>

+=   -=   \*=   /=   &=   /=   ^=   %=   <<=   >>=   >>>=

Ogni operatore è caratterizzato da alcune proprietà, che permettono di interpretare in modo univoco il significato di una espressione. Queste proprietà sono:

- la *posizione* rispetto ai suoi operandi (o *argomenti*) (un operatore si dice *prefisso* se precede gli operandi, *postfisso* se li segue, *infixo*

negli altri casi (gli operatori ‘++’ e ‘--’ possono essere sia prefissi che postfissi);

- il *numero di operandi* (o arietà);
- la *precedenza* (o priorità) nell'ordine di esecuzione (gli operatori con priorità più alta vengono eseguiti per primi: per esempio, l'espressione  $1+1*3$ , dove '+' rappresenta l'addizione e '\*' la moltiplicazione, viene calcolata come  $1+(1*3)$ , poiché l'operatore di moltiplicazione in Java ha una priorità maggiore dell'operatore di addizione);
- l'*associatività*, cioè l'ordine in cui vengono eseguiti operatori aventi la stessa priorità (gli operatori associativi a sinistra vengono eseguiti da sinistra a destra, come nell'espressione  $6/3/1$ , che viene calcolata come  $(6/3)/1$ ; quelli associativi a destra vengono eseguiti da destra a sinistra).

La posizione rispetto agli operandi è la seguente:

- prefissi:	++ -- + - ! ~
- postfixi:	++ --
- infissi:	tutti gli altri

Il numero di operandi è il seguente

- un operando (operatori unari): + - ++ -- ! ~
- tre operandi (operatore ternario): ? :
- due operandi (operatori binari): tutti gli altri

La precedenza e l'associatività sono le seguenti:

*Precedenza decrescente*

- unari postfixi: ++, --
- unari prefissi: ! ~ ++ -- + (unario) - (unario)
- moltiplicativi: \* / %
- additivi: + -
- di traslazione: << >> >>>
- di relazione: < <= > >=
- di uguaglianza: == !=
- and: &
- or esclusivo: ^

### Associatività

nessuna  
da destra a sinistra  
da sinistra a destra  
da sinistra a destra  
da sinistra a destra  
da sinistra a destra  
da sinistra a destra  
da sinistra a destra

- or:	da sinistra a destra
- and-condizionale: &&	da sinistra a destra
- or-condizionale:	da sinistra a destra
- condizionale: ? :	da sinistra a destra
- assegnamento: = += -= ...	da destra a sinistra

Gli operatori verranno descritti più in dettaglio nel Capitolo 3.

## 2.9. Uso delle spaziature

Tenendo conto di quanto detto nel paragrafo 2.3, nello scrivere un programma, occorre fare in modo che possano essere individuati come token gli identificatori. Un identificatore può essere adiacente a un separatore o a un operatore, in quanto la distinzione fra queste entità viene effettuata tenendo conto dei possibili caratteri utilizzati. Al contrario, un identificatore non può essere adiacente a una parola chiave, ma le due entità vanno separati con spazi bianchi o commenti (*spaziature*). Consideriamo, per esempio, il seguente caso (*int* è una parola chiave e *num* un identificatore):

```
int num;
```

Se *int* e *num* non fossero separati, non sarebbe possibile distinguerli, ma formerebbero un unico identificatore *intnum*.

Notare che nessun costrutto del linguaggio prevede che un identificatore possa essere adiacente a un letterale, in particolare a un letterale booleano o al letterale nullo, escludendo in tal modo la necessità di inserire spaziature.

In tutte le situazioni diverse dal caso di adiacenza fra un identificatore e una parola chiave, le spaziature non sono necessarie, ma possono essere usate per migliorare la leggibilità dei programmi. Nel seguente esempio mostriamo alcune versioni equivalenti di un frammento di programma, con qualche giudizio sul grado di leggibilità (ignoriamo il significato dell'esempio, osservando solo l'aspetto lessicale):

```
if(n+1>0){n=n-1;m=3*n;}           // poche spaziature

if                                // troppe spaziature
( n + 1 > 0 )
{
    n = n - 1;
    m = 3 * n ; }

if (n + 1 > 0)                    // buono
{
    n = n - 1;
    m = 3 * n;
}
```

Notare che nel testo, per motivi di impaginazione, non sempre sarà possibile utilizzare uno stile simile all'ultimo indicato.



## 3. Tipi, variabili, espressioni

### 3.1. Tipi

Il linguaggio Java prevede una tipizzazione stretta (è un linguaggio *strongly typed*): costanti, variabili ed espressioni hanno tutte un tipo noto a tempo di compilazione. Un tipo è costituito da un insieme di valori e da un gruppo di operazioni su quei valori. I tipi Java sono i seguenti:

```
type
  primitive-type
  derived-type
primitive-type
  boolean
  byte
  short
  int
  long
  char
  float
  double
derived-type
  enumeration-type
  array-type
  class-type
  interface-type
```

I tipi primitivi possono essere raggruppati nel seguente modo (gli interi sono rappresentati in complemento a 2, i reali sono rappresentati in forma

esponenziale,  $\text{mantissa} * 2^{\text{esponente}}$ , dove *mantissa* è un numero frazionario della forma  $\pm 0.\text{CC} \dots \text{C}$  (C sta per cifra), ed *esponente* un numero intero):

```

boolean
tipi numerici
  tipi "integral"
    byte (interi su 8 bit)
    short (interi su 16 bit)
    int (interi su 32 bit)
    long (interi su 64 bit)
    char (caratteri unicode o naturali su 16 bit)
  tipi "floating"
    float (reali su 32 bit: 24 di mantissa, 8 di esponente)
    double (reali su 64 bit: 53 di mantissa, 11 di esponente)

```

I tipi derivati (detti anche tipi *riferimento*) verranno esaminati a partire dal Capitolo 6.

### 3.1.1. Tipo *boolean*

Il tipo *boolean* (booleano) ha come insieme di valori le due costanti simboliche *false* e *true*.

Con elementi di questo tipo si possono fare operazioni logiche, che sono:

	OR logico ( <i>disgiunzione</i> )
^	OR-ESCLUSIVO logico o XOR
&	AND logico ( <i>congiunzione</i> )
!	NOT logico ( <i>negazione</i> )

Il loro risultato, anch'esso di tipo logico, è mostrato in tabella 3.1.

Possono essere anche utilizzati gli operatori OR condizionale (*'/'*) e AND condizionale (*'&&'*), che valutano inizialmente solo il primo operando, e se questo vale *true* o *false*, rispettivamente, producono subito il risultato, che vale quanto il primo operando, altrimenti valutano anche il secondo operando.

Con i booleani si possono usare anche gli operatori di uguaglianza:

== uguale  
!= diverso

Essi producono un risultato booleano, secondo le regole convenzionali.

p	q	p   q	p ^ q	p & q	! p
false	false	false	false	false	true
false	true	true	true	false	true
true	false	true	true	false	false
true	true	true	false	true	false

Tabella 3.1. Operatori logici

### 3.1.2. Tipi “integral”

Un tipo intero ha per valori tutti i numeri interi compresi fra  $-2^{N-1}$  e  $+2^{N-1}-1$ , dove  $N$  è il numero di bit usati per la sua rappresentazione.

Le due operazioni elementari sugli interi sono il *più unario* (‘+’, che lascia invariato il segno) e il *meno unario* (‘-’ che cambia il segno). Altre operazioni comuni sono quelle aritmetiche, realizzate mediante operatori aritmetici binari, con operandi e risultato di tipo intero:

- sottrazione  
\* moltiplicazione  
/ quoziente della divisione  
% resto della divisione

Il quoziente della divisione fra interi viene arrotondato all’intero di valore assoluto minore: per esempio, il quoziente fra 2 e 3 è 0. Il resto della divisione è un intero che ha per segno quello del dividendo. È quindi vero che  $(a/b)*b + a\%b$  è uguale ad  $a$ , con  $a$  e  $b$  interi e  $b$  diverso da 0.

Altre operazioni sugli interi sono l’incremento e il decremento (con ovvio significato):

++ incremento  
-- decremento

Inoltre, con gli interi possono essere effettuate operazioni bit a bit, che richiedono peraltro delle conoscenze sulle rappresentazioni dei numeri interi (un'operazione di traslazione produce lo spostamento di ogni bit del primo operando da una posizione a un'altra, verso destra o verso sinistra, e la distanza fra le due posizioni è un numero intero che costituisce il secondo operando (questo vale 1 nel caso di posizioni adiacenti)):

~	complemento (unario) bit a bit
&	AND bit a bit
	OR bit a bit
^	OR ESCLUSIVO bit a bit
<<	traslazione a sinistra della distanza specificata inserendo bit uguali a 0
>>	traslazione a destra della distanza specificata replicando il segno (bit più significativo)
>>>	traslazione a destra della distanza specificata inserendo bit uguali a 0.

Il risultato di tutti i precedenti operatori (tranne incremento e decremento, il cui risultato è dello stesso tipo dell'operando) è di tipo *int* se gli operandi sono di tipo *byte*, *short*, *int*, *char*, di tipo *long* se gli operandi sono di tipo *long*.

Infine, agli interi si possono anche applicare gli operatori di uguaglianza e gli operatori di relazione (insieme costituiscono gli operatori di confronto), che producono un risultato di tipo booleano:

==	uguale
!=	diverso
>	maggiore
>=	maggiore o uguale
<	minore
<=	minore o uguale

I tipi "integral" comprendono anche il tipo *char*. Esso è costituito dai numeri naturali compresi fra 0 e  $2^{16}-1$ , che rappresentano le codifiche unicode dei caratteri. Sui naturali possono essere fatte tutte le operazioni valide per gli interi, con la precisazione che il risultato di un'operazione aritmetica binaria o bit a bit (salvo il caso in cui il risultato stesso sia

compreso fra 0 e  $2^{16}-1$  e gli operandi siano espressioni costanti, paragrafo 3.3) non è di tipo *char*, ma di tipo *int* o *long*.

### 3.1.3. Tipi “floating”

I tipi reali hanno come insieme di valori un sottoinsieme dei numeri reali, ossia quelli rappresentabili all'interno dell'elaboratore con un formato prefissato (in realtà, un sottoinsieme dei numeri razionali).

Per i reali sono definite le seguenti operazioni:

- $+$ ,  $-$     più unario e meno unario
- $++$ ,  $--$    incremento e decremento
- $+$ ,  $-$     somma e sottrazione
- $*$ ,  $/$     moltiplicazione e divisione reale
- $\%$        resto  $r$  della divisione reale fra  $n$  e  $d$ :  
 $r = n - (d*q)$   
 con  $q$  intero, il cui segno è quello di  $n/d$  e il cui valore assoluto è il più grande che non supera il valore assoluto di  $n/d$

Con i reali si possono effettuare tutte le operazioni di confronto, che producono come risultato un valore booleano.

## 3.2. Definizioni di variabili

Le variabili di un tipo primitivo si definiscono in accordo alla seguente sintassi:

```
variable-definition
    type-identifier variable-specifier-list ;
variable-specifier
    variable-identifier initializer[opt]
initializer
    = expression
...
```

Pertanto, la definizione di una variabile richiede l'indicazione dell'identificatore del tipo, la specifica dell'identificatore della variabile ed eventualmente del suo valore iniziale, che nel caso più comune consiste in un'espressione preceduta da '=' (esiste anche un'altra modalità per specificare un valore iniziale, relativa agli array (Capitolo 6)). Notare che con una stessa definizione si possono definire più variabili dello stesso tipo, e che ogni definizione (di una o più variabili) termina con ';'.

A titolo di esempio, effettuiamo le seguenti definizioni:

```
int n;
double dd = 10.0;
boolean b, bb = true;
char cc = '\t';
```

L'identificatore utilizzato corrisponde all'indirizzo della variabile (per semplicità, *indirizzo della variabile* e *variabile* si considerano sinonimi). Il valore della variabile viene modificato ogni volta che si esegue un *assegnamento* (operatore '='), compreso quello utilizzato per l'inizializzazione (Fig. 3.1).



Figura 3.1. Variabile di un tipo primitivo

Una definizione di variabile può comparire ovunque nel programma.

### 3.2.1. Variabili *final* e costanti

Una variabile può essere definita con il modificatore *final*, e il suo valore, una volta stabilito, non può essere più modificato:

```
final boolean b = true;
int m;
final int n;
...
n = m + 10;
n = 20;                // errore
```

Una costante è una variabile *final* con valore iniziale specificato nella sua definizione mediante un'espressione costante (sottoparagrafo 3.3.1), come nei seguenti esempi:

```
final int n = 10;  
final int m = n*3;
```

### 3.3. Espressioni

Un'espressione è composta da operandi e operatori, e viene calcolata eseguendo in sequenza una serie di operazioni. Per ogni operatore, vengono prima valutati gli operandi e quindi applicato l'operatore stesso. Gli operandi sono in genere altre espressioni, e possono ridursi a letterali o a variabili (in quest'ultimo caso viene semplicemente considerato il loro valore), e il risultato prodotto dall'operatore è sempre un valore. Alcuni operatori richiedono esplicitamente che un operando sia una variabile, in quanto hanno un effetto collaterale che consiste nello scrivere un nuovo valore nella variabile stessa (vedi l'operazione di assegnamento).

Come accennato nel Capitolo 2, l'ordine in cui gli operatori vengono valutati in un'espressione dipende dalla precedenza e associatività. L'utilizzo dei delimitatori "parentesi tonde" individua sottoespressioni che vengono considerate operandi e che quindi vengono calcolate per prime.

Gli operatori che possono essere utilizzati con i tipi primitivi sono stati visti nel paragrafo 3.1, nel quale però non sono stati esaminati l'operatore ternario, gli operatori di assegnamento, e non sono stati trattati a fondo gli operatori di incremento e decremento: tali operatori saranno esaminati tra breve in paragrafi appositi.

#### 3.3.1. Espressioni costanti

Un'espressione è costante se gli operandi sono soltanto letterali o costanti (anche gli operatori utilizzabili sono soggetti ad alcune limitazioni).

### 3.3.2. Esempi

**Esempio 1** ( $x$  è una variabile reale)

`x / 3 * 2`

Gli operatori ‘/’ e ‘\*’ hanno la stessa precedenza, per cui l'ordine di esecuzione viene determinato dall'associatività: gli operatori aritmetici binari sono associativi a sinistra, e l'espressione viene calcolata come se fosse scritta:

`(x / 3) * 2`

**Esempio 2** ( $x$  e  $y$  sono due variabili reali)

`x != 0 && y/x > 0.5`

Il secondo operando, contenente la divisione  $y/x$ , non viene calcolato se il primo operando è falso, ossia se  $x$  vale 0.

**Esempio 3** ( $x$  è una variabile reale)

`2 < x && x <= 4`

Questa espressione corrisponde alla formula matematica  $2 < x \leq 4$ .  
Notare che la semplice espressione:

`2 < x <= 4`

è sintatticamente sbagliata: essa viene calcolata come  $(2 < x) \leq 4$ , dove l'espressione fra parentesi produce un risultato di tipo booleano, che non può essere confrontato con un intero.

## 3.4. Operatore condizionale

L'operatore condizionale ha la forma:



$op1 \text{ ? } op2 : op3$

Esso richiede che il primo operando  $op1$  sia di tipo booleano: se questo vale *true* il risultato è il valore dell'operando  $op2$ , altrimenti il valore dell'operando  $op3$ .

### 3.5. Operatori di assegnamento

Come detto nel paragrafo 3.3, un'espressione produce sempre come risultato un valore. In una espressione può essere usato l'operatore di assegnamento '=', che richiede due operandi dello stesso tipo, quello sinistro costituito da una variabile e quello destro da un'espressione, e produce a) come risultato il valore dell'espressione che compare alla sua destra e b) come effetto collaterale l'assegnamento di questo valore alla variabile che compare alla sua sinistra. Per esempio, l'espressione ( $x$  è una variabile intera):

**$x = 3 + 2$**

produce come risultato 5 e come effetto collaterale l'assegnamento di 5 alla variabile  $x$ .

L'espressione che compare a destra dell'operatore di assegnamento può a sua volta contenere un operatore di assegnamento, come nel seguente esempio ( $x$  e  $y$  sono due variabili intere):

**$x = (y = 3 + 2)$**

(in base alle regole di precedenza e associatività, le parentesi non sono necessarie). L'assegnamento che viene eseguito per primo ( **$y = 3 + 2$** ) produce come risultato il valore 5, che viene poi assegnato a  $x$  (il risultato prodotto da quest'ultimo assegnamento non viene ulteriormente considerato)

Notare che le seguenti espressione di assegnamento ( $c$  e  $cc$  sono due variabili di tipo *char*):

```
c = cc + 1
c = ~ cc
```

non sono corrette, in quanto l'operando destro è di tipo *int* e non di tipo *char* come l'operando sinistro (vedi il sottoparagrafo 3.1.2), mentre l'espressione:

```
c = cc++
```

è corretta, in quanto l'operatore di incremento non è né binario né bit a bit.

Quando una stessa variabile compare alla sinistra e alla destra dell'operatore di assegnamento, si può utilizzare una forma compressa di assegnamento (ossia un altro operatore di assegnamento), come nel seguente caso:

```
- operatore +=   a = a+b   si può scrivere come  a += b
- operatore -=   a = a-b   si può scrivere come  a -= b
...
```

### 3.6. Incremento e decremento

Gli operatori (unari) '++' e '--' richiedono come operando una variabile: essi producono come risultato il valore aggiornato della variabile se prefissi, o il valore originario della variabile se postfissi, e come effetto collaterale l'incremento o il decremento della variabile stessa. Pertanto, si hanno le seguenti equivalenze (fra espressioni):

```
x = x+1           ++x
x = x-1           --x
```

Notare che tutti gli operatori unari prefissi, compresi l'incremento e il decremento prefissi, sono associativi a destra, per cui l'espressione:

```
++--x
```

è corretta (gli operatori prodotti dall'analisi lessicale sono '+' e '--': prima viene eseguito il decremento, poi il più unario), mentre l'espressione:

**+++x**

è errata (gli operatori prodotti dall'analisi lessicale sono '++' e '+': prima viene eseguito il più unario, che produce come risultato un valore, poi l'incremento, che però richiede una variabile).

### 3.7. Altri operatori

Nel linguaggio Java sono definiti ulteriori operatori, che non rientrano nella categoria di quelli enucleati in fase di analisi lessicale. Il loro comportamento dipende dal costrutto linguistico in cui compaiono, e il risultato prodotto può non essere un valore. Tali operatori sono i seguenti (essi saranno meglio descritti contestualmente al loro utilizzo):

Operatore	Uso	Descrizione
[]	tipo[]	definisce un tipo array, che contiene elementi di tipo <i>tipo</i>
	tipo[op]	crea un oggetto array con <i>op</i> elementi (operatore <i>new</i> )
	array[op]	referisce l'elemento di indice <i>op</i> dell'oggetto array <i>array</i>
.	oggetto.membro	referisce il membro <i>membro</i> dell'oggetto <i>oggetto</i>
(parametri)	metodo(parametri)	definisce o chiama il metodo <i>metodo</i> con parametri <i>parametri</i>
(tipo)	(tipo) op	converte ( <i>cast</i> ) <i>op</i> al tipo <i>tipo</i>
new	new costruttore	crea un nuovo oggetto
	new tipo[op]	crea un nuovo oggetto array di <i>op</i> elementi di tipo <i>tipo</i>
instanceof	op1 instanceof op2	restituisce <i>true</i> se <i>op1</i> è un'istanza di <i>op2</i>

Gli operatori '[ ]', '.' e '(parametri)' hanno la stessa precedenza di quelli unari postfissi. Gli operatori '(tipo)' e 'new' hanno precedenza compresa fra quella degli operatori unari prefissi e quella degli operatori moltiplicativi. L'operatore 'instanceof' ha la stessa precedenza degli operatori di relazione.

Gli operatori precedenti sono tutti associativi a destra, tranne l'operatore 'instanceof'.

Occorre notare che la selezione di un elemento di un array e la chiamata di una funzione (operatori 'array[op]' e '(parametri)') hanno la precedenza massima (insieme a '++' e '--'), per cui nelle espressioni vengono valutati per primi.

### 3.8. Conversione di tipo

Una conversione di tipo consente:

- nel caso di un operatore aritmetico binario, di trasformare il tipo di un operando nel tipo dell'altro operando;
- nel caso dell'operatore di assegnamento, di trasformare il tipo del risultato di una espressione nel tipo della variabile a cui il nuovo valore deve essere assegnato.

Una conversione di tipo può essere automatica (o implicita) se determinata dal compilatore, oppure esplicita se indicata dal programmatore.

La conversione automatica (*widening conversion*) avviene:

- nell'ambito dei tipi "integral" e "floating", rispettivamente, verso il tipo che usa un maggior numero di bit per la rappresentazione, con nessuna perdita di informazione;
- da un tipo "integral" a un tipo "floating" (non viceversa), con nessuna perdita sull'ordine di grandezza, ma una eventuale perdita di precisione.

Il grafico delle conversioni automatiche è riportato in Fig. 3.2, dove le linee tratteggiate indicano una eventuale perdita di precisione.

Le conversioni esplicite (*cast*) possono essere effettuate tra tutti i tipi numerici (escluso quindi il tipo booleano), e richiedono che l'indicazione del nuovo tipo venga indicata tra parentesi tonde. Come esempi, possiamo scrivere le seguenti definizioni ed espressioni:

```
char c = 'a';  
int i = 10;  
double d = 10.2;  
...  
d = i  
i = (int)d  
i = c  
c = (char)i
```

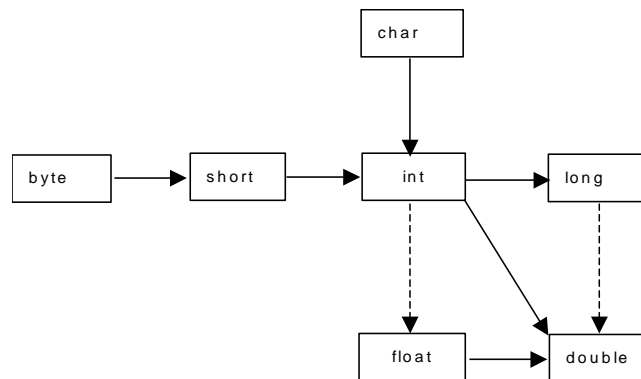


Figura 3.2. Conversioni automatiche di tipo

### 3.9. Classe *Math*

La classe *Math*, che fa parte del package *java.lang*, contiene come membri statici (Capitolo 9) alcune costanti e diverse funzioni matematiche di uso comune. Ogni membro ha un tipo (sia per quanto riguarda le costanti che il risultato delle funzioni) e viene individuato dall'utente con un nome costituito da un *identificatore qualificato* (identificatore della classe e identificatore del membro separati dal carattere '.'), e per le funzioni membro, anche con la specifica, fra parentesi tonde, della

eventuale lista di parametri formali (Capitolo 5). I membri più comunemente utilizzati sono i seguenti:

<b><i>double Math.PI</i></b>	costante <i>pigreco</i>
<b><i>double Math.E</i></b>	costante <i>e</i>
<b><i>int Math.abs ( int n )</i></b>	valore assoluto di <i>n</i>
<b><i>double Math.abs ( double x )</i></b>	valore assoluto di <i>x</i>
<b><i>double Math.sqrt ( double x )</i></b>	radice quadrata di <i>x</i>
<b><i>double Math.pow ( double x , double a )</i></b>	<i>x</i> elevato ad <i>a</i>
<b><i>double Math.sin ( double x )</i></b>	seno di <i>x</i>
...	
<b><i>double Math.asin ( double a )</i></b>	arco seno di <i>a</i>
...	
<b><i>double Math.exp ( double x )</i></b>	<i>e</i> elevato ad <i>x</i>
<b><i>double Math.log ( double x )</i></b>	logaritmo in base <i>e</i> di <i>x</i>
<b><i>double Math.random ()</i></b>	numero casuale <i>r</i> , $0.0 \leq r < 1.0$
<b><i>int Math.round ( float f )</i></b>	arrotondamento di <i>f</i> all'intero più vicino

Utilizzando i membri della classe *Math* si può scrivere la seguente espressione (*x*, *y* e *z* sono variabili reali):

```
x = Math.PI + Math.pow(y, z)
```

## 4. Istruzioni e programma

### 4.1. Struttura di un semplice programma

Come visto nel Capitolo 1, un programma Java è costituito da una classe principale e da eventuali classi secondarie (anche *enumerazioni* o *interfacce*): nel caso più semplice, ciascuna di queste entità è memorizzata in un file avente lo stesso nome della entità stessa, e tutti i file risiedono in una cartella che costituisce il direttorio corrente. La classe principale deve contenere la funzione *main()*, che viene eseguita per prima e che costituisce quindi il programma principale (sia la classe che la funzione *main()* vengono definiti *public*, e la funzione *main()* anche *static*, per motivazione che saranno chiarite nel Capitolo 9). Nel caso di una sola classe, un programma ha quindi la seguente forma:

```
// file Programma.java
public class Programma
{ public static void main(String[] args)
  { definizioni e istruzioni
  }
}
```

Le definizioni e le istruzioni specificano, rispettivamente, quali sono le variabili utilizzate e quali sono le azioni che devono essere effettuate sulle variabili stesse per ottenere il risultato: definizioni e istruzioni vengono esaminate in sequenza, dalla parentesi graffa aperta della funzione *main()* fino o alla parentesi graffa chiusa della stessa funzione, o al comando

*System.exit()* (corrispondente a una chiamata di funzione con un argomento attuale intero, che per convenzione vale 0 se il programma termina correttamente).

La funzione *main()* ha la stessa struttura di tutte le funzioni (Capitolo 5): essa ha un parametro formale costituito da un array di stringhe (Capitolo 6), e il corrispondente argomento attuale, che può essere anche un array vuoto (costituito da 0 stringhe) viene specificato con il comando di esecuzione. Per esempio, si consideri il programma:

```
// file Programma.java
public class Programma
{ public static void main(String[] args)
  { // ...
    // scrittura di args[0] e args[1];
    // ...
  }
}
```

Con i comandi:

```
javac Programma.java
java Programma uno due
```

viene tradotto ed eseguito il programma. L'argomento attuale della funzione *main()* è l'array di stringhe avente due elementi, espressi mediante i due letterali stringa *uno* e *due*: l'esecuzione produce pertanto la stampa delle stringhe suddette.

Nel seguito, l'argomento attuale della funzione *main()* sarà di norma un array vuoto.

## 4.2. Classe *Console*

Un programma deve poter effettuare operazioni di ingresso/uscita. Per motivi di chiarezza espositiva, non si è ritenuto opportuno esaminare a questo punto il meccanismo di gestione dell'ingresso/uscita in Java, ma si è pensato di definire allo scopo una classe apposita, che abbiamo chiamato *Console*, con funzioni (membri statici, Capitolo 9) in grado di effettuare



letture da tastiera e scritture su video di valori appartenenti ai principali tipi primitivi (tipi *boolean*, *char*, *int*, *double*) e al tipo stringa (classe *String*, Capitolo 6).

Le funzioni della classe *Console*, che l'utente può utilizzare mediante un identificatore qualificato comprendente l'identificatore della classe, effettuano le cosiddette "letture e scritture formattate", nel senso che trasformano sequenze di caratteri in rappresentazioni interne alla macchina (per esempio, in numeri in base due) e viceversa. Per ogni funzione, viene specificato il tipo del risultato (parola chiave *void*: nessun risultato) e, fra parentesi tonde, l'eventuale parametro formale.

Più in dettaglio, le funzioni previste dalla classe *Console* sono le seguenti:

#### **Letture/scritture di singoli caratteri (qualsivoglia):**

```
char Console.leggiUnCarattere ()  
void Console.scriviUnCarattere ( char c )
```

#### **Letture di singoli dati:**

```
boolean Console.leggiBooleano ()  
int Console.leggiIntero ()  
double Console.leggiReale ()  
char Console.leggiCarattere ()  
String Console.leggiStringa ()
```

I dati possono essere scritti sulla stessa linea o su linee diverse, purché separati da spazi bianchi (che non vengono letti). I numeri positivi vanno scritti senza il segno +. I caratteri e le stringhe vanno scritti senza delimitatori (non come letterali).

Notare che la funzione *Console.leggiUnCarattere()* legge il carattere successivo, qualunque esso sia, mentre la funzione *Console.leggiCarattere()* salta eventuali spazi bianchi fino a incontrare il primo carattere che non sia uno spazio bianco.

#### **Scrittura di un dato e posizionamento all'inizio di una nuova linea:**

```
void Console.scriviBooleano ( boolean b )  
void Console.scriviIntero ( int i )  
void Console.scriviReale ( double d )  
void Console.scriviCarattere ( char c )
```

```
void Console.scriviStringa ( String s )
```

**Scrittura di un dato più il carattere spazio (sulla stessa linea):**

```
void Console.scriviBool ( boolean b )  
void Console.scriviInt ( int i )  
void Console.scriviReal ( double d )  
void Console.scriviCar ( char c )  
void Console.scriviStr ( String s )
```

I dati vengono effettivamente trasferiti su video ogni volta che avviene una scrittura che comporta il posizionamento all'inizio di una nuova linea.

**Posizionamento (senza scrittura di dati) all'inizio di una nuova linea:**

```
void Console.nuovaLinea ()
```

La definizione completa della classe *Console* è riportata nel Capitolo 12.

Alcune espressioni che utilizzano le funzioni precedenti sono (*a* e *b* sono variabili intere):

```
a = Console.leggiIntero()  
Console.scriviStr("Risultato:")  
Console.scriviIntero(b)
```

La classe è memorizzata nel file *Console.java*, che va inserito nella stessa cartella che contiene gli altri file che costituiscono il programma. Tale file può essere prelevato all'indirizzo Internet:

*<http://www2.ing.unipi.it/LibroJava>*

## 4.3. Blocchi

Il corpo della funzione *main()* (come quello di una qualunque altra funzione) è un *blocco*: esso rappresenta la forma più generale di istruzione,

ed è a sua volta costituito da definizioni locali e da istruzioni (anche nessuna) racchiuse fra parentesi graffe:

```

block
    { block-component-seqopt }
block-component
    local-definition
    instruction
local-definition
    local-variable-definition
    local-class-or-interface-definition
local-variable-definition
    finalopt variable-definition
local-class-or-interface-definition
    finalopt class-or-interface-definition

```

Le definizioni di classi e di interfacce saranno viste nei Capitoli 8, 9 e 10.

## 4.4. Istruzioni

Le istruzioni vengono classificate nel seguente modo:

```

instruction
    ;
    expression-instruction
    conditional-instruction
    repetitive-instruction
    jump-instruction
    throw-instruction
    try-catch-instruction
    synchronized-instruction
    identifier : instruction
    block

```

La forma più semplice di istruzione è il carattere ‘;’: in questo caso si ha un’istruzione vuota, che non specifica alcuna azione. La forma più completa di istruzione è il blocco, che consente di trasformare più definizioni locali e istruzioni in un’unica istruzione.

Fra le rimanenti istruzioni, tratteremo nei paragrafo successivi quelle espressione, quelle condizionali, quelle ripetitive, quelle di salto e quelle con identificatore. Le istruzioni *throw* e *try-catch* saranno trattate nel Capitolo 11 (eccezioni) e l'istruzione *synchronized* nel Capitolo 15 (thread).

## 4.5. Istruzioni espressione

Le istruzioni espressione consistono nell'aggiungere il carattere ';' a particolari espressioni, secondo le seguenti regole:

```
expression-instruction
    instruction-expression ;
instruction-expression
    assignment-expression
    increment/decrement-expression
    method-invocation
    class-object-creation
```

L'espressione può essere un assegnamento oppure un incremento/decremento, ma anche una chiamata di funzioni (*method-invocation*), o una creazione di oggetto classe (Capitolo 7). Per esempio, l'utilizzo delle funzioni della classe *Math* o della classe *Console* rientrano nella categoria delle istruzioni espressione: l'eventuale risultato, se non esplicitamente memorizzato con un assegnamento, viene perduto.

Un esempio di blocco con istruzioni espressione è il seguente:

```
{                                // inizio blocco
    int a, b, c, d; double v;    // definizioni locali
    ...
    a = (b = c + d);             // istruzioni espressione
    a++;
    v = Math.PI;
    Console.scriviReale(v);
}
```

Notare che le istruzioni di incremento/decremento sono equivalenti a istruzioni di assegnamento:

```

n++;    equivalente a    n = n+1;
++n;    equivalente a    n = n+1;
n--;    equivalente a    n = n-1;
--n;    equivalente a    n = n-1;

```

Utilizzando gli operatori di assegnamento e di incremento/decremento, si hanno altre equivalenze fra istruzioni, per esempio le seguenti:

```

m = n++; equivalente a    m = n; n = n+1;
m = ++n; equivalente a    n = n+1; m = n;
m = n--; equivalente a    m = n; n = n-1;
m = --n; equivalente a    n = n-1; m = n;

```

#### 4.5.1. Esempi

Utilizzando istruzioni espressione, possiamo scrivere questo semplice programma riferito alla classe *Console*:

```

// file InOut.java
public class InOut
{ public static void main (String[] args)
  { boolean bb; int ii; double dd; char cc;
    // ...
    bb = Console.leggiBooleano();
    ii = Console.leggiIntero();
    dd = Console.leggiReale();
    cc = Console.leggiCarattere();
    // ...
    Console.scriviBool(bb);
    Console.scriviInt(ii);
    Console.scriviReale(dd);
    Console.scriviCar(cc);
    Console.scriviStr("Ciao");
    Console.nuovaLinea();
    // ...
  }
}

```

Una tipica esecuzione è la seguente (i dati battuti sulla tastiera sono scritti in corsivo):

```

false -130 +13.5e2 a
false -130 1350.0
a Ciao

```

Come altro esempio, possiamo scrivere un programma per sommare due periodi di tempo (espressi in *giorni*, *ore*, *minuti* e *secondi*):

```

// file Sommap.java
public class Sommap
{ public static void main(String[] args)
  { int giornia, orea, minutia, secondia,
      giornib, oreb, minutib, secondib,
      giornir, orer, minutir, secondir,
      riporto, totale;
    Console.scriviStringa
      ("Scrivi il primo periodo (g h m s):");
    giornia = Console.leggiIntero();
    orea = Console.leggiIntero();
    minutia = Console.leggiIntero();
    secondia = Console.leggiIntero();
    Console.scriviStringa
      ("Scrivi il secondo periodo (g h m s):");
    giornib = Console.leggiIntero();
    oreb = Console.leggiIntero();
    minutib = Console.leggiIntero();
    secondib = Console.leggiIntero();
    totale = secondia + secondib;
    secondir = totale % 60; riporto = totale / 60;
    totale = minutia + minutib + riporto;
    minutir = totale % 60; riporto = totale / 60;
    totale = orea + oreb + riporto;
    orer = totale % 24; riporto = totale / 24;
    giornir = giornia + giornib + riporto;
    Console.scriviStringa("Risultato:\t");
    Console.scriviInt (giornir);
    Console.scriviInt (orer);
    Console.scriviInt (minutir);
    Console.scriviInt (secondir);
    Console.nuovaLinea();
  }
}

```

Un semplice esempio di esecuzione è il seguente:

```
Scrivi il primo periodo (g h m s):  
10 15 30 20  
Scrivi il secondo periodo (g h m s):  
250 22 40 50  
Risultato: 261 14 11 10
```

## 4.6. Istruzioni condizionali

Le istruzioni condizionali comprendono l'istruzione *if* e l'istruzione *switch*.

### 4.6.1. Istruzione *if*

Questa istruzione ha la seguente forma:

```
if-instruction  
if ( if-condition ) instruction  
if ( if-condition ) instruction else instruction
```

La condizione è un'espressione che deve produrre un risultato di tipo booleano (notare che non è possibile nessuna conversione di tipo fra intero e booleano, né implicita né indicata esplicitamente dal programmatore). La parte *else* può anche mancare. Se la condizione è vera, viene eseguita l'istruzione che segue *if*, altrimenti quella che segue *else* (se presente).

Un esempio di istruzione *if* è il seguente:

```
...  
boolean b = ... ; int a = ... ;  
if (b) then a++; else a--;  
...
```

In caso di annidamento, in accordo alla precedente regola sintattica, la parte *else* si riferisce allo *if* più vicino. Per esempio, nel seguente caso:

```
if (a>0) if (a<10) alfa = 0; else alfa = 1000;
```

la parte *else* si riferisce allo *if* relativo alla condizione  $a < 10$ . Se vogliamo fare diversamente, si deve usare un blocco:

```
if (a>0) {if (a<10) alfa = 0;} else alfa = 1000;
```

Come esempio di programma, riferiamoci al calcolo delle radici  $r_1$  ed  $r_2$  di un'equazione di secondo grado con coefficienti reali  $a$ ,  $b$  e  $c$ :

```
// file Radici.java
public class Radici
{ public static void main(String[] args)
  { double a, b, c, delta;
    a = Console.leggiReale();
    b = Console.leggiReale();
    c = Console.leggiReale();
    if (a==0 && b==0) Console.scriviStringa
                        ("Equazione degenera");

    else if (a==0)
    { Console.scriviStringa
      ("Equazione di primo grado");
      Console.scriviReale(-c/b);
    }
    else
    { delta = b*b - 4*a*c;
      if (delta<0) Console.scriviStringa
                  ("Determinante negativo");

      else
      { delta = Math.sqrt(delta);
        Console.scriviStringa
          ("Equazione di secondo grado");
        Console.scriviReal((-b+delta)/(2*a));
        Console.scriviReal((-b-delta)/(2*a));
        Console.nuovaLinea();
      }
    }
  }
}
```

Un esempio di esecuzione è il seguente:



1 -3 2  
**Equazione di secondo grado**  
 2.0 1.0

#### 4.6.2. Istruzione *switch*

Questa istruzione ha la seguente forma (semplificata):

```
switch-instruction
  switch ( switch-condition )
    { normal-alternative-seq default-alternative|opt }
normal-alternative
  case-label-seq instruction-seq
case-label
  case constant-expression :
default-alternative
  default : instruction-seq
```

La condizione deve essere una espressione di tipo *byte*, *char*, *short* o *int*, ed ogni espressione costante (*constant-espression*), valutata a tempo di compilazione, deve produrre un possibile valore della condizione: alcuni o tutti i valori sono raggruppati nelle alternative normali. L'alternativa *default* può anche mancare, ma, se presente, deve comparire per ultima. In esecuzione viene selezionata quella alternativa che corrisponde al valore prodotto dalla condizione: se tale valore non è previsto da qualche alternativa normale, viene selezionata l'alternativa *default*, se presente, altrimenti l'esecuzione dell'istruzione *switch* termina.

La sequenza di istruzioni contenute nell'alternativa selezionata viene quindi eseguita: l'ultima istruzione della sequenza deve produrre la terminazione dell'istruzione *switch*. Tale terminazione può essere ottenuta con un'istruzione *break*, meglio illustrata nel paragrafo 4.8. L'ultima alternativa, a prescindere dalla presenza dell'istruzione *break*, provoca comunque la terminazione dell'istruzione *switch*.

Un esempio di istruzione *switch* è il seguente:

```
...
int giorno = ... ;
int ore_lavorate = 0;
switch (giorno)
```

```

{ case 1: case 2: case 3: case 4: case 5:
    orelavorate +=8; break;
  case 6: case 7: break;
  default: Console.scriviStringa("Giorno non valido");
}
...

```

Come esempio di programma, riferiamoci al calcolo della data successiva rispetto alla data specificata (una data è espressa con tre interi, *giorno*, *mese* e *anno*). Nel programma viene calcolata la durata del mese in esame, compreso il caso del mese di febbraio: per verificare se un anno è bisestile si applicano le regole del calendario Gregoriano (valido per il periodo compreso tra il 15 Ottobre 1582 e l'anno 4317): un anno è bisestile se è un multiplo di 4 e non un secolo, oppure un secolo multiplo di 400. Il programma è il seguente:

```

// file DataSucc.java
public class DataSucc
{ public static void main(String[] args)
  { int giorno, mese, anno, giornimese;
    giorno = Console.leggiIntero();
    mese = Console.leggiIntero();
    anno = Console.leggiIntero();
    switch (mese)
    { case 1: case 3: case 5: case 7:
      case 8: case 10: case 12:
        giornimese = 31; break;
      case 4: case 6: case 9: case 11:
        giornimese = 30; break;
      case 2:
        if (anno % 4 == 0 && anno % 100 != 0
            || anno % 400 == 0)
          giornimese = 29;
        else giornimese = 28; break;
      default:
        Console.scriviStringa("Mese inesistente");
        giornimese = 0; System.exit(1);
    }
    if (giorno > giornimese)
      Console.scriviStringa("Giorno inesistente");
    else
      { if (giorno < giornimese) giorno++;

```

```
        else {giorno=1; if (mese<12) mese++;  
                else { mese=1; anno++; } }  
        Console.scriviInt(giorno);  
        Console.scriviInt(mese); Console.scriviInt(anno);  
        Console.nuovaLinea();  
    }  
}
```

Un esempio di esecuzione è il seguente:

```
28 2 2020  
29 2 2020
```

## 4.7. Istruzioni ripetitive

Le istruzioni ripetitive comprendono l'istruzione *while*, l'istruzione *do-while* e l'istruzione *for*.

### 4.7.1. Istruzione *while*

Questa istruzione ha la seguente forma:

```
while-instruction  
while ( while-condition ) instruction
```

La condizione è un'espressione che deve produrre un risultato di tipo booleano. Se la condizione è vera, viene eseguita l'istruzione che costituisce il corpo del costrutto e l'istruzione *while* viene ripetuta, altrimenti l'istruzione *while* termina. Se la condizione non è vera in partenza, il corpo del costrutto non viene mai eseguito.

Nel seguente esempio, l'istruzione *while* è utilizzata per esaminare i caratteri battuti a tastiera, scorrendo e contando i caratteri spazio, fino a leggere il primo carattere non spazio:

```

...
int cont = 0; char c;
c = Console.leggiUnCarattere();
while (c == ' ')
{ cont++; c = Console.leggiUnCarattere(); }
...

```

Come esempio di programma, riferiamoci al caso del calcolo del massimo comun divisore *mcd* tra due interi (maggiori di 0) *alfa* e *beta* utilizzando l'algoritmo di Euclide (il numero maggiore viene decrementato di una quantità pari al numero minore, e così via, fintanto che i due numeri si mantengono diversi; quando divengono uguali ognuno di essi rappresenta il massimo comun divisore):

```

// file MasComDiv.java
public class MasComDiv
{ public static void main(String[] args)
  { int alfa, beta, mcd;
    alfa = Console.leggiIntero();
    beta = Console.leggiIntero();
    while (alfa != beta)
      if (alfa > beta)
        alfa -= beta; else beta -= alfa;
    mcd = alfa;
    Console.scriviIntero(mcd);
  }
}

```

Un esempio di esecuzione è il seguente:

```

21 56
7

```

L'algoritmo di Euclide può essere realizzato in maniera più efficiente, calcolando ciclicamente il resto della divisione intera fra gli operandi:

```

// file MasComDiv1.java
public class MasComDiv1
{ public static void main(String[] args)
  { int alfa, beta, resto, mcd;
    alfa = Console.leggiIntero();

```

```
        beta = Console.leggiIntero();
        while (beta != 0)
        { resto = alfa % beta;
          alfa = beta; beta = resto;
        }
        mcd = alfa;
        Console.scriviIntero(mcd);
    }
}
```

Notare che in questo caso i numeri possono anche valere 0: se solo uno dei due vale 0, il massimo comun divisore è l'altro numero (0 può essere diviso per qualunque numero), mentre se entrambi valgono 0 il massimo comun divisore è 0.

Un altro esempio d'uso dell'istruzione *while* si ha nel calcolo delle cifre binarie di un intero positivo e della loro trasformazione in caratteri. Le cifre 0 e 1 (interi) vengono trasformate nei caratteri '0' e '1' sommandoci la codifica del carattere '0' (notare che in qualunque codifica i caratteri che rappresentano le cifre sono codificati in modo crescente, a partire dalla codifica del carattere '0'). Per semplicità, le cifre binarie vengono calcolate in ordine inverso (dalla meno significativa alla più significativa):

```
// file CalcolaCifre.java
class CalcolaCifre
{ public static void main(String[] args)
  { char c;
    final int base = 2;
    int cifra, numero;
    numero = Console.leggiIntero();
    while(numero !=0)
    { cifra = numero % base;
      numero = numero / base;
      c = (char)(cifra + '0');
      Console.scriviUnCarattere(c);
    }
    Console.nuovaLinea();
  }
}
```

Un esempio di esecuzione è il seguente (ricordare che le cifre binarie sono generate in ordine inverso):

```
18  
01001
```

#### 4.7.2. Istruzione *do-while*

Questa istruzione ha la seguente forma:

```
do-instruction  
do istruzione while ( do-condition ) ;
```

La condizione è un'espressione che deve produrre un risultato di tipo booleano: questa viene valutata dopo che è stata eseguita l'istruzione che costituisce il corpo del costrutto, e, se vera, l'istruzione *do-while* viene ripetuta (il corpo del costrutto viene eseguito almeno una volta).

Nel seguente esempio, analogo a quello del sottoparagrafo precedente, l'istruzione *do-while* è utilizzata per esaminare i caratteri battuti a tastiera, scorrendo e contando i caratteri spazio, fino a leggere il primo carattere non spazio:

```
...  
cont = -1; char c;  
do  
{ cin.get(c) ;  
  cont++;  
}  
while (c == ' ');  
...
```

La differenza rispetto alla versione che utilizza l'istruzione *while* sta nel fatto che viene contato anche il carattere non spazio che causa la terminazione del ciclo, per cui il contatore viene inizializzato col valore -1.

Come esempio di programma, ricaviamo un intero (positivo) partendo da una sequenza di caratteri '0' e '1' che rappresentano cifre binarie. I caratteri vengono trasformati negli interi 0 e 1 sottraendoci la codifica del carattere '0' (come detto nel sottoparagrafo precedente, la codifica dei

caratteri corrispondenti alle cifre è ordinata). Supponiamo che i caratteri che costituiscono il numero siano preceduti da eventuali caratteri spazio (nessun segno o altro tipo di carattere), e siano seguiti da un carattere che non corrisponda a una cifra binaria. Il programma è il seguente:

```
// file CalcolaInt.java
class CalcolaInt
{ public static void main(String[] args)
  { char c; final int base = 2;
    int risultato = 0;
    do
      c = Console leggiUnCarattere();
    while(c == ' ');
    do
      { risultato = risultato * base + (c - '0');
        c = Console leggiUnCarattere();
      }
    while (c >='0' && c <= '1');
    Console.scriviIntero(risultato);
  }
}
```

Un esempio di esecuzione è il seguente (alla fine c'è un carattere spazio):

```
    10010
18
```

### 4.7.3. Istruzione *for*

Questa istruzione ha la seguente forma:

```
for-instruction
  for ( for-init|opt ; for-condition|opt ; for-update|opt ) instruction
for-init
  instruction-expression-list
final|opt type variable-specifier-list
for-update
  instruction-expression-list
```

La condizione è un'espressione che deve produrre un risultato di tipo

booleano. Tipicamente, l'inizializzazione e l'aggiornamento non prevedono liste: l'inizializzazione è un'espressione di assegnamento che inizializza una variabile di controllo (eventualmente con definizione di tale variabile), e l'aggiornamento (o passo) è un'espressione che dà un nuovo valore alla variabile di controllo stessa (la incrementa o la decrementa). L'istruzione che costituisce il corpo del *for* può far uso della variabile di controllo. L'istruzione *for* viene eseguita come se fosse scritta nel seguente modo:

```
{  inizializzazione
    while (condizione)
    {  istruzione           // corpo del for
        aggiornamento;
    }
}
```

Un semplice esempio di utilizzo dell'istruzione *for* si ha nel calcolo del fattoriale di un numero intero positivo *num*:

```
// file Fatt.java
public class Fatt
{  public static void main(String[] args)
    {  int num, fattoriale = 1;
        num = Console.leggiIntero();
        for (int i = 2; i <= num; i++)
            fattoriale *= i;
        Console.scriviIntero(fattoriale);
    }
}
```

Un esempio di esecuzione è il seguente:

```
15
2004310016
```

Come successivo esempio, riportiamo il seguente programma, che legge un numero intero positivo in base dieci (compreso fra 0 e  $2^7-1$ ), calcola le 8 cifre binarie della sua rappresentazione interna (il bit più significativo è sempre 0), trasforma le cifre in caratteri e quindi li stampa:



```
// file NumeroBin.java
class NumeroBin
{ public static void main(String[] args)
  { int numero, cifra, n = (int)Math.pow(2, 7);
    char c;
    numero = Console.leggiIntero();
    for (int i = 7; i >=0; i--)
    { cifra = numero/n;
      c = (char)(cifra + '0');
      numero %= n; n /= 2;
      Console.scriviUnCarattere(c);
    }
    Console.nuovaLinea();
  }
}
```

Un esempio di esecuzione è il seguente:

```
67
01000011
```

Come altro esempio, riportiamo il seguente programma, che legge un numero intero positivo e stampa i numeri primi minori o uguali ad esso:

```
// file CalcolaPrimi.java
class CalcolaPrimi
{ public static void main(String[] args)
  { int numero, n, resto;
    numero = Console.leggiIntero();
    for (int i = numero; i > 1; i--)
    { n = i-1; resto = i%n;
      while((n>0) && (resto!=0))
      { n--;
        resto = i%n;
      }
      if (n==1) Console.scriviInt(i);
    }
    Console.nuovaLinea();
  }
}
```

Un esempio di esecuzione è il seguente:

23

23 19 17 13 11 7 5 3 2

Come esempio finale, riportiamo un programma che legge un numero intero positivo rappresentato in una data base *base*, compresa tra due e dieci, e riscrive la sua rappresentazione in ognuna di tali basi. In ingresso vengono richiesti, su due linee separate, la base del numero e il numero stesso (senza segno). La lettura del numero avviene carattere per carattere, saltando gli eventuali caratteri spazio di testa, e prelevando quindi caratteri fino a che rappresentano cifre della base data. Il numero letto viene anzitutto convertito in rappresentazione interna (base due). Per ogni valore della base, compreso fra due e dieci, viene poi calcolata la quantità *basen*, che rappresenta il peso della cifra più significativa del numero espresso in quella base. Il numero viene poi scritto (rappresentato nella base considerata) a partire dalla cifra più significativa. Il programma è il seguente:

```
// file TutteBasi.java
class TutteBasi
{ public static void main(String[] args)
  { int base, basen, n, ris, cifra;
    char c;
    Console.scriviStringa
      ("Scrivi la base del numero");
    base = Console.leggiIntero();
    Console.scriviStr ("Scrivi il numero in base ");
    Console.scriviIntero(base);
    do
      c = Console.leggiUnCarattere();
    while(c == ' ');
    ris = 0; cifra = c - '0';
    while (cifra >= 0 && cifra < base)
    { ris = ris * base + cifra;
      c = Console.leggiUnCarattere();
      cifra = c - '0';
    }
    for (base = 2; base <= 10; base++)
    { n = ris; basen = 1;
      while (n >= base)
      { n /= base; basen *= base; }
      n = ris;
```

```
        Console.scriviStr("Numero dato in base ");
        Console.scriviIntero(base);
        do
        { cifra = n/basen;
          n %= basen; basen /= base;

          c = (char)(cifra + '0');
          Console.scriviUnCarattere(c);
        }
        while (basen != 0);
        Console.nuovaLinea();
    }
}
```

Una tipica esecuzione produce le seguenti linee di scrittura sul video:

```
Scrivi la base del numero
8
Scrivi il numero in base 8
67
Numero dato in base 2
110111
Numero dato in base 3
2001
Numero dato in base 4
313
Numero dato in base 5
210
Numero dato in base 6
131
Numero dato in base 7
106
Numero dato in base 8
67
Numero dato in base 9
61
Numero dato in base 10
55
```

Nell'istruzione *for* la condizione può anche mancare, e in questo caso il ciclo diventa infinito. Inoltre, può mancare l'inizializzazione (la variabile

di controllo può avere già un valore significativo) e può essere omesso il passo (l'aggiornamento della variabile di controllo può avvenire nel corpo del *for*).

## 4.8. Istruzioni di salto

In accordo alla regola sintattica del paragrafo 4.4, le istruzioni possono avere una etichetta costituita da un identificatore seguito dal carattere ':'. Un'istruzione di salto può specificare o meno un identificatore, che si riferisce all'etichetta di un'altra istruzione.

```
while ( ... )  
{  
    ...  
    while ( ... )  
    {  
        ...  
        break;  
        ...  
    }  
    ...  
}
```

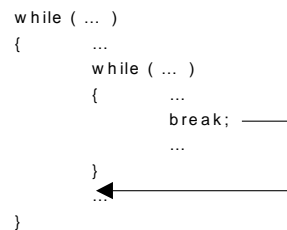


Figura 4.1. Istruzione *break* senza identificatore

```
esci: while ( ... )  
{  
    ...  
    while ( ... )  
    {  
        ...  
        break esci;  
        ...  
    }  
    ...  
}
```

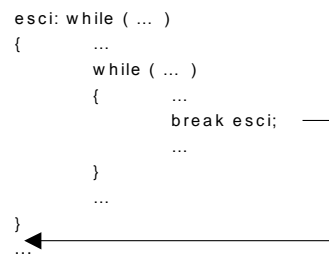


Figura 4.2. Istruzione *break* con identificatore

#### 4.8.1. Istruzione *break*

Questa istruzione ha la seguente forma:

*break-instruction*  
**break** *identifier**|opt* ;

Se l'identificatore è omissso, essa produce la terminazione del costrutto *switch*, *while*, *do-while* o *for* in cui compare (Fig. 4.1). Con specifica di identificatore, invece, essa produce la terminazione del costrutto esterno *switch*, *while*, *do-while* o *for* che ha l'etichetta corrispondente (Fig. 4.2).

#### 4.8.2. Istruzione *continue*

Questa istruzione ha la seguente forma:

*continue-instruction*  
**continue** *identifier**|opt* ;

Se l'identificatore è omissso, essa produce la terminazione della iterazione corrente del ciclo *while*, *do-while* o *for* in cui compare, con esecuzione della iterazione successiva. Con specifica di identificatore, invece, essa produce la terminazione della iterazione corrente del ciclo esterno *while*, *do-while* o *for* che ha l'etichetta corrispondente.

#### 4.8.3. Istruzione *return*

Questa istruzione ha la seguente forma:

*return-instruction*  
**return** *expression**|opt* ;

Essa produce la terminazione della funzione in cui compare. Se la funzione prevede la restituzione di un risultato, l'istruzione *return* specifica il valore di ritorno (mediante un'espressione). In una funzione possono esservi nessuna (se la funzione è di tipo *void*), una o più istruzioni *return*.

**Osservazione.** L'istruzione *return* può essere usata per ultima nelle alternative di

un'istruzione *switch*, in quanto la terminazione della funzione produce anche quella della istruzione *switch* stessa.

## 4.9. Regole di visibilità nei blocchi

In un blocco, un identificatore deve essere prima definito e poi utilizzato (non è necessario che tutte le definizioni precedano le istruzioni).

Un blocco, essendo un'istruzione, può essere annidato in un altro blocco, ovvero più blocchi possono essere paralleli. In presenza di più blocchi devono essere specificate le cosiddette regole di visibilità degli identificatori (o regole di *scopo* di un blocco), che stabiliscono quali sono le zone di un programma in cui un dato identificatore può essere o meno utilizzato. Le regole di visibilità sono le seguenti:

- un identificatore definito in un blocco è visibile dal punto della sua definizione fino alla fine del blocco; pertanto:
  - un identificatore definito in un blocco più esterno è visibile in un blocco più interno, mentre un identificatore definito in un blocco più interno non è visibile in un blocco più esterno;
  - un identificatore definito in un blocco non è visibile in un blocco parallelo.
- non è consentito ridefinire in un blocco più interno lo stesso identificatore definito in un blocco più esterno.

Per esempio, il seguente programma contiene istruzioni lecite e non lecite:

```
// file Programma.java
public class Programma
{ public static void main(String[] args)
  { // blocco A
    int i = 10;
    { // blocco B
      double i;
      // errore: i ridefinito
      int j = 2;
      int k = 3;
    }
  }
}
```

```

        i++;          // i e' definito solo nel blocco A
        // ...
    }
    { // blocco C
        int j = 3;     // j diversa da quella del blocco B
        k++;
        // errore: k non visibile
        j++;          // j definito nel blocco C
        i++;
        // ...
    }
    i++;
    j++;
    // errore: j non visibile
}
}

```

Un identificatore definito in un blocco si dice *identificatore locale* di quel blocco.

Come visto nel sottoparagrafo 4.7.3, l'istruzione *for* prevede la possibilità che l'inizializzazione contenga una definizione: in questo caso l'identificatore definito è visibile, oltre che nella intestazione, nell'istruzione che costituisce il corpo del *for* e solo in essa (se il corpo del *for* è un blocco, l'identificatore non può essere ridefinito in tale blocco).

Notare che l'eventuale identificatore definito nell'intestazione di un'istruzione *for* non deve essere stato già definito nel blocco che contiene l'istruzione *for* stessa. Notare anche che, se in uno stesso blocco compaiono (in parallelo) più istruzioni *for*, le varie inizializzazioni possono contenere la definizione dello stesso identificatore.

La definizione di una variabile locale può prevedere o meno un inizializzatore, e questo viene valutato quando l'esecuzione raggiunge tale definizione; se l'inizializzatore non è presente, la variabile assume un valore non definito.

## 4.10. Memorizzazione di più entità in un file

Come detto nel paragrafo 4.1, comunemente le entità che compongono

un programma (classi, enumerazioni o interfacce) vengono scritte ciascuna su un file separato, avente il nome della entità che esso contiene ed estensione *java*. Tuttavia, più entità possono anche essere memorizzate in un unico file: se questo contiene una entità pubblica (per esempio, la classe principale) il suo nome deve essere quello di tale classe.

#### 4.10.1. Traduzione

Il comando di traduzione specifica un nome di file avente estensione *java*. Il traduttore produce tanti file oggetto (con estensione *class*) quante sono le entità contenute nel file sorgente, ciascuno avente il nome della corrispondente entità ed estensione *class*. Se una entità fa uso di un'altra entità per la quale non è stato prodotto il file con estensione *class*, il compilatore cerca il file sorgente avente il nome di tale entità e lo compila, e così via. Occorre quindi memorizzare le entità in file aventi un nome che rispetta, nell'ordine, quello delle entità utilizzate. Così facendo, basta emettere il comando di traduzione per il file che contiene la classe principale, per ottenere la traduzione di tutto il programma.

Per esempio, supponiamo di avere la seguente situazione:

```
// file Princ.java
class Uno
{ /* ... */ }

public class Princ
{ public static void main(String[] args)
  { // utilizzazione della classe Uno;
    // utilizzo della classe Sec;
    // utilizzo della classe Sec1;
    // ...
  }
}

// file Sec.java
class Sec
{ /* ... */ }

class Sec1
```



```
{ /* ... */ }
```

Con il comando ***javac Princ.java*** viene anzitutto compilato il file *Princ.java* e prodotti i file *Uno.class* e *Princ.class*; poichè la classe *Princ* utilizza la classe *Sec*, viene prima cercato (e non trovato) il file *Sec.class*, e quindi cercato e trovato il file *Sec.java*, che viene compilato con produzione dei file *Sec.class* e *Sec1.class*; infine, poiché la classe *Princ* utilizza la classe *Sec1*, viene cercato e trovato il file *Sec1.class*.

Notare che se nella funzione *main()* della classe *Princ* invertissimo l'ordine di utilizzo delle classi *Sec* e *Sec1*, con l'unico comando sopra specificato la compilazione si arresterebbe, in quanto il traduttore non riuscirebbe a trovare il file *Sec1.java*.

#### 4.10.2. Esecuzione

Il comando di esecuzione, nell'ipotesi che siano presenti i file oggetto contenenti ciascuno la rispettiva entità, va dato specificando il nome della classe principale. Nel caso del sottoparagrafo precedente, essendo presenti i file *Uno.class*, *Princ.class*, *Sec.class*, *Sec1.class*, si può emettere il comando di esecuzione ***java Princ***.

## 5. Funzioni

### 5.1. Struttura di una funzione

La classe principale di un programma può prevedere, oltre alla funzione *main()*, altre funzioni che possono essere richiamate dalla funzione *main()* come funzioni di utilità: in questo caso devono essere definite statiche, come verrà chiarito nel Capitolo 9. Pertanto, le regole sintattiche che verranno presentate si riferiscono solo alle funzioni statiche (senza altri modificatori), che chiameremo semplicemente *funzioni*.

Nella definizione di una funzione occorre indicare, oltre al nome della funzione stessa, il tipo del risultato e gli eventuali parametri su cui essa deve operare (parametri formali), e specificare quali azioni deve eseguire per ottenere il risultato (corpo della funzione), secondo la seguente sintassi:

```
function-definition
    static function-specification
function-specification
    function-header function-body
function-header
    result-type identifier ( formal-parameter-listopt )
result-type
    void
    type
formal-parameter
    finalopt type variable-identifier
function-body
    block
```

Il corpo di una funzione (essendo un blocco) non può contenere altre definizioni di funzione, ma solo definizioni di variabili e di classi (o di interfacce).

La chiamata di una funzione avviene per mezzo di un designatore di funzione, il quale, oltre al nome, contiene sotto forma di espressioni gli argomenti su cui la funzione opera per quella chiamata (argomenti attuali), che devono essere in corrispondenza con i parametri formali in numero, ordine, e tipo:

*function-designator*  
*function-identifier ( expression-list~~opt~~ )*

La chiamata produce una istanza di funzione, ossia l'esecuzione della funzione con quegli argomenti attuali.

I parametri formali sono contenitori nei quali vengono trasferiti gli argomenti attuali all'atto della chiamata della funzione (sono come variabili che vengono inizializzate): gli argomenti attuali sono quindi i valori iniziali dei parametri formali per quella chiamata.

Una funzione può restituire come valore un risultato, oppure non avere nessun risultato (parola chiave *void*). Nel corpo della funzione si possono compiere elaborazioni utilizzando i parametri formali (come normali variabili): questi possono anche essere dichiarati *final*, nel qual caso non sono consentite operazioni di scrittura. Se la funzione restituisce un risultato, il corpo deve prevedere almeno una istruzione *return*, con la specifica dell'espressione il cui valore rappresenta il risultato stesso. Il corpo di una funzione può anche fare uso di altre funzioni.

Si consideri il seguente esempio, nel quale si definisce la funzione *mcd()* per il calcolo del massimo comun divisore fra coppie di numeri interi maggiori di 0:

```
// file Elabora.java
public class Elabora
{ static int mcd(int a, int b)
  { while (a != b)
    { if (a > b) a -= b; else b -= a;
      return a;
    }
  }
  public static void main(String[] args)
  { int h, k, m, n, i;
```

```

        Console.scriviStringa("Numero di mcd:");
        n = Console.leggiIntero();
        Console.scriviStringa("Coppie di numeri:");
        for (i=0; i<n; i++)
        { h = Console.leggiIntero();
          k = Console.leggiIntero();
          m = mcd(h, k);
          // ...
          Console.scriviIntero(m);
        }
    }
}

```

La funzione ha due parametri formali interi,  $a$  e  $b$ , e produce un risultato intero. Ogni volta che la funzione viene chiamata, i parametri formali assumono i valori di  $h$  e  $k$ , e il risultato prodotto dalla funzione viene assegnato alla variabile intera  $m$ .

I contenitori relativi ai parametri formali memorizzano sempre *valori*, inizialmente quelli specificati come argomenti attuali. La funzione può anche modificare il contenuto dei parametri formali, ma non vi è alcuna ripercussione sugli argomenti attuali. A titolo di esempio, proviamo a realizzare una banale funzione che scambia i valori di due parametri formali interi:

```

// file Scambia.java
public class Scambia
{ static void scambia(int a, int b)
  { int c;
    c = a; a = b; b = c;
  }
  public static void main(String[] args)
  { int h, k;
    h = Console.leggiInt(); k = Console.leggiInt();
    scambia(h, k);
    // scambia i valori di a e di b, che hanno come
    // valori iniziali
    // quelli di h e di k, rispettivamente
    // non scambia i valori di h e k
    Console.scriviInt(h); Console.scriviInt(k);
    Console.nuovaLinea();
  }
}

```

Quando la funzione *scambia()* viene chiamata (dalla funzione *main()*), il valore di *h* viene ricopiato in *a*, e il valore di *k* in *b*. La funzione scambia i valori di *a* e *b*, e non di *h* e *k*: la funzione non ha quindi alcun effetto sugli argomenti attuali.

### 5.1.1. Espressione designatore di funzione

Un designatore di funzione è un'espressione, il cui valore è costituito dal risultato dell'istanza della funzione stessa: tale espressione (se significativa, ossia non *void*) può a sua volta far parte di un'altra espressione, e in particolare di una espressione di assegnamento. Per esempio, nella funzione *main()* si può scrivere una semplice espressione per il calcolo del minimo comune multiplo:

```
int a, b;  
...  
mcm = a * b / mcd(a, b);
```

Un designatore di funzione contiene l'operatore '()', che come detto nel Capitolo 3, ha una precedenza maggiore degli altri (uguale solo al '++' e '--' postfissi), per cui nelle espressioni viene eseguito con alta precedenza.

**Osservazione.** L'espressione "designatore di funzione", che corrisponde a una chiamata di funzione, può divenire un'istruzione con l'aggiunta del carattere ';' finale: questa istruzione è significativa per funzioni che non restituiscono un risultato, ma che effettuano altri generi di azione.

## 5.2. Visibilità e tempo di vita

Il corpo di una funzione è un blocco: pertanto, le variabili definite nella funzione hanno visibilità di blocco. I parametri formali di una funzione hanno le stesse caratteristiche delle variabili locali (hanno visibilità di blocco): essi sono visibili in tutto il blocco che costituisce il corpo della

funzione, e il loro identificatore non può essere ridefinito. Parametri formali e variabili locali di funzioni diverse sono comunque diversi, anche se hanno lo stesso identificatore.

I parametri formali e le variabili locali di una funzione nascono quando inizia l'esecuzione della funzione e muoiono quando la sua esecuzione giunge al termine (hanno lo stesso tempo di vita di un'istanza della funzione): pertanto, eventuali modifiche che possono subire i parametri formali e le variabili locali durante una esecuzione non si conservano per eventuali esecuzioni successive, ma vengono perdute. In altri termini, se una stessa funzione viene eseguita più volte, ogni volta nascono e muoiono i parametri formali e le variabili locali, senza che vi sia nessun "ricordo" dei loro valori nelle esecuzioni precedenti.

### 5.3. Variabili comuni

Le funzioni di un programma, oltre che mediante argomenti attuali, possono scambiarsi informazioni anche mediante variabili comuni (o condivise), che vengono definite nella classe principale (come membri della classe, al di fuori delle funzioni): a tale scopo esse devono essere definite statiche, come verrà chiarito nel Capitolo 9. Pertanto, le regole sintattiche che verranno presentate si riferiscono solo alle variabili comuni (variabili membro statiche), senza altri modificatori tranne *final*. Esse si definiscono nel seguente modo:

*shared-variable-definition:*  
**final***opt* **static** *variable-definition*

Le variabili comuni sono visibili nel corpo di tutte le funzioni, e hanno un tempo di vita uguale a quello del programma: esse mantengono quindi il loro valore fra un'istanza e un'altra della stessa o di funzioni diverse. A titolo di esempio, consideriamo il seguente programma:

```
// file Calcola.java
public class Calcola
{ static int g = 0;
  static void incrementa()
```

```

    { g++; }
    static void somma()
    { g += 4; }
    public static void main(String[] args)
    { for (int i=0; i<10; i++)
      { incrementa(); if (i > 7) somma(); }
      Console.scriviIntero(g);
    }
}

```

In esecuzione, abbiamo il seguente risultato:

```
18
```

## 5.4. Overloading

Funzioni differenti possono avere lo stesso identificatore (sovrapposizione o *overloading*): esse devono però avere una differente firma (*signature*), ossia parametri formali differenti in numero e/o tipo e/o ordine (non è consentito che la differenza riguardi solo il tipo del risultato).

Quando viene chiamata una funzione che appartiene a un gruppo di funzioni sovrapposte, la discriminazione su quale funzione debba essere effettivamente invocata avviene in fase di traduzione ad opera del compilatore, il quale effettua la selezione in base al numero e/o tipo e/o ordine degli argomenti attuali. Per esempio, si può avere:

```

// file Over.java
public class Over
{ static int fai(int a, int b)          // fai_int_int
  { return a+b; }
  static double fai(double a)          // fai_double
  { return a/2; }
  public static void main(String[] args)
  { int i = fai(10, 20);                // chiamata di fai_int_int
    Console.scriviInt(i);
    double d = fai (5.4);              // chiamata di fai_double
    Console.scriviReal(d);
  }
}

```

```
        Console.nuovaLinea();  
    }  
}
```

## 5.5. Ricorsione

In Java (come nella maggior parte dei linguaggi di programmazione) una funzione può invocare non solo un'altra funzione, ma anche se stessa: in questo caso la funzione è *ricorsiva* (si ha una nuova istanza della medesima funzione mentre l'istanza attuale non è ancora conclusa). Questa caratteristica del linguaggio rende agevole la programmazione in tutti quei casi in cui risulta naturale formulare il problema da risolvere in maniera ricorsiva. Un classico esempio è quello del calcolo del fattoriale di un numero intero positivo  $n$ , che può essere espresso ricorsivamente nel seguente modo:

$$\text{fattoriale}(n) = \begin{array}{ll} 1 & \text{se } n = 0 \\ n * \text{fattoriale}(n-1) & \text{se } n > 0 \end{array}$$

Questo procedimento di calcolo può essere agevolmente espresso mediante la seguente funzione ricorsiva:

```
static int fatt(int n)  
{ if (n == 0) return 1;  
  return n*fatt(n-1);  
}
```

Per comprendere come il calcolo indicato venga effettivamente svolto, si supponga che nella funzione *main()* compaia la seguente istruzione:

```
ris = fatt(3);
```

La chiamata *fatt(3)* attiva il seguente calcolo:

```
return 3*fatt(2);    // fatt(3)
```



Il calcolo della precedente espressione comporta preliminarmente la chiamata di *fatt(2)*, che richiede a sua volta il seguente calcolo:

```
return 2*fatt(1);      // fatt(2)
```

Analogamente, il calcolo della precedente espressione richiede la preliminare chiamata di *fatt(1)*, e quindi il seguente calcolo:

```
return 1*fatt(0);      // fatt(1)
```

Infine, per valutare la precedente espressione è necessario chiamare anzitutto *fatt(0)*, che viene eseguita nel seguente modo:

```
return 1;               // fatt(0)
```

A questo punto, trovato *fatt(0)*, vengono calcolati prima *fatt(1)*, poi *fatt(2)*, quindi *fatt(3)*, e infine la quantità *ris*. Le varie istanze della funzione ricorsiva vengono quindi chiuse in ordine inverso rispetto alla loro chiamata, e il valore associato alla funzione *fatt()* assume nell'ordine i valori 1, 1, 2, 6.

Anche il procedimento di calcolo del massimo comun divisore fra due interi positivi può essere formulato come funzione ricorsiva, nel seguente modo:

```
static int mcd(int alfa, int beta)  
{ if (beta == 0) return alfa;  
  return mcd(beta, alfa % beta);  
}
```

Un altro semplice calcolo che può essere espresso come funzione ricorsiva è la somma dei primi *n* numeri interi positivi. Si ha infatti:

```
static int somma(int n)  
{ if (n == 0) return 0;  
  return n + somma(n - 1);  
}
```

(è probabile che il calcolo della formula  $(1/2)*n*(n+1)$  sia più veloce).

Analogamente, si può scrivere la seguente funzione ricorsiva per effettuare l'elevamento a potenza di un numero reale per un numero intero positivo:

```
static double pot(double x, int n)
{ if (n == 0) return 1;
  return x*pot(x, n-1);
}
```

(anche in questo caso esistono algoritmi migliori, come illustrato successivamente).

Come ulteriore esempio, si può scrivere come funzione ricorsiva il calcolo dell' $n$ -mo elemento ( $n = 1, 2, \dots$ ) della serie di Fibonacci (partendo dagli interi 0 e 1, ogni elemento della serie si ottiene sommando i due precedenti):

```
static int fib(int n)
{ if (n == 1) return 0;
  if (n == 2) return 1;
  return fib(n-1)+fib(n-2);
}
```

Si è visto nel paragrafo 5.2 che per ogni esecuzione di una funzione si ha una differente copia dei parametri formali e delle variabili locali. Questo è vero anche quando la funzione chiama se stessa: in questo caso, inoltre, poiché le precedenti esecuzioni non sono ancora terminate, i parametri formali e le variabili locali di tutte le istanze precedenti (nel processo ricorsivo) esistono ancora (con gli stessi identificatori). Tuttavia, durante la  $i$ -ma esecuzione si può fare riferimento solo alla  $i$ -ma copia dei parametri formali e delle variabili locali, mentre le altre copie non sono visibili. Un esempio di funzione ricorsiva che utilizza variabili locali si ha nell'elevamento a potenza di un numero reale per un numero intero positivo. Questo calcolo, già risolto con una delle precedenti funzioni, può essere espresso in maniera più efficiente nel seguente modo:

potenza (x, n) =	1	se n = 0
	x	se n = 1
	potenza (x <sup>2</sup> , n/2)	se n > 1 e pari
	x*potenza (x <sup>2</sup> , (n-1)/2)	se n > 1 e dispari

Ne discende la seguente funzione Java:

```
static double pot(double x, int n)
{ double t;
  if (n == 0) return 1;
  if (n == 1) return x;
  if (n%2 == 0) t = 1; else t = x;
  return t*pot(x*x, n/2);
}
```

Dagli esempi visti finora, si deduce che per scrivere una funzione ricorsiva bisogna individuare due componenti fondamentali dell'algoritmo: i) uno o più *casi base*, per i quali termina la successione delle chiamate ricorsive, e ii) uno o più *passi ricorsivi*. La ricorsione è controllata dal valore di uno (in genere) dei parametri (parametro di controllo), in base al quale si sceglie se eseguire le istruzioni relative a un caso base o a un passo ricorsivo. In un passo ricorsivo la funzione viene chiamata nuovamente passandole come argomento attuale un nuovo valore del parametro di controllo (se il valore venisse passato immutato la ricorsione non terminerebbe). Il risultato di questa chiamata, spesso ulteriormente elaborato, viene restituito al chiamante dell'istanza corrente: in uno dei casi base il risultato viene calcolato senza altre chiamate ricorsive.

Osserviamo che questo schema concettuale è parallelo a quello usato nelle computazioni iterative, in cui la ripetizione di un ciclo viene condizionata dai valori di una variabile di controllo, e uno (o alcuni) di tali valori provoca la terminazione del ciclo. L'aggiornamento della variabile di controllo eseguito nel corpo del ciclo corrisponde al passaggio di un nuovo valore del parametro di controllo a una chiamata ricorsiva.

Notare che per ogni funzione formulata in maniera ricorsiva esiste una corrispondente funzione formulata in maniera iterativa (per esempio, la formulazione iterativa del calcolo del fattoriale di un numero intero positivo è stata riportata nel Capitolo 4). Spesso, inoltre, la formulazione iterativa è più conveniente, in termini di tempo di esecuzione e di occupazione di memoria. Tuttavia, in diversi casi è più agevole (per il programmatore) esprimere un procedimento di calcolo in maniera ricorsiva, e questo può riflettersi in una maggiore concisione e chiarezza del programma, e quindi una minore probabilità di commettere errori. Il classico esempio della Torre di Hanoi, riportato nel paragrafo seguente,

ben evidenzia l'immediatezza e la semplicità espressiva della formulazione ricorsiva.

## 5.6. Esempio: Torre di Hanoi

Un classico problema che risulta facilmente risolubile con un procedimento ricorsivo è il gioco della torre di Hanoi. Sono dati tre paletti, indicati come paletto 1, paletto 2 e paletto 3. Sul paletto 1 è infilata una pila di dischi di diametro decrescente, in modo che il disco di diametro maggiore stia sul fondo. Il problema consiste nel definire quali mosse sono necessarie per spostare tutti i dischi dal paletto 1 a quello 3 (utilizzando il paletto 2 come appoggio). Ciascuna mossa deve essere fatta rispettando le seguenti regole:

- è possibile spostare un solo disco alla volta da un paletto a un altro;
- non è possibile mettere un disco di diametro maggiore su uno di diametro minore;
- dopo ogni mossa ogni disco deve trovarsi in uno dei tre paletti.

Il problema può essere formulato in maniera ricorsiva nel seguente modo:

- se si ha un solo disco, basta spostarlo dal paletto dove si trova sul paletto dove si vuol formare la nuova torre;
- se si hanno  $n$  dischi ( $n > 1$ ) su un dato paletto (paletto sorgente), e si vogliono spostare su un nuovo paletto (paletto destinatario), utilizzando un paletto ausiliario, è sufficiente:
  - muovere gli  $(n-1)$  dischi più in alto dal paletto sorgente al paletto ausiliario, utilizzando il paletto destinatario come appoggio;
  - spostare il disco rimasto dal paletto sorgente al paletto destinatario;
  - muovere gli  $(n-1)$  dischi dal paletto ausiliario al paletto destinatario, utilizzando il paletto sorgente come appoggio.

Tale procedimento ricorsivo si può esprimere col seguente programma:

```
// file Hanoi.java
public class Hanoi
{ static void sposta(int pall1, int pal2)
  { Console.scriviInt(pall1);
    Console.scriviInt(pal2);
    Console.nuovaLinea();
  }
  static void hanoi(int numero,
                    int sorg, int aus, int dest)
  { if (numero == 1) sposta(sorg, dest);
    else
    { hanoi(numero-1, sorg, dest, aus);
      sposta(sorg, dest);
      hanoi(numero-1, aus, sorg, dest);
    }
  }
  public static void main(String[] args)
  { int n;
    Console.scriviStringa("Numero di dischi:");
    n = Console.leggiIntero();
    Console.scriviStringa("Le mosse sono:");
    hanoi(n, 1, 2, 3);
  }
}
```

Le soluzioni prodotte da questo programma per un numero di dischi pari a tre e a quattro, rispettivamente, sono le seguenti:

Numero di dischi:

3

Le mosse sono:

1 3

1 2

3 2

1 3

2 1

2 3

1 3

Numero di dischi:

4

Le mosse sono:

1 2

```
1 3
2 3
1 2
3 1
3 2
1 2
1 3
2 3
2 1
3 1
2 3
1 2
1 3
2 3
```

Come si può osservare, se il numero dei dischi del gioco è  $n$ , il numero delle mosse è  $2^n - 1$ .

## 5.7. Funzione *exit()*

Nel corpo di una funzione (scritta dal programmatore) si può usare la funzione statica *exit()* della classe *System* (package *java.lang*), che ha un parametro formale di tipo intero: quando eseguita, essa provoca la terminazione forzata dell'intero programma, lasciando non conclusa l'istanza della funzione che la invoca, nonché eventuali altre istanze ancora aperte della stessa funzione (in caso di ricorsione) o di altre funzioni, e restituisce al sistema operativo il valore dell'argomento attuale (per convenzione questo è 0 se non vi sono stati errori, diverso da 0 altrimenti).

L'effetto della funzione *exit()* non va quindi confuso con quello dell'istruzione *return*, la quale chiude l'istanza di funzione che la esegue, e ritorna ad un'eventuale istanza precedente della stessa funzione o di un'altra funzione che l'ha chiamata. Per esempio, il programma:

```
public class Fine
{ static void f()
  { // ...
    System.exit(0);
```

```
    }  
    public static void main(String[] args)  
    {  
        f();  
        Console.scriviStringa("Fine di f()");  
    }  
}
```

non produce alcuna uscita, mentre il programma:

```
public class Fine  
{ static void f()  
  { // ...  
    return;  
  }  
  public static void main(String[] args)  
  {  
      f();  
      Console.scriviStringa("Fine di f()");  
  }  
}
```

stampa il messaggio:

```
Fine di f()
```

## 5.8. Ambiente automatico e ambiente statico

Come detto nel paragrafo 5.2, i parametri formali e le variabili locali di una funzione nascono quando inizia l'esecuzione della funzione e muoiono quando la sua esecuzione giunge al termine (hanno lo stesso tempo di vita di un'istanza della funzione). Tali entità vengono dette *automatiche* (e costituiscono l'ambiente automatico). Esistono quindi tanti ambienti automatici quante sono le istanze delle funzioni.

Gli ambienti automatici vengono allocati in una zona di memoria le cui dimensioni crescono a seguito di una chiamata di funzione (nuova istanza) e decrescono quando la funzione termina (fine istanza). Per consentire

l'annidamento delle istanze (come le chiamate ricorsive), tale la zona di memoria viene gestita con la regola LIFO (*Last In First Out*), caratteristica delle organizzazioni dei dati a *pila*.

Le variabili che compaiono nella classe che costituisce il programma (al di fuori delle funzioni), vengono (al momento) definite *statiche*, per poter essere comuni a tutte le funzioni compresa la funzione *main()*: esse sono allocate in una zona di memoria riservata alle classi, detta appunto memoria statica, e hanno lo stesso tempo di vita della classe che costituisce il programma. Le variabili statiche fanno parte dell'ambiente statico.

Tutte le funzioni vengono (al momento) definite statiche, e fanno anch'esse parte dell'ambiente statico. La funzione *main()* viene eseguita per prima e richiama tutte le altre: ogni istanza di funzione può far uso, oltre che dell'ambiente automatico di quell'istanza, dell'ambiente statico.

Come verrà chiarito nel Capitolo 9, l'ambiente statico di una classe è associato alla classe, senza che sia richiesta alcuna istanza della classe stessa.



## 6. Enumerazioni, array, stringhe

### 6.1. Tipi derivati, variabili e oggetti

I tipi primitivi prevedono la *definizione di variabili* di un dato tipo. Una variabile di un tipo primitivo ha un nome e un valore: tipicamente, la variabile ha per nome un identificatore, e per valore uno di quelli previsti dal tipo della variabile stessa. Il tempo di vita di una tale variabile dipende dal luogo in cui è definita (per esempio, le variabili definite nel corpo di una funzione sono automatiche).

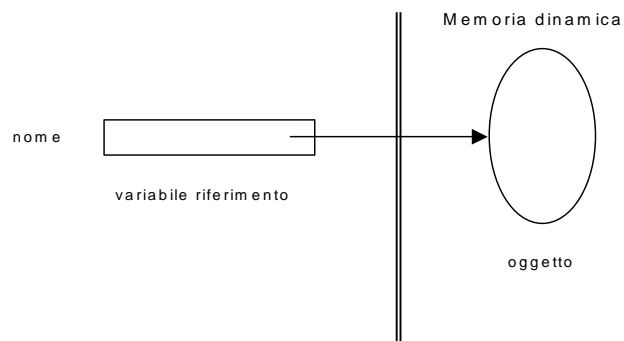


Figura 6.1. Riferimenti e oggetti

I tipi derivati (*enumerazioni, array, stringhe, eccetera*) prevedono,

invece, la *definizione del tipo*, la *definizione di variabili* di quel tipo e la *creazione di oggetti* di quel tipo. Un tipo derivato è costituito da più componenti (anche una sola) raggruppate in un'unica entità. Una variabile di un tipo derivato ha le stesse caratteristiche di una variabile di un tipo primitivo, con la differenza che il suo valore è un indirizzo, quello di un oggetto di quel tipo (la variabile prende anche il nome di *riferimento*). Un oggetto di un tipo derivato ha un tempo di vita dinamico: viene creato esplicitamente quando serve (per mezzo dell'operatore *new*), e viene distrutto automaticamente quando non vi sono più riferimenti di quell'oggetto. L'oggetto non ha un proprio identificatore, ma viene individuato da una variabile riferimento (Fig. 6.1), mediante la quale si individuano le singole componenti dell'oggetto stesso. Gli oggetti vengono allocati in una zona di memoria detta appunto *memoria dinamica*.

I tipi derivati possono essere già definiti nelle Java API, oppure possono essere definiti dall'utente. Per semplicità, in questo capitolo le definizioni di tipo verranno effettuate nel file contenente la classe principale, come entità autonome fuori da questa.

Le variabili di un tipo derivato, come quelle di un tipo primitivo, possono essere parametri o variabili locali di una funzione, oppure variabili comuni alle funzioni definite nella classe che costituisce il programma.

## 6.2. Tipi e variabili enumerazione

Un *tipo enumerazione* è un insieme di costanti simboliche definite dal programmatore, ciascuna individuata da un identificatore e detta *enumeratore*. I tipi enumerazione vengono usati per rappresentare un numero limitato di valori associati a informazioni non numeriche. Per esempio, un programma che deve compiere elaborazioni sui semafori può usare un tipo enumerazione le cui costanti sono costituite dai possibili colori di un semaforo.

Un tipo enumerazione viene definito dal programmatore, e la forma più semplice di definizione è la seguente:

```
enum-definition  
    enum identifier { enumerator-list }  
enumerator
```

*identifier*

Un esempio di tipo enumerazione è il seguente:

```
enum Semaforo (verde, giallo, rosso )
```

Gli enumeratori sono identificatori di costanti: il valore di ogni costante è un *referimento*, ossia l'indirizzo di un oggetto che può racchiudere al suo interno altre informazioni legate all'enumeratore stesso (al momento non consideriamo questa possibilità). Gli oggetti riferiti dagli enumeratori vengono creati implicitamente nel momento in cui viene definito il tipo. Per esempio, nel caso precedente si ha la situazione di Fig. 6.2.

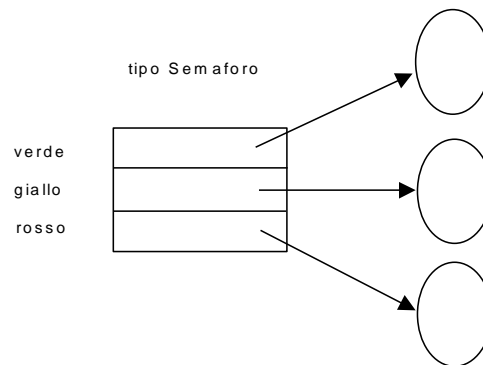


Figura 6.2. Tipo enumerazione ed enumeratori

Un tipo enumerazione è in realtà un caso particolare del più generale tipo classe, e gli enumeratori sono riferimenti statici costanti di oggetti di quella classe (Capitolo 9).

Al di fuori del tipo, un enumeratore viene individuato attraverso un identificatore qualificato:

*qualified-enumerator-identifier*  
*enumerator-identifier . enumerator*

Una *variabile* di un tipo enumerazione è un riferimento che può assumere come valori o il letterale *null* (riferimento di nessun oggetto) o

quelli degli enumeratori del tipo, e riferire quindi uno degli oggetti del tipo.

Le variabili di un tipo enumerazione si definiscono come le variabili di un tipo primitivo (Capitolo 3), con la seguente sintassi:

```
enumeration-variable-definition
enumeration-identifier variable-specifier-list ;
```

Per esempio, scrivendo:

```
Semaforo sem1 = Semaforo.verde, sem2;
Semaforo sem3 = sem1;
...
sem1 = Semaforo.rosso;
```

si definiscono tre variabili, la prima (*sem1*) inizializzata con il contenuto dell'enumeratore *verde*, la seconda (*sem2*) non inizializzata, la terza (*sem3*) inizializzata con il valore di *sem1*, che possono assumere come valori quelli degli enumeratori *verde*, *giallo* e *rosso*.

A una variabile di un tipo enumerazione si può applicare la funzione *name()*, che restituisce come stringa l'identificatore dell'enumeratore corrispondente al valore della variabile stessa. Per esempio, scrivendo:

```
Console.scriviStringa(sem1.name());
```

si ottiene la stampa di una delle stringhe *verde*, *giallo* o *rosso*, a seconda del valore di *sem1*.

Non possono essere creati esplicitamente oggetti di un tipo enumerazione.

Usando i tipi enumerazione, si può riscrivere il programma della torre di Hanoi, chiamando i paletti *sinistro*, *centrale* e *destro*:

```
// file Hanoi1.java
enum Pal
{ sinistro, centrale, destro }

public class Hanoi1
{ static void sposta(Pal pal1, Pal pal2)
  { Console.scriviStr(pal1.name());
    Console.scriviStr(pal2.name());
  }
}
```

```
        Console.nuovaLinea();
    }
    static void hanoi(int numero,
        Pal sorg, Pal aus, Pal dest)
    { if (numero == 1) sposta(sorg, dest);
      else
      { hanoi(numero-1, sorg, dest, aus);
        sposta(sorg, dest);
        hanoi(numero-1, aus, sorg, dest);
      }
    }
    public static void main(String[] args)
    { int n;
      Console.scriviStringa("Numero di dischi:");
      n = Console.leggiIntero();
      Console.scriviStringa("Le mosse sono:");
      hanoi(n, Pal.sinistro, Pal.centrale, Pal.destro);
    }
}
```

Un tipico esempio di esecuzione è il seguente:

```
Numero di dischi:
3
Le mosse sono:
sinistro destro
sinistro centrale
destro centrale
sinistro destro
centrale sinistro
centrale destro
sinistro destro
```

I tipi enumerazione possono essere utilizzati anche nelle funzioni (tipo di parametri formali e di variabili locali, e tipo del risultato).

### 6.3. Tipi array, variabili array e oggetti array

Un *tipo array* rappresenta un aggregato di elementi di uno stesso tipo,

ai quali si accede mediante un indice che ne individua la posizione (un tipo array non dipende quindi dal numero di elementi, ma unicamente dal loro tipo). Un *oggetto array* memorizza uno di questi aggregati, che può non contenere alcun elemento (array vuoto), ovvero contenere un numero intero positivo  $N$  di elementi, il cui indice va da  $0$  a  $N-1$ . Oggetti array relativi ad aggregati di elementi dello stesso tipo sono quindi dello stesso tipo.

Gli elementi di un array possono essere di qualunque tipo: al momento, tuttavia, facciamo l'ipotesi semplificativa che questi siano di un tipo primitivo.

Un oggetto array viene individuato tramite una *variabile array* (il cui tipo è quello dell'oggetto array stesso), che può assumere come valore o il letterale *null* (riferimento di nessun oggetto) o il riferimento di un oggetto array di quel tipo (Fig. 6.3).

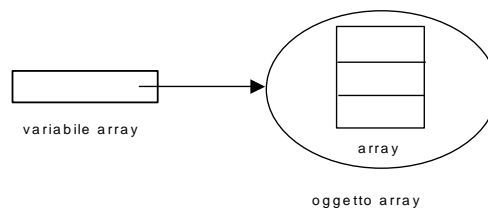


Fig. 6.3. Variabile array e oggetto array

La definizione di un tipo array non avviene esplicitamente, mediante un identificatore di tipo, ma il tipo viene semplicemente indicato specificando il tipo degli elementi:

*array-type-indicator*  
*element-type* []

Pertanto, la definizione di variabili di un tipo array comporta la definizione contestuale del tipo, secondo la seguente sintassi:

*array-variable-definition*  
*array-type-indicator* *array-variable-specifier-list* ;  
*array-variable-specifier*  
*array-variable-identifier* *array-initializer* **opt**

```

array-initializer
    = array-variable-initializer
    = array-object-initializer

```

La prima forma di inizializzazione prevede un'espressione che ha come risultato un riferimento. Per esempio, possiamo scrivere (Fig. 6.4):

```

int[] ar, ar1;
// definizione di due variabili array ar e ar1
// di tipo int[], non inizializzate
...
int[] ar2 = ar;
// definizione di una variabile ar2 inizializzata

```

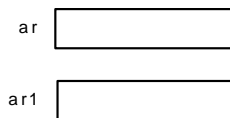


Fig. 6.4. Definizione delle variabili array *ar* e *ar1*.

Un oggetto array viene creato (e allocato in memoria dinamica) per mezzo dell'operatore *new* (che richiede la specifica del tipo e del numero di elementi (mediante un'espressione), e che restituisce l'indirizzo dell'oggetto allocato), secondo la seguente sintassi:

```

array-object-creation
    new element-type [ expression ]

```

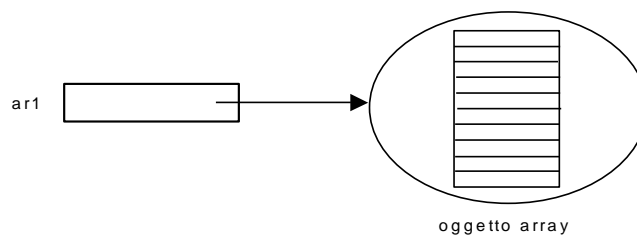


Fig. 6.5. Creazione di un oggetto array riferito da *ar1*.

L'indirizzo restituito dall'operatore *new* può essere assegnato a una variabile array, oppure può essere utilizzato come iniziatore. A titolo di esempio possiamo scrivere (Fig. 6.5):

```
ar1 = new int[10];  
// allocazione di un oggetto array  
// costituito da 10 interi,  
// il cui indirizzo viene assegnato a ar1  
int[] ar3 = new int[15];
```

Quando viene creato un oggetto array, i suoi elementi assumono un valore iniziale predefinito, ossia 0 se di tipo numerico, *false* se di tipo booleano, *null* se di un tipo derivato.

La definizione di una variabile array inizializzata e la creazione contestuale di un oggetto array inizializzato può avvenire attraverso una forma di inizializzazione comprendente un aggregato (*array-object-initializer*), composto a sua volta da una lista di espressioni racchiuse tra parentesi graffe (il numero di elementi dell'array è determinato dal numero di espressioni). Si consideri, per esempio, la seguente istruzione:

```
int[] ar4 = { 1, 3, 4, 2, 6 };
```

Questa crea un oggetto array di cinque elementi, inizializzati come specificato, produce come risultato il suo riferimento che viene memorizzato in *ar4*. Notare che non è consentito definire separatamente una variabile array e creare un oggetto array inizializzato con un aggregato.

Il numero di elementi di un oggetto array è memorizzato in una variabile *length* di tipo intero (e di valore costante, quindi *final*), che viene creata insieme all'oggetto array (fa parte dell'oggetto array stesso). Per esempio, la lunghezza dell'array *ar* può essere assegnata a una variabile intera *n* nel seguente modo:

```
int n = ar.length;
```

### 6.3.1. Elementi di un array

Gli elementi di un array vengono selezionati per mezzo dell'operatore



di indicizzazione '[' applicato a una variabile array, il quale racchiude un'espressione intera (indice):

*array-object-component*  
*array-variable [ expression ]*

Il valore dell'espressione può andare da 0 a *length-1* (la validità dell'indice viene controllata in fase di esecuzione). L'operatore di indicizzazione restituisce una variabile (*variabile componente*) avente per tipo quello delle componenti, che può essere utilizzata compatibilmente con il suo tipo. Per esempio, si può avere:

```
int n, i;
...
int[] aa = new int[10];
aa[5] = 15;
aa[10] = 5;    // errore di indice
n = aa[i];    // i puo` andare da 0 a 9
               // altrimenti si ha un errore
```

Come esempio, riportiamo un programma che ricava le 32 cifre binarie di un numero intero positivo le trasforma nei caratteri '0' e '1':

```
// file CalcolaArrayBinario.java
class CalcolaArrayBinario
{ public static void main(String[] args)
  { char c;
    int i, cifra, numero;
    char[] ac = new char[32];
    numero = Console.leggiIntero();
    for (i=0; i < 32; i++)
      // dal bit meno significativo
      // a quello piu` significativo
      { cifra = numero%2;
        numero = numero/2;
        c = (char)(cifra + '0');
        ac[i] = c;
      }
    for(i = 0; i < 32; i++)
      Console.scriviUnCarattere(ac[31-i]);
  }
```



consentendo di avere funzioni che restituiscono il riferimento di oggetti array di dimensioni non prefissate. Per esempio, il seguente programma contiene una funzione che restituisce il riferimento di un oggetto array costituito dai primi numeri interi positivi minori di un intero specificato come parametro formale:

```
// file GeneraArray.java
public class GeneraArray
{   static int[] genera(int n)
    {   int[] ar = new int[n];
        for (int i = 0; i < n; i++) ar[i] = i;
        return ar;
    }
    static public void main(String[] args)
    {   int[] a;
        a = genera(3);
        for (int i = 0; i < a.length; i++)
            Console.scriviInt(a[i]);
        Console.nuovaLinea();
        a = genera(6);
        for (int i = 0; i < a.length; i++)
            Console.scriviInt (a[i]);
        Console.nuovaLinea();
        // ...
    }
}
```

In esecuzione si ha la seguente uscita:

```
0 1 2
0 1 2 3 4 5
```

Notare che, in una funzione, un parametro formale di un tipo array (analogamente a una variabile di un tipo array) è un riferimento: all'atto della chiamata della funzione stessa si ha il trasferimento del valore dell'argomento attuale (anch'esso un riferimento) nel parametro formale. Come detto nel Capitolo 5, eventuali modifiche del parametro formale che vengono effettuate nel corpo della funzione non hanno alcun effetto sull'argomento attuale.

Notare anche che le funzioni che producono come risultato un riferimento di un oggetto array creato localmente non danno luogo a malfunzionamenti dovuti al tempo di vita delle variabili locali: infatti, un eventuale riferimento locale ha un tempo di vita pari a quello di esecuzione della funzione, ma un oggetto array creato localmente, in quanto allocato in memoria dinamica, non muore quando termina l'esecuzione della funzione, per cui il risultato riferisce un oggetto array ancora in vita. Per esempio, nella precedente funzione *genera()*, la variabile *ar* ha un tempo di vita legato all'esecuzione della funzione, ma l'oggetto array riferito da *ar* resta in vita, e dopo l'esecuzione della funzione viene riferito da *a*.

Quando il parametro formale di una funzione è un riferimento, nel corpo della funzione si possono effettuare modifiche sull'oggetto riferito: all'atto della chiamata, il riferimento attuale viene ricopiato in quello formale, e l'esecuzione del corpo della funzione comporta modifiche dell'oggetto attuale.

Nel caso degli array, abbiamo la seguente situazione: mentre una variabile array *ar* è un riferimento, una variabile componente *ar[i]* è un elemento dell'oggetto array riferito da *ar*, come illustrato in Fig. 6.6. Pertanto, si può avere una funzione che modifica gli elementi dell'array riferito da un parametro formale, la quale, quando viene chiamata, modifica gli elementi dell'array riferito dal corrispondenti argomento attuale, come nel seguente caso:

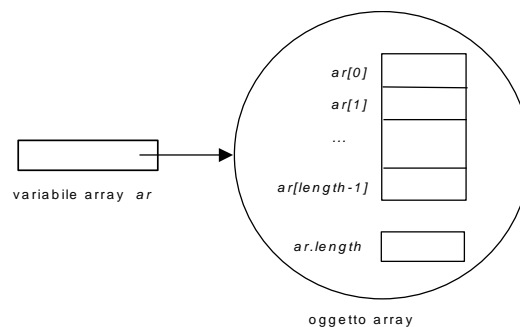


Fig. 6.6. Variabile array e variabili componenti.

```
// file ModificaArray.java
public class ModificaArray
```

```
{ static void modifica(int[] v)
  { for (int i = 0; i < v.length; i++) v[i] *= 2;
  }
static public void main(String[] args)
{ int[] a = { 1, 2, 3, 4 };
  modifica(a);
  for (int i = 0; i < a.length; i++)
    Console.scriviInt(a[i]);
  Console.nuovaLinea();
}
}
```

In esecuzione si ha la seguente uscita:

**2 4 6 8**

### 6.3.3. Assegnamento e ricopiamento di array

L'operatore di assegnamento può essere applicato anche a variabili array dello stesso tipo: vengono però coinvolti i riferimenti, e non gli oggetti array. Per esempio, possiamo scrivere:

```
int[] v1 = { 1, 5, 4 },
      v2 = { 9, 7 };
v2 = v1;
```

I due riferimenti v1 e v2 indirizzano alla fine lo stesso oggetto, ma gli oggetti restano immutati (Fig. 6.7).

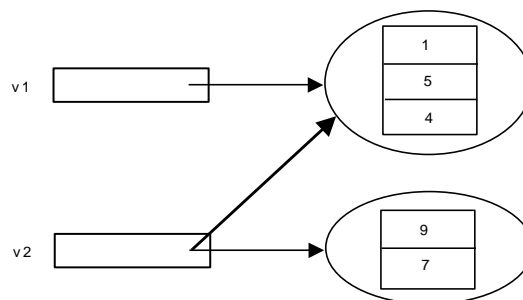


Figura 6.7. Assegnamento a v2 del valore di v1 (freccia in grassetto)

Se vogliamo, invece, ricopiare (in tutto o in parte) gli elementi di un array in un altro array dello stesso tipo, si deve usare la funzione *System.arraycopy()* definita nella classe *System*, (package *java.lang*):

```
void System.arraycopy  
    ( Object ar1 , int i1 , Object ar2 , int i2 , int quanti )
```

(*ar1* riferisce l'oggetto array da cui si deve copiare, *i1* l'indice di partenza di tale oggetto, *ar2* riferisce l'array nel quale si vuole che venga effettuata la copia, *i2* l'indice di partenza di tale oggetto, e *quanti* il numero di elementi da copiare). Per esempio, scrivendo:

```
System.arraycopy(v1, 1, v2, 0, 2);
```

si intende ricopiare 2 elementi dall'oggetto array riferito da *v1*, partendo dal suo indice 1, nell'oggetto array riferito da *v2*, partendo dal suo indice 0.

#### 6.3.4. Uguaglianza fra array

L'operatore di uguaglianza '==' può essere applicato anche a variabili array dello stesso tipo, ma il confronto, che avviene fra riferimenti, risulta vero solo se le due variabili riferiscono lo stesso oggetto.

Per verificare, invece, se due oggetti array dello stesso tipo sono uguali si usa la funzione *Array.equals()* definita nella classe *Arrays* (package *java.util*):

```
boolean Arrays.equals ( Object ar1 , Object ar2 )
```

Due oggetti array dello stesso tipo sono uguali se:

- hanno la stessa lunghezza;
- gli elementi in posizione corrispondente sono uguali.

Per esempio, possiamo avere:

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

```
int[] ar1 = { 1, 4, 2, 3 },
      ar2 = { 1, 4, 2 };
boolean b = Arrays.equals(ar1, ar2); // b vale false
```

### 6.3.5. Array multidimensionali

Un oggetto array (array primario) può avere come elementi altri oggetti array (array secondari): in questo caso negli elementi dell'array primario sono memorizzati i riferimenti degli array secondari (e non direttamente gli array secondari). Gli array secondari, pur dovendo essere dello stesso tipo, non è detto che abbiano lo stesso numero di elementi. Per esempio, si può avere (Fig. 6.8):

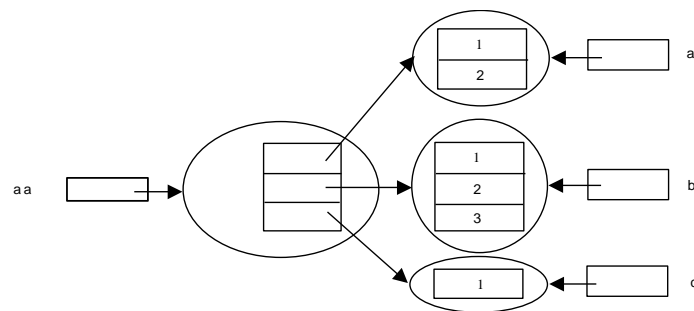


Figura 6.8. Array bidimensionale

```
int[] a = { 1, 2 }, b = { 1, 2, 3 }, c = { 1 };
int[][] aa = new int[3][];
// la variabile array aa ha tre elementi,
// ognuno dei quali è un array di interi
// ...
aa[0] = a; aa[1] = b; aa[2] = c;
```

Anche per gli array primari si possono avere inizializzatori costituiti da aggregati contenenti riferimenti di array secondari, come nell'esempio seguente:

```
int[][] bb = { a, b, c };
```

L'aggregato può anche specificare direttamente gli elementi degli array secondari, nel seguente modo:

```
int[][] cc = { { 1, 2 }, { 1, 2, 3 }, { 1 } };
```

La selezione degli elementi dell'array primario, o degli elementi di un array secondario avviene, rispettivamente, utilizzando uno o due indici. Si consideri, per esempio, il seguente programma:

```
// file ArrayArray.java
class ArrayArray
{ public static void main(String[] args)
  { int i, j;
    int[][] aa = { { 1, 2 }, { 1, 2, 3 }, { 1 } };
    for(i=0; i<aa.length; i++)
      { for(j=0; j<aa[i].length; j++)
        Console.scriviInt(aa[i][j]);
        Console.nuovaLinea();
      }
  }
}
```

In esecuzione si ha la seguente uscita:

```
1 2
1 2 3
1
```

Se tutti gli array secondari sono della stessa dimensione, otteniamo una matrice, come nel seguente esempio (*nr* è il numero righe, *nc* il numero colonne):

```
int[][] m = new int[nr][];
for(int i=0; i<nr; i++) m[i] = new int[nc];
```

In questo caso si può anche applicare direttamente la notazione delle matrici, scrivendo:

```
int[][] m = new int[nr][nc];
```



Come esempio, si può scrivere il seguente programma:

```
// file Matrice.java
class Matrice
{   public static void main(String[] args)
    {   int i, j;
        int[][] m = new int[3][4];
        for(i=0; i<3; i++)
            for(j=0; j<4; j++) m[i][j] = i*4+j;
        for(i=0; i<3; i++)
        {   for(j=0; j<4; j++)
            Console.scriviInt(m[i][j]);
            Console.nuovaLinea();
        }
    }
}
```

In esecuzione si ha la seguente uscita:

```
0 1 2 3
4 5 6 7
8 9 10 11
```

### 6.3.6. Ordinamento di un array

Per ordinare un array, il modo più semplice è di scorrerlo confrontando ogni elemento con il successivo, con un eventuale scambio di posto quando l'ordinamento non è soddisfatto, e di ripetere il tutto fino a quando l'array viene scorso senza scambi (algoritmo *bubblesort*). Quanto detto viene realizzato dal seguente programma

```
// file Ordina.java
class Ordina
{   static void sort(int[] v)
    {   boolean ordinato; int t;
        do
        {   ordinato = true;
            for (int i = 0; i < v.length-1; i++)
                if (v[i] > v[i+1])
                {   ordinato = false;
                    t = v[i]; v[i] = v[i+1]; v[i+1] = t;
                }
        } while (!ordinato);
    }
}
```

```

        t = v[i+1]; v[i+1] = v[i]; v[i] = t;
    }
}
while (!ordinato);
}
public static void main(String[] args)
{
    int[] vett = new int[8];
    Console.scriviStringa("Scrivi 8 interi:");
    for (int i=0; i<vett.length; i++)
        vett[i] = Console.leggiIntero();
    sort(vett);
    for (int i=0; i<vett.length; i++)
        Console.scriviInt(vett[i]);
    Console.nuovaLinea();
}
}

```

Un esempio di esecuzione è il seguente:

```

Scrivi 8 interi:
1 9 7 3 5 4 9 2
1 2 3 4 5 7 9 9

```

### 6.3.7. Ricerca in un array

Per ricercare se un dato elemento è presente all'interno di un array, ed eventualmente conoscerne la posizione, occorre precisare se l'array è o meno ordinato. Se l'array non è ordinato, non si può fare altro che una *ricerca completa*, scorrendo in sequenza gli elementi fino a trovare l'indice di quello cercato o a raggiungere la fine dell'array.

Un programma che realizza una ricerca completa è il seguente (se l'elemento non è presente, assumiamo che il valore della sua posizione sia -1):

```

// file RicercaComp.java
public class RicercaComp
{
    static int ricComp(int[] v, int k)
    {
        for (int i = 0; i < v.length; i++)
            if (k == v[i]) return i;
        return -1;
    }
}

```

```

public static void main(String[] args)
{
    int pos; int[] vett = new int[8];
    Console.scriviStringa("Scrivi 8 interi:");
    for (int i=0; i<vett.length; i++)
        vett[i] = Console.leggiIntero();
    pos = ricComp(vett, 5);
    if (pos != -1) Console.scriviIntero(pos);
    else Console.scriviStringa("Non trovato");
}
}

```

Un esempio di esecuzione è il seguente:

```

Scrivi 8 interi:
1 9 7 3 5 4 9 2
4

```

Se, invece, l'array è ordinato, si può fare una *ricerca binaria* (o *logaritmica*), che consiste nelle seguenti azioni:

- se l'array non ha elementi, la ricerca non ha successo;
- se l'array ha almeno un elemento, confrontare l'elemento cercato con l'elemento di mezzo;
  - se questi sono uguali la ricerca termina con successo;
  - se l'elemento cercato è minore dell'elemento di mezzo, la ricerca prosegue (con le stesse regole) nella prima metà dell'array;
  - se l'elemento cercato è maggiore dell'elemento di mezzo, la ricerca prosegue (con le stesse regole) nella seconda metà dell'array.

Un programma che realizza una ricerca binaria è il seguente (se l'elemento non è presente, assumiamo che il valore della sua posizione sia -1):

```

// file RicercaBin.java
public class RicercaBin
{
    static int ricBin
        (int[] v, int infe, int supe, int k)
    {
        if (supe < infe) return -1; // vettore vuoto
        int medio = (infe + supe)/2;
    }
}

```

```

        if (k == v[medio]) return medio; // trovato
        if (k < v[medio]) return
            ricBin(v, infe, medio-1, k);
        return ricBin(v, medio+1, supe, k);
    }
    public static void main(String[] args)
    {   int pos; int[] vett = new int[8];
        Console.scriviStringa("Scrvi 8 interi:");
        for(int i=0; i<vett.length; i++)
            vett[i] = Console.leggiIntero();
        pos = ricBin(vett, 0, 7, 5);
        if (pos != -1) Console.scriviIntero(pos);
        else Console.scriviStringa("Non trovato");
    }
}

```

Un esempio di esecuzione è il seguente:

```

Scrivi 8 interi:
1 3 4 6 8 9 10 13
Non trovato

```

## 6.4. Classe *String*

Una stringa è una sequenza di caratteri, che può anche non avere alcun carattere (stringa vuota).

In Java esiste come classe il tipo *String* (che chiameremo tipo stringa) (package *java.lang*): una variabile appartenente a questo tipo, detta *variabile stringa*, rappresenta il riferimento di un *oggetto stringa* che memorizza una stringa costante (Figura 6.9).

La definizione di una variabile stringa avviene secondo le regole valide per le variabili di un tipo primitivo, ossia nel seguente modo:

```

string-variable-definition
    String variable-specifier-list ;

```

Un oggetto stringa viene creato (e allocato in memoria dinamica) per mezzo dell'operatore *new* seguito da un *costruttore*, nel seguente modo:

*string-object-creation-expression*  
**new String** ( *string-expression***|opt** )

dove l'espressione stringa deve produrre una stringa o un array di caratteri (esistono anche altre possibilità). Se l'espressione stringa non è presente, viene creato un oggetto contenente una stringa vuota

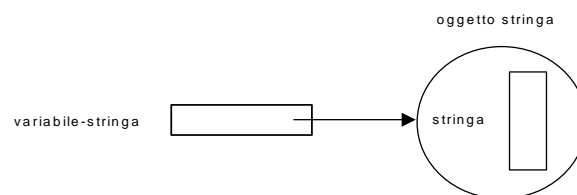


Figura 6.9. Variabile stringa e oggetto stringa

Per esempio, si può scrivere:

```
int n = ...;
char[] ac = new char[n];
...
String s1, s2 = new String();
...
String s3 = new String(s2), s4 = new String(ac);
```

Un letterale stringa (stringa racchiusa fra doppi apici) viene convertito dal compilatore in un oggetto stringa, riferito da una variabile stringa *anonima*. Pertanto, un oggetto *stringa* può essere costruito anche a partire da un letterale stringa:

```
String s4 = new String("abc");
```

#### 6.4.1. Assegnamento fra variabili stringa

Una variabile stringa può assumere come valore quello di un'altra

variabile stringa, anche anonima, come nel seguente esempio:

```
String s1 = s2;  
String s3;  
s3 = "abc";
```

Naturalmente, l'assegnamento coinvolge i riferimenti e non gli oggetti riferiti. Per esempio, l'effetto dell'ultimo assegnamento è illustrato in Fig. 6.10.

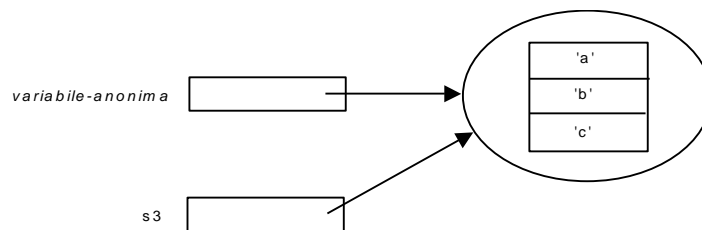


Figura 6.10. Assegnamento tra variabili stringa

#### 6.4.2. Concatenazione di stringhe

Per le variabili stringa è definito l'operatore di concatenazione '+', che ha il seguente effetto:

- se i due operandi sono di tipo stringa, l'operatore produce un nuovo oggetto stringa che memorizza una stringa ottenuta concatenando le due stringhe date;
- se un solo operando è di tipo stringa e l'altro di tipo diverso, quest'ultimo viene preliminarmente convertito al tipo stringa (in accordo a quanto illustrato nel sottoparagrafo 6.4.8).

Oltre alla creazione del nuovo oggetto stringa, l'operatore restituisce il suo indirizzo, che può essere assegnato a una variabile stringa.

I seguenti esempi mostrano quale stringa risultato viene ottenuta per concatenazione:

"abc" + "de"	stringa risultato: <i>abcde</i>
3 + 2 + "abc"	stringa risultato: <i>5abc</i>
"abc" + 3 + 2	stringa risultato: <i>abc32</i>
"abc" + b (b <i>boolean</i> di valore <i>true</i> )	stringa risultato: <i>abctrue</i>

L'esempio che segue, invece, illustra l'effetto completo dell'operatore di concatenazione:

```
String s1, s2, s3 = "ab";
s2 = s3 + "cd";
...
s1 = s2;
// s1 ed s2 riferiscono lo stesso oggetto,
// che memorizza la stringa abcd
```

### 6.4.3. Lunghezza di una stringa

La lunghezza di una stringa è determinata dal valore restituito dalla funzione *length()*, membro della classe *String*, applicata alla variabile stringa considerata:

***int* actualString.length ()**

Per esempio, si può scrivere:

```
String s;
...
int n = s.length();
int m = (s+"ab").length();
```

### 6.4.4. Uguaglianza fra stringhe

Per confrontare le stringhe memorizzate in due oggetti stringa si utilizza la funzione *equals()*, membro della classe *String*, applicata alla variabile stringa considerata:

***boolean* actualString.equals ( *String* s )**

Per esempio, si può scrivere:

```
String s1 = "ab", s2 = "a", s3;  
boolean b1, b2;  
...  
b1 = s1.equals(s2 + "b");           // b1 vale true  
b2 = (s1+"cd").equals(s2 + "bcd");  // b2 vale true
```

#### 6.4.5. Selezione di un carattere

I caratteri che compongono una stringa di lunghezza  $n$  sono numerati da 0 a  $n-1$ . Per avere il carattere in posizione *indice* si usa la funzione *charAt()*, membro della classe *String*, applicata alla variabile stringa considerata:

```
char actualString.charAt ( int indice )
```

Per esempio, si può scrivere:

```
String s; char c;  
...  
c = s.charAt(3);
```

#### 6.4.6. Confronto fra stringhe

Le stringhe memorizzate in due oggetti stringa possono essere confrontate fra loro (lexicograficamente) per mezzo della funzione *compareTo()*, membro della classe *String*, applicata alla variabile stringa considerata:

```
int actualString.compareTo ( String s )
```

Il risultato è negativo, zero o positivo a seconda del risultato del confronto. Il confronto avviene in base alla codifica (unicode) dei caratteri, a partire dai caratteri più significativi: se le stringhe hanno lunghezza diversa, l'assenza di caratteri è inferiore alla presenza di un qualunque carattere. Per esempio, possiamo avere:



```
String s1 = "abc", s2 = "ab";  
int i = s1.compareTo(s2);  
// i vale 1: s1 è maggiore di s2
```

#### 6.4.7. Sottostringhe

Un nuovo oggetto stringa, che memorizza una sottostringa di quella memorizzata in un oggetto stringa dato, si ottiene con la funzione *substring()*, membro della classe *String*, applicata alla variabile stringa considerata:

```
String actualString.substring ( int indiceIniziale , int indiceFinale )
```

La sottostringa va dal carattere di indice *indiceIniziale* al carattere di indice *indiceFinale-1*. Per esempio, si può scrivere:

```
String s1 = "abcd";  
String s2 = s1.substring(1,3);  
// s2 memorizza la stringa bc
```

#### 6.4.8. Conversione di altri tipi in stringhe

La classe *String* contiene alcune funzioni che consentono di trasformare in stringhe valori di tipi primitivi, di array di caratteri, o di oggetti di qualunque altro tipo:

```
String String.valueOf ( boolean b )  
String String.valueOf ( int i )  
String String.valueOf ( long l )  
String String.valueOf ( char c )  
String String.valueOf ( float f )  
String String.valueOf ( double d )  
String String.valueOf ( char[] ac )  
String String.valueOf ( Object obj )
```

Per quanto riguarda l'ultimo caso (le variabili classe sono riferimenti), se *obj* vale *null* viene restituita la stringa "null", altrimenti viene restituita la stringa prodotta dalla funzione *toString()* (presente in tutte le classi, come illustrato nel Capitolo 10).

## 6.5. Array di caratteri e stringhe

Un oggetto stringa è cosa diversa da un oggetto array di caratteri, in quanto il primo contiene un aggregato costante, mentre il secondo contiene un aggregato che ha solo il numero di elementi costante, mentre gli elementi possono cambiare valore. Naturalmente, sia una variabile stringa che una variabile array di caratteri possono contenere riferimenti differenti (rispettivamente, di oggetti stringa e di oggetti array di caratteri), e pertanto possono riferire aggregati di lunghezza differente l'uno dall'altro.

Un oggetto stringa può essere creato a partire da un oggetto array di caratteri, come indicato nel paragrafo 6.3. Inoltre, esiste la possibilità di convertire in un oggetto stringa un oggetto array di caratteri, come descritto nel sottoparagrafo precedente, oppure di convertire in un oggetto array di caratteri un oggetto stringa, con la funzione *toCharArray()*, membro della classe *String*, applicata alla variabile stringa considerata:

***char[] actualString.toCharArray ()***

Per esempio, possiamo avere:

```
char[] ac = { 'a', 'b', 'c' };
String s = new String(ac);
ac[0] = '0';
s = String.valueOf(ac);
// la stringa riferita da s vale "0bc"
s += "d";
ac = s.toCharArray();
// gli elementi di ac sono '0', 'b', 'c', 'd'
```

## 6.6. Istruzioni *switch* e *for*

L'istruzione *switch* può anche avere, come condizione, un'espressione di tipo enumerazione (che non deve produrre il valore *null*), e in questo caso le espressioni costanti (*case-label*) relative alle varie alternative

devono essere enumeratori (e non identificatori qualificati). Per illustrare quanto detto, consideriamo il seguente esempio:

```
// file SwitchEnum.java
enum Sem { verde, giallo, rosso };

public class SwitchEnum
{   public static void main(String[] args)
    {   int n; Sem s = null;
        Console.scriviStringa
            ("Scrivi un intero, 1, 2 o 3:");
        n = Console.leggiIntero();
        switch (n)
        {   case 1: s = Sem.verde; break;
            case 2: s = Sem.giallo; break;
            case 3: s = Sem.rosso; break;
            default: Console.scriviStringa
                ("Errore in ingresso");
                System.exit(1);
        }
        // ...
        switch (s)
        {   case verde:
            Console.scriviStringa("Avanti"); break;
            case giallo:
            Console.scriviStringa("Attenzione"); break;
            case rosso:
            Console.scriviStringa("Alt");
        }
    }
}
```

Una tipica esecuzione è la seguente:

```
Scrivi un intero, 1, 2 o 3:
3
Alt
```

L'istruzione *for*, oltre a quanto visto nel Capitolo 4, prevede una forma ulteriore (detta *for-each*), che scorre tutti gli elementi di un oggetto array o di una collezione iterabile (le *collezioni* non sono prese in esame in questo volume). Per gli oggetti array l'istruzione *for-each* ha la seguente sintassi:

*for-each-instruction*  
**for** ( *element-type identifier : variabile-array* ) *istruzione*

L'esecuzione avviene facendo assumere in sequenza alla variabile *identifier* i valori di tutti gli elementi dell'oggetto array riferito da *variabile-array* (l'identificatore di variabile è visibile solo nell'istruzione che costituisce il corpo del *for*). Tutto avviene come se l'istruzione fosse scritta nel seguente modo (*ar* è la variabile array):

**for** (int i = 0; i < ar.length; i++) **istruzione**

e nell'istruzione, invece dell'identificatore del generico elemento, si usasse *ar[i]*.

Per esempio, si può scrivere il seguente programma:

```
// file ForArray.java
public class ForArray
{
    public static void main(String[] args)
    {
        double[] ar = { 1.1, 2.2, 3.3 };
        for (double d: ar)
            Console.scriviReale(d);
    }
}
```

Tutto avviene come se il programma fosse scritto nel seguente modo:

```
public class ForArray1
{
    public static void main(String[] args)
    {
        double[] ar = { 1.1, 2.2, 3.3 };
        for (int i=0; i<ar.length; i++)
            Console.scriviReale(ar[i]);
    }
}
```

La forma *for-each* può essere usata anche con variabili di un tipo enumerazione, in quanto esiste la funzione *values()* che applicata a un tipo enumerazione restituisce un array di enumeratori. Per esempio, si può scrivere il seguente programma:

```
// file ForEnum.java
enum Sem { verde, giallo, rosso };

public class ForEnum
{   public static void main(String[] args)
    {   for (Sem s: Sem.values())
        Console.scriviStringa(s.name());
        // Sem[] ss = Sem.values();
        // for (Sem s: ss)
        //     Console.scriviStringa(s.name());
    }
}
```

Analogamente, la forma *for-each* può essere applicata anche a variabili di un tipo stringa, in quanto la funzione *toCharArray()* applicata a una variabile di questo tipo restituisce un array di caratteri. Per esempio, si può scrivere il seguente programma:

```
// file ForString.java
public class ForString
{   public static void main(String[] args)
    {   String st = new String("Ciao");
        for (char c: st.toCharArray())
            Console.scriviCarattere(c);
        // char[] cc = st.toCharArray();
        // for(char c:cc)
        //     Console.scriviCarattere(c);
    }
}
```

## 7. Classi

### 7.1. Classi, variabili e oggetti

Una *classe* è un modello che descrive una certa categoria di oggetti. Essa comprende un certo numero di membri, alcuni dei quali (*variabili membro* o *campi dati*) rappresentano lo stato degli oggetti, altri (*funzioni membro* o *metodi*) costituiscono le operazioni sugli oggetti stessi, altri ancora (*costruttori*) sono funzioni membro aventi lo stesso nome della classe, utilizzate per la creazione degli oggetti di tale classe. Una classe rappresenta quindi *il tipo* di una categoria di oggetti.

Infine (questo aspetto verrà approfondito nel Capitolo 9), una classe può avere come membri anche dei tipi, enumerazione, classe o interfaccia.

Per esempio, la definizione di una classe è la seguente:

```
class MiaClasse
{ // tipi
  // ...
  // campi dati
  int n;
  // ...
  // costruttori
  MiaClasse()
  { /* ... */ }
  // ...
  // metodi
  int fai(int a)
  { /* ... */ }
```

```
// ...  
}
```

Una *variabile di un tipo classe* (brevemente, *variabile classe*) è un riferimento di oggetti di quella classe, e si definisce in modo simile alle variabili degli altri tipi. Per esempio, si può avere:

```
MiaClasse var, var1;  
...  
Miaclasse var2 = var;
```

Un *oggetto di un tipo classe* (brevemente, *oggetto classe*) è un'istanza della classe (oggetto istanza): viene ottenuto per mezzo dell'operatore *new* seguito dalla chiamata di un costruttore (nel caso più semplice, un costruttore senza parametri), e viene allocato in memoria dinamica. L'operatore restituisce l'indirizzo dell'oggetto allocato, che può essere 1) assegnato a una variabile classe già definita o 2) usato come iniziatore di una nuova variabile classe. Per esempio, si può avere (la Fig. 7.1 si riferisce alla variabile *vara*):

```
MiaClasse vara;  
vara = new MiaClasse();  
MiaClasse varb = new MiaClasse();
```

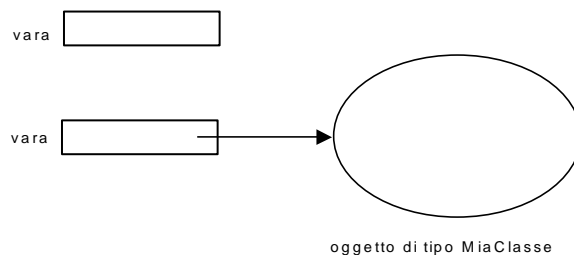


Figura 7.1. Variabile classe e oggetto classe.

I tipi enumerazione e i tipi array visti nel Capitolo 6 hanno proprietà simili a quelle delle classi.

## 7.2. Forma sintattica di una classe

Una classe (primaria, ossia non membro di un'altra classe) si definisce in accordo alle seguenti regole:

```

primary-class-definition
    publicopt class-definition
class-definition
    class-heading class-body
class-heading
    class identifier
class-body
    { member-definition-seqopt }
member-definition
    member-derived-type-definition
    member-variable-definition
    constructor-definition
    method-definition

```

L'eventuale modificatore che compare nella definizione della classe (assieme ad analoghi modificatori che si hanno nella definizione dei membri) verrà esaminato nel paragrafo 7.4.

In una classe possono essere definiti tipi derivati, tipicamente altre classi (classi annidate), che normalmente vengono utilizzati nella classe stessa:

```

member-derived--type-definition
    derived-type-modifier-seqopt type-definition
type-definition
    enum-definition
    class-definition
    interface-definition

```

Le variabili membro determinano lo stato degli oggetti della classe, e sono dette anche *variabili istanza*: esse hanno valore iniziale default (0 se di tipo numerico, *false* se di tipo booleano, *null* se di un tipo derivato), ma possono avere, essendo sintatticamente definizioni, anche inizializzatori espliciti:

```

member-variable-definition
    variable-modifier-seqopt variable-definition

```



Un costruttore è una particolare funzione membro avente lo stesso nome della classe, che non prevede la specifica del tipo del risultato (neppure la specifica *void*):

```

constructor-definition:
    constructor-modifier-seq|opt constructor-header constructor-body
constructor-header
    class-identifier ( formal-parameter-list|opt )
constructor-body
    block

```

Il compito di un costruttore è quello di assegnare ai campi dati dell'oggetto creato valori specificati all'atto della costruzione. In una classe, possono esserci più costruttori, in virtù del meccanismo di overloading.

Una metodo ha la forma di una funzione (funzione membro), e rappresenta un'operazione definita sugli oggetti della classe:

```

method-definition
    method-modifier-seq|opt function-specification

```

Anche i metodi possono far uso del meccanismo di overloading.

Dal punto di vista sintattico, i costruttori e i metodi possono essere compresi in un'unica categoria, differendo soltanto per la specifica o meno del tipo del risultato. I parametri formali dei costruttori e dei metodi e il risultato dei metodi possono essere di tipo qualsivoglia (anche di un tipo classe).

### 7.3. Variabili classe e oggetti classe

La definizione di variabili classe si effettua utilizzando l'identificatore della classe come tipo:

```

class-variable-definition
    class-identifier variable-specifier-list ;

```

La creazione di oggetti classe (*oggetti istanza* o *istanze della classe*) si ottiene in accordo alla seguente regola sintattica (ogni costruttore ha per identificatore quello della classe):

*class-object-creation*  
**new** *constructor-identifier* ( *actual-parameter-list* **opt** )

Quando viene creato un oggetto, viene allocata in memoria dinamica una copia delle variabili membro definite nella classe (o variabili istanza) e vengono valutati gli eventuali inizializzatori (espressioni): le variabili istanza assumono tutte un valore iniziale, o quello default (0 se di tipo numerico, *false* se di tipo booleano, *null* se di tipo derivato), o quello determinato dagli inizializzatori. Viene quindi invocato un costruttore (come specificato dall'operatore *new*), che effettua ulteriori inizializzazioni, facendo assumere allo stato dell'oggetto creato un valore dipendente dalle elaborazioni effettuate dal costruttore stesso.

Una classe può non prevedere esplicitamente costruttori: in questo caso opera il cosiddetto costruttore default implicito, che non ha parametri formali e non esegue alcuna operazione. Se nella classe sono definiti costruttori, deve essere sempre invocato uno di essi, e non è più disponibile il costruttore default implicito: pertanto, nel caso in cui tutti i costruttori definiti prevedano parametri formali, non è lecito creare un oggetto classe senza la specifica di argomenti attuali.

Come esempio, consideriamo il seguente programma:

```
class TuaClasse
{ int a = 1;
  int b = a++;
  TuaClasse(int n)
  { a = n;
    //
  }
  // ...
}

class ProvaT
{ public static void main(String[] args)
  { // ...
    TuaClasse mc1 = new TuaClasse(5);
    // per l'oggetto riferito da mc1
    // a vale 5 e b vale 2
  }
}
```

```

    TuaClasse mc2 = new TuaClasse(10);
    // per l'oggetto riferito da mc2
    // a vale 10 e b vale 2
    TuaClasse mc3 = new TuaClasse(); // errore
    //
}
}

```

Per quanto riguarda i costruttori e i metodi, tutto avviene come se una loro copia venisse a far parte dell'oggetto creato (in realtà, questi esistono in copia unica e vengono mandati in esecuzione mediante meccanismi di chiamata di funzione, e il riferimento dell'oggetto a cui si applicano costituisce un parametro aggiuntivo nascosto).

### 7.3.1. Oggetto implicito

In una classe si possono utilizzare tutti i membri definiti nella classe stessa. In particolare, gli inizializzatori, i costruttori e i metodi possono utilizzare le variabili istanza: in questo caso, si intende che queste siano relative all'oggetto implicito (generico oggetto di quella classe). Analogamente, gli inizializzatori, i costruttori e i metodi possono invocare metodi, e in questo caso si applicano all'oggetto implicito.

### 7.3.2. Selezione dei membri di un oggetto classe

Dato un oggetto classe, si possono individuare i tipi e le variabili dell'oggetto o eseguire i metodi dell'oggetto mediante un riferimento dell'oggetto e l'operatore di selezione, nel seguente modo:

```

class-object-member
  class-variable . member-derived-type-identifier
  class-variable . member-variable-identifier
  class-variable . method-identifier ( expression-list )

```

L'operatore di selezione nel primo caso restituisce un nome di tipo, nel secondo un nome di variabile, nel terzo un valore (il risultato prodotto dal metodo, se questo non è void). Se una variabile membro è di un tipo derivato, ad essa si può a sua volta applicare l'operatore di selezione o quello di indicizzazione, e così via.

A titolo di esempio, si consideri il seguente programma:

```
class SuaClasse
{ // ...
  int c;      // valore iniziale 0
  SuaClasse()
  {}
  void fai()
  { c++; }
  // ...
}

public class ProvaS
{ public static void main(String[] args)
  { // ...
    SuaClasse o1 = new SuaClasse(),
              o2 = new SuaClasse();
    // o1.c vale 0 e o2.c vale 0
    o1.fai();
    // o1.c vale 1 e o2.c vale 0
  }
}
```

## 7.4. Visibilità e modificatori di accesso

L'identificatore della classe e quello dei membri della classe sono sempre visibili all'interno della classe: gli inizializzatori delle variabili istanza possono contenere solo identificatori già definiti, mentre i corpi delle classi annidate, dei costruttori e dei metodi possono contenere qualsiasi identificatore, a prescindere dal fatto che la sua definizione preceda o meno il suo utilizzo (regole di visibilità di classe). A titolo di esempio, è possibile definire la seguente classe:

```
class UnaClasse
{ void fai(int n)
  { a += n;
    // ...
  }
  UnaClasse()
```

```
{ a = 10;  
  //  
}  
int a = 1;  
int b = a++;  
// ...  
}
```

Una classe e i suoi membri possono avere modificatori di accesso: questi possono indicare specifiche di visibilità, che peraltro hanno effetto solo al di fuori della classe stessa. In questo paragrafo considereremo solo alcuni modificatori di visibilità, rimandando al paragrafo 7.13 una loro trattazione più completa (raggruppamento di classi in *package*).

Una classe può avere il modificatore *public*, che la rende visibile al di fuori del package a cui appartiene. Nel caso semplice in cui tutte le classi facciano parte di un unico package, tale modificatore non si rende pertanto necessario.

Un membro di una classe può avere uno dei modificatori *public* o *private*: nel caso semplice in cui tutte le classi appartengono a un'unico package, tali modificatori hanno il seguente significato:

- modificatore *public* (o assenza di modificatore): membro visibile alle altre classi;
- modificatore *private*: membro non visibile alle altre classi.

Notare che il metodo *main()* deve necessariamente contenere il modificatore *public*, per essere visibile dalla piattaforma Java, anche al di fuori del package a cui appartiene la classe principale. Normalmente, noi useremo tale modificatore anche per la classe principale, non essendo immediato comprendere come al di fuori del package possano esservi metodi visibili di classi non visibili. Quanto detto è tuttavia possibile, come sarà chiarito nel Capitolo 10.

## 7.5. Esempi

Come primo esempio, con riferimento ai punti di un piano cartesiano,

consideriamo la seguente definizione di classe:

```
// file Punto.java
class Punto
{ // stato
    private double x, y;
    // costruttore
    Punto (double a, double b )
    { x = a;
      y = b;
    }
    // metodo: calcola la distanza dall'origine
    double distanza()
    { return Math.sqrt(x*x + y*y);
    }
    // metodo: trasla le coordinate del punto
    // della quantita` t
    void traslazione(double t)
    { x += t;
      y += t;
    }
}
```

Gli oggetti della classe *Punto* possono essere creati fuori della classe medesima (in un'altra classe), perché il costruttore è visibile. Gli oggetti creati al di fuori possono utilizzare i metodi *distanza()* e *traslazione()*, in quanto visibili, mentre non possono utilizzare le variabili *x* e *y* in quanto private. Per esempio, si può scrivere un programma comprendente, oltre che la precedente, anche la seguente classe (vedi anche la Fig. 7.2):

```
// file ProvaPunto.java
public class ProvaPunto
{ public static void main(String[] args)
  { double d;
    Punto p1 = new Punto(1, 2); // azione 1)
    d = p1.distanza();           // azione 2)
    Console.scriviReale(d);
    Punto p2 = p1;               // azione 3)
    // ...
    p2.traslazione(2.5);
    Console.scriviReale(p2.distanza());
    // ...
  }
}
```

```

    }
}

```

In esecuzione si ha il seguente risultato:

```

2.23606797749979
5.70087712549569

```

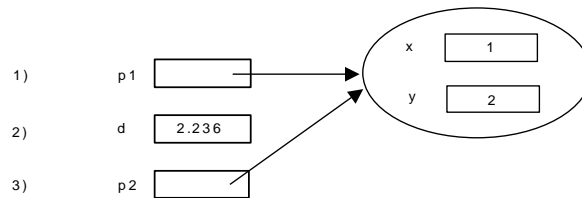


Figura 7.2. Azioni 1), 2) e 3) della classe *ProvaPunto*.

Come secondo esempio, consideriamo il caso dei poligoni regolari, per i quali intendiamo calcolare perimetro e area. Per ogni poligono, vengono specificati sia il numero di lati che la lunghezza di un lato. Definiamo quindi la seguente classe:

```

// file PolRegolare.java
class PolRegolare
{ private double lato;
  private int numLati;
  private double apotema()
  { // metodo di lavoro
    double alfa = Math.PI/numLati;
    // meta` angolo di ogni triangolo
    return lato/2 * Math.cos(alfa)/Math.sin(alfa);
  }
  PolRegolare(double l, int n)
  { // costruttore
    lato = l; numLati = n;
  }
  double perimetro()
  { // metodo per calcolare il perimetro
    return lato * numLati;
  }
}

```

```
double area()
{ // metodo per calcolare l'area,
  // che utilizza il metodo di lavoro apotema()
  double ap = apotema();
  return lato/2 * ap * numLati;
}
```

Tale classe può essere utilizzata nel seguente modo:

```
// file ProvaPolRegolare.java
public class ProvaPolRegolare
{ public static void main(String[] args)
  { int nn; double ll;
    Console.scriviStringa
      ("Numero di lati del poligono:");
    nn = Console.leggiIntero();
    Console.scriviStringa
      ("Lunghezza del lato del poligono:");
    ll = Console.leggiReale();
    PolRegolare pp = new PolRegolare(ll, nn);
    Console.scriviReale(pp.perimetro() );
    Console.scriviReale(pp.area() );
  }
}
```

Un esempio di esecuzione è il seguente:

```
Numero di lati del poligono:
8
Lunghezza del lato del poligono:
5
40
120.71067811865476
```

Come terzo esempio, consideriamo una classe *Direzione*: essa rappresenta, in uno spazio a tre dimensioni, segmenti orientati (vettori in matematica) che hanno un vertice nell'origine e l'altro vertice in un punto di coordinate maggiori di 0. La rappresentazione interna di un vettore viene fatta in coordinate cartesiane (coordinate x, y e z del punto terminale). Il costruttore permette di costruire un vettore a partire dalle



coordinate del punto terminale, e i metodi consentono di stampare il valore dell'angolo (in radianti) fra la proiezione del vettore sul piano  $xy$  e l'asse  $x$  (*angolo in scostamento*), e il valore dell'angolo (in radianti) fra il vettore e il piano  $xy$  (*angolo in altezza*). La definizione della classe è la seguente:

```
// file Direzione.java
class Direzione
{ // rappresentazione interna
  private double x, y, z;
  Direzione(double a, double b, double c)
  { // costruttore
    x = a; y = b; z = c;
  }
  void stampaAlfa()
  { // metodo: stampa l'angolo in scostamento
    double lun_xy = Math.sqrt(x*x + y*y);
    Console.scriviReale(Math.asin(y/lun_xy));
  }
  void stampaGamma()
  { // metodo: stampa l'angolo in altezza
    double lun = Math.sqrt(x*x+y*y+z*z);
    Console.scriviReale(Math.asin(z/lun));
  }
}
```

Tale classe può essere utilizzata nel seguente modo:

```
// file ProvaDirezione.java
public class ProvaDirezione
{ public static void main(String[] args)
  { Console.scriviStringa ("Scrivi tre coordinate:");
    double aa = Console.leggiReale(),
          bb = Console.leggiReale(),
          cc = Console.leggiReale();
    Direzione dir = new Direzione(aa, bb, cc);
    Console.scriviStringa ("Angoli alfa e gamma:");
    dir.stampaAlfa(); dir.stampaGamma();
  }
}
```

Un esempio di esecuzione è il seguente:

```

Scrivi 3 coordinate:
1 1 1
Angoli alfa e gamma:
0.7853981633974482
0.6154797086703874

```

Notare che nell'esempio *alfa* vale  $45^0$ , mentre *gamma* vale di meno.

Come quarto esempio, si riporta un programma per il calcolo delle radici di un'equazione di secondo grado, contenente una classe *Risultato* (classe con soli campi dati: un membro intero *quante* che specifica il numero delle radici (vale  $-1$  se l'equazione è degenere), e due campi reali *rad1* e *rad2* che contengono le eventuali radici) e una classe *Equazione* con una funzione membro il cui risultato è di tipo *Risultato*:

```

// file Equazione.java
class Risultato
{ int quante; double rad1, rad2; }

class Equazione
{ private double aa, bb, cc;
  Equazione(double a, double b, double c)
  { aa = a; bb = b; cc = c; }
  Risultato calcolaRad()
  { double delta; Risultato rr = new Risultato();
    if ((aa==0)&&(bb==0)) rr.quante = -1;
    else if (aa==0)
    { rr.quante = 1;
      rr.rad1 = -cc/bb;
    }
    else
    { delta = bb*bb - 4*aa*cc;
      if (delta<0) rr.quante = 0;
      else
      { delta = Math.sqrt(delta);
        rr.quante = 2;
        rr.rad1 = (-bb+delta)/(2*aa);
        rr.rad2 = (-bb-delta)/(2*aa);
      }
    }
    return rr;
  }
}

```

```
// file ProvaEquazione.java
public class ProvaEquazione
{ public static void main(String[] args)
  { double ra, rb, rc; Equazione eq; Risultato ris;
    Console.scriviStringa("Coefficienti:");
    ra = Console.leggiReale();
    rb = Console.leggiReale();
    rc = Console.leggiReale();
    eq = new Equazione(ra, rb, rc);
    ris = eq.calcolaRad();
    switch (ris.quante)
    { case -1:
      Console.scriviStringa("Equazione degenera");
      break;
      case 0:
      Console.scriviStringa("Nessuna soluzione");
      break;
      case 1:
      Console.scriviStringa
        ("Una radice: " + ris.rad1);
      break;
      case 2:
      Console.scriviStringa
        ("Due radici: " + ris.rad1 + ' ' + ris.rad2);
    }
  }
}
```

Un esempio di esecuzione è il seguente:

```
Coefficienti:
1 -2 1
Due radici: 1.0 1.0
```

### 7.5.1. Pile

Una *pila* è un insieme ordinato di dati di ugual tipo, in cui è possibile effettuare operazioni di inserimento e di estrazione secondo la seguente regola di accesso: l'ultimo dato inserito è il primo ad essere estratto (regola

di accesso LIFO: Last In First Out). Su una pila vengono generalmente definite operazioni per verificare le condizioni di pila piena e pila vuota, per effettuare inserimenti ed estrazioni, e per stampare il valore degli elementi in essa contenuti.

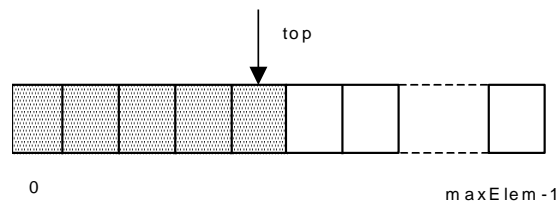


Figura 7.3. Una pila

Una pila può essere realizzata con un array e con un indice *top* (detto impropriamente puntatore): l'array memorizza i dati nelle sue componenti e il puntatore *top* individua in ogni istante la posizione relativa all'ultimo inserimento (il livello di riempimento della pila). Un inserimento avviene nella componente dell'array successiva a quella individuata da *top*, mentre un'estrazione riguarda la componente dell'array individuata da *top* stesso (Fig. 7.3).

La pila è vuota quando il puntatore ha un valore che precede di 1 quello che individua la prima componente dell'array, mentre la pila è piena quando il puntatore individua l'ultima componente dell'array stesso. Il numero di elementi contenuti nella pila è dato dal valore di *top* aumentato di uno.

In accordo a questa modalità realizzativa, un tipo pila può essere ottenuto con la seguente classe *PilaArray*:

```
// file PilaArray.java
class PilaArray
{ private int top = -1, maxElem;
  private int[] vett;
  PilaArray(int n)
  { maxElem = n; vett = new int[maxElem]; }
  boolean empty()
  { if (top == -1) return true;
```

```

        return false;
    }
    boolean full()
    { if(top == maxElem-1) return true;
      return false;
    }
    void push(int s)
    { if (top < maxElem-1) vett[++top] = s;
    }
    int pop()
    { int s = 0;
      if (top > -1) s = vett[top--];
      return s;
    }
    void stampa()
    { if (empty()) Console.scriviStringa("Pila vuota");
      else
      { Console.scriviStringa
        ("Numero di elementi: " + (top+1));
        Console.scriviStringa("Valori:");
        for(int i=0; i<=top; i++)
            Console.scriviInt(vett[i]);
        Console.nuovaLinea();
      }
    }
}

// file ProvaPilaArray.java
public class ProvaPilaArray
{ public static void main(String[] args)
  { int n, m, i;
    Console.scriviStringa
      ("Scrivi la dimensione della pila:");
    m = Console.leggiIntero();
    PilaArray pp = new PilaArray(m);
    Console.scriviStringa
      ("Scrivi quanti elementi vuoi inserire:");
    m = Console.leggiIntero();
    Console.scriviStringa
      ("Scrivi " + m + " numeri:");
    for (i = 0; i<m; i++)
    { n = Console.leggiIntero();
      if (!pp.full())

```

```
        { pp.push(n);  
          Console.scriviStringa("Inserito " + n);  
        }  
        else Console.scriviStringa  
              ("Inserimento impossibile");  
      }  
      Console.scriviStringa  
        ("Scrivi quanti elementi vuoi estrarre");  
      m = Console.leggiIntero();  
      for (i = 0; i<m; i++)  
        if (!pp.empty())  
        { n = pp.pop();  
          Console.scriviStringa("Estratto " + n);  
        }  
        else Console.scriviStringa  
              ("Estrazione impossibile");  
      Console.scriviStringa("Contenuto della pila:");  
      pp.stampa();  
    }  
  }  
}
```

Una esecuzione tipica è la seguente:

```
Scrivi la dimensione della pila:  
5  
Scrivi quanti numeri vuoi inserire  
5  
Scrivi 5 numeri  
1 2 3 4 5  
Inserito 1  
Inserito 2  
Inserito 3  
Inserito 4  
Inserito 5  
Scrivi quanti numeri vuoi estrarre  
2  
Estratto 5  
Estratto 4  
Contenuto della pila:  
Numero di elementi: 3  
Valori degli elementi:  
3 2 1
```

### 7.5.2. Code

Una *coda* è un insieme ordinato di dati di ugual tipo, in cui è possibile effettuare operazioni di inserimento e di estrazione secondo la seguente regola di accesso: il primo dato inserito è il primo ad essere estratto (regola di accesso FIFO: First In First Out). Su una coda vengono generalmente definite operazioni per verificare le condizioni di coda piena e coda vuota, per effettuare inserimenti ed estrazioni, e per stampare il valore degli elementi in essa contenuti.

Una coda può essere realizzata con un array e con due indici *back* e *front* (detti impropriamente puntatori): l'array memorizza i dati nelle sue componenti, il puntatore *back* individua in ogni istante la posizione relativa al prossimo inserimento e *front* la posizione relativa alla prossima estrazione. Dopo ciascuna delle due operazioni, il puntatore interessato viene incrementato in modo circolare (Fig. 7.4).

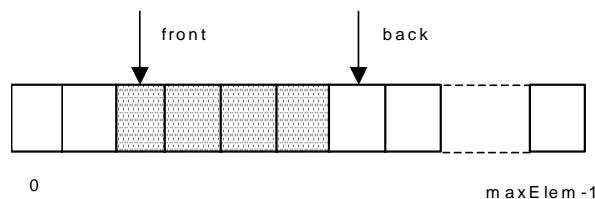


Figura 7.4. Una coda

Il numero di elementi contenuti nella coda può essere determinato dai valori di *back* e di *front*: tuttavia, è preferibile prevedere per questo numero una variabile apposita *quanti*, poiché i due puntatori da soli non consentono di discriminare un array vuoto da un array pieno, avendo valore coincidente in entrambi i casi.

In accordo a questa modalità realizzativa, un tipo coda può essere ottenuto con la seguente classe *CodaArray*:

```
// file CodaArray.java
class CodaArray
{ private int front, back, quanti, maxElem;
  private int[] vett;
```

```
CodaArray(int n)
{ maxElem = n; vett = new int[maxElem]; }
boolean empty()
{ if (quanti == 0) return true;
  return false;
}
boolean full()
{ if (quanti == maxElem) return true;
  return false;
}
void enqueue(int s)
{ if (quanti != maxElem)
  { vett[back] = s; quanti++;
    back = (back+1)%vett.length;
  }
}
int dequeue()
{ int s = 0;
  if (quanti != 0)
  { s = vett[front]; quanti--;
    front = (front+1)%vett.length;
  }
  return s;
}
void stampa()
{ if (empty()) Console.scriviStringa("Coda vuota");
  else
  { Console.scriviStringa
    ("Numero di elementi: " + quanti);
    Console.scriviStringa("Valori:");
    int k = front;
    for(int i = 0; i < quanti; i++)
    { Console.scriviInt(vett[k]);
      k=(k+1)%vett.length;
    }
    Console.nuovaLinea();
  }
}

// file ProvaCodaArray.java
public class ProvaCodaArray
{ public static void main(String[] args)
```



```

{ int n, m, i;
  Console.scriviStringa
    ("Scrivi la dimensione della coda:");
  m = Console.leggiIntero();
  CodaArray ll = new CodaArray(m);
  Console.scriviStringa
    ("Scrivi quanti elementi vuoi inserire:");
  m = Console.leggiIntero();
  Console.scriviStringa
    ("Scrivi " + m + " numeri:");
  for (i = 0; i < m; i++)
  { n = Console.leggiIntero();
    if (!ll.full())
    { ll.enqueue(n);
      Console.scriviStringa("Inserito " + n);
    }
    else Console.scriviStringa
      ("Inserimento impossibile");
  }
  Console.scriviStringa
    ("Scrivi quanti elementi vuoi estrarre");
  m = Console.leggiIntero();
  for (i = 0; i < m; i++)
  { if (!ll.empty())
    { n = ll.dequeue();
      Console.scriviStringa("Estratto " + n);
    }
    else Console.scriviStringa
      ("Estrazione impossibile");
  }
  Console.scriviStringa("Contenuto della coda:");
  ll.stampa();
}
}

```

## 7.6. Classi e tipi di dato

Le classi consentono di effettuare astrazioni sui dati, permettendo di

realizzare i cosiddetti di *tipi di dato*.

Un tipo di dato è costituito da un insieme di valori (elementi del tipo) e da un gruppo di operazioni (operazioni proprie del tipo), realizzate utilizzando direttamente la modalità di rappresentazione interna degli elementi. La rappresentazione interna degli elementi non è accessibile all'utente (è incapsulata nella realizzazione del tipo): l'utente può definire per quel tipo elaborazioni aggiuntive (esprese, per esempio, mediante funzioni), ma per definirle, non può fare uso della rappresentazione interna degli elementi, ma può utilizzare unicamente le operazioni proprie del tipo. Pertanto la modalità di rappresentazione degli elementi influenza solo la realizzazione delle operazioni proprie del tipo, ma non le eventuali operazioni aggiuntive.

I tipi fondamentali rispettano in modo completo il concetto di tipo di dato. Per esempio, il tipo *int* ha come valori gli interi compresi fra  $+(2^{31}-1)$  e  $-2^{31}$ , e possiede come operazioni proprie la somma (+ binario), la sottrazione (- binario), eccetera. Le operazioni proprie sono realizzate facendo uso della rappresentazione interna degli elementi, effettuata con 32 bit in complemento a 2. Eventuali elaborazioni aggiuntive (per esempio, radice quadrata) non possono far uso della rappresentazione degli elementi, ma possono utilizzare solo le operazioni proprie.

Le classi consentono al programmatore di definire propri tipi realizzando compiutamente il concetto di tipo di dato. I possibili valori del tipo sono costituiti dallo stato della classe, e le operazioni proprie da metodi della classe. Utilizzando per i singoli campi dati il modificatore *private*, la rappresentazione interna degli oggetti non è accessibile all'utente. Utilizzando per i metodi che rappresentano le operazioni proprie il modificatore *public* (o nessun modificatore), le operazioni proprie stesse sono invece disponibili per l'utente. Il corpo dei metodi può fare liberamente uso dei campi dati, quindi della modalità con cui gli elementi del tipo sono rappresentati. Possono anche esserci metodi con il modificatore *private*, e questi rappresentano funzioni di utilità.

Un tipo di dato è *astratto* se i valori e le operazioni proprie prescindono dalle modalità di realizzazione interna degli elementi. Quindi, cambiando la rappresentazione interna, ma lasciando invariate le specifiche delle operazioni proprie, non si hanno ripercussioni sull'utente. Le classi consentono di effettuare buone realizzazioni di tipi di dato astratti, nel

senso che una classe può essere sostituita con un'altra con diversa rappresentazione interna degli elementi, ma con metodi aventi le stesse intestazioni (*interfacce* dei metodi). Per esempio, le pile e le code viste in precedenza possono essere realizzate anche con classi differenti, aventi metodi con la stessa intestazione, come sarà visto ne Capitolo 8.

I tipi di dato astratti vengono realizzati compiutamente mediante le classi astratte o le interfacce, come chiarito nel Capitolo 10.

## 7.7. Distruttori

In Java non esistono distruttori (come per esempio in C++), per cui non avviene (come per la creazione) una distruzione esplicita degli oggetti ad opera del programmatore. Il recupero della memoria dinamica non più utilizzata viene effettuata dalla piattaforma Java per mezzo di una specifica routine (*Garbage Collector*), che periodicamente libera quelle zone di memoria occupate da oggetti per i quali non esiste più alcuna variabile classe che li riferisce.

Si consideri, per esempio, il seguente spezzone di programma, che utilizza la classe *Punto* definita nel paragrafo 7.5:

```
Punto pp;  
pp = new Punto(2,3) ;  
pp = new Punto(4,5) ;
```

Dopo l'ultima istruzione, l'oggetto creato con *new Punto(2,3)* (distinto dall'oggetto creato con *new Punto(4,5)*) non ha più riferimenti, e può essere distrutto.

Il *Garbage Collector* può essere mandato esplicitamente in esecuzione dal programmatore, invocando il metodo *gc()* della classe *System* (package *java.lang*).

Prima di distruggere un oggetto, il Garbage Collector richiama il metodo *finalize()* della classe *Object* (package *java.lang*), che non esegue nessuna azione. Tale metodo può essere ridefinito per una data classe, facendogli compiere azioni opportune, come altre azioni di ripulitura o la introduzione di un nuovo riferimento per l'oggetto (*resurrection*).

## 7.8. Variabili e parametri

Come detto in precedenza, le variabili definite in una classe sono dette *variabili istanza*, in quanto descrivono lo stato di ogni oggetto istanza. Le loro proprietà possono essere così riassunte:

- se non hanno un iniziatore esplicito, hanno valore iniziale default;
- hanno visibilità di classe;
- con il modificatore di accesso *public* (o nessun modificatore) possono essere utilizzate da altre classi (tramite variabili di tipo classe e l'operatore di selezione);
- sono diverse da oggetto classe a oggetto classe;
- hanno lo stesso tempo di vita di un oggetto classe (sono dinamiche).

Le variabili definite in un costruttore o in un metodo sono dette *variabili locali*. Le loro proprietà possono essere così riassunte:

- se non hanno un iniziatore esplicito, hanno un valore iniziale non definito (non hanno valore iniziale default);
- hanno visibilità di blocco;
- non possono utilizzare modificatori di accesso;
- sono diverse per ogni invocazione del costruttore o del metodo;
- hanno un tempo di vita uguale a quello di esecuzione del costruttore o del metodo (sono automatiche).

I tipi delle variabili, se non sono predefiniti, possono essere definiti, oltre che con classi autonome, con classi annidate (membri della classe esterna) e con classi locali ai metodi.

I parametri formali dei costruttori e dei metodi hanno le stesse proprietà delle variabili locali (proprietà precedentemente elencate), con la differenza che per ogni chiamata essi vengono inizializzati con i valori dei corrispondenti argomenti attuali.

### 7.8.1. Riferimenti come parametri: modifica degli oggetti riferiti

Consideriamo il caso in cui un parametro formale di un metodo sia di

un tipo classe (riferimento). All'atto della invocazione del metodo, il riferimento attuale viene ricopiato in quello formale. Nel corpo del metodo, eventuali modifiche apportate al valore del parametro formale non vengono conservate, in quanto esso conclude il proprio tempo di vita con la fine dell'esecuzione del metodo stesso. Nel corpo del metodo, però, si possono eventualmente apportare modifiche allo stato dell'oggetto riferito: in questo caso, quando il metodo viene invocato, per mezzo del riferimento attuale cambia lo stato dell'oggetto attualmente riferito.

Come esempio, consideriamo il seguente programma (Fig. 7.5):

```
// file Modifica.java
class Num
{ int x;
  Num(int i)
  { x = i; }
}

class MM
{ Num a = new Num(5);
  void modifical(Num n)
  { n = a;
  }
  void modifica2(Num n)
  { n.x = a.x;
  }
}

public class Modifica
{ public static void main(String[] args)
  { Num n1 = new Num(1), n2 = new Num(2);
    MM m = new MM();
    m.modifical(n1);
    // modifica il riferimento n
    Console.scriviStringa
      ("Prima modifica: " + n2.x);
    m.modifica2(n2);
    // modifica n.x, ossia n2.x
    Console.scriviStringa
      ("Seconda modifica: " + n2.x);
  }
}
```

In esecuzione si ha il seguente risultato:

**Prima modifica: 2**  
**Seconda modifica: 5**

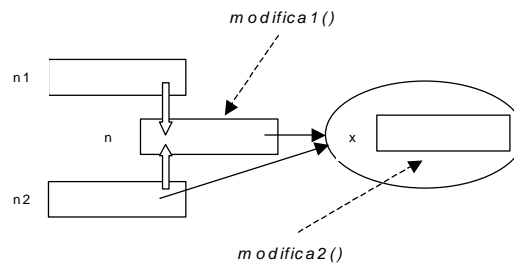


Figura 7.5. Effetto dei metodi *modifica1()* e *modifica2()*

### 7.8.2. Utilizzo dei riferimenti nelle classi e nei metodi

Quando si definisce una classe, si possono avere, come variabili istanza o variabili locali, variabili della stessa o di altra classe.

La specifica esplicita dell'eventuale valore iniziale di una variabile istanza non deve produrre (direttamente o indirettamente) la creazione di un oggetto della stessa classe: infatti, tale azione richiederebbe a sua volta la creazione di un oggetto della stessa classe, e così via, dando luogo a un errore per riempimento della memoria dinamica, errore che si manifesterebbe in fase di esecuzione, quando viene creato un oggetto e valutati gli inizializzatori delle variabili istanza.

Invece, in un metodo vi può essere la creazione di un oggetto della stessa classe: il metodo, tuttavia, non deve invocare in modo incondizionato il metodo stesso (direttamente o indirettamente) sull'oggetto creato, per non avere lo stesso inconveniente sopra menzionato.

A titolo di esempio, si consideri la seguente classe:

```
class Uno
{ // ...
    Uno aa;                // variabile istanza
```

```
    Uno bb = new Uno();    // errore
    // ...
    void f1()
    { Uno bb = new Uno(); // variabile locale
      // ...
      bb.f1();             // occorre una condizione
    }
    // ...
}
```

## 7.9. Riferimenti di oggetti creati in un metodo

Il risultato di un metodo può anche essere un riferimento, della stessa classe a cui il metodo appartiene o di un'altra classe, e riferire un oggetto creato nel corpo del metodo stesso e riferito da una variabile classe locale al metodo. Infatti, quando il metodo viene eseguito, inizia il tempo di vita della variabile classe locale al metodo, e viene creato un oggetto da essa riferito. Quando il metodo termina, viene restituito un riferimento dell'oggetto creato: cessa il tempo di vita della variabile classe locale al metodo che riferisce l'oggetto, ma l'oggetto creato rimane in vita in quanto allocato nella memoria dinamica. A titolo di esempio, si consideri il seguente programma:

```
// file RifClasse.java
class RifClasse
{ private int i;
  RifClasse(int n)
  { i = n; }
  RifClasse rifMetodo()
  { RifClasse un = new RifClasse(2);
    // ...
    return un;
  }
}

// file Prog.java
public class Prog
{ public static void main(String[] args)
```

```
{ RifClasse u1, u2;  
    u1 = new RifClasse(1);  
    u2 = u1.rifMetodo();  
    // ...  
}
```

Nella classe *Prog* viene richiamato *rifMetodo()* applicato a *u1*. Il metodo crea un oggetto riferito dalla variabile locale *un* e il valore di tale variabile viene restituito con la fine dell'esecuzione del metodo stesso. La variabile locale *un* conclude il suo tempo di vita, ma l'oggetto riferito rimane in vita in quanto allocato nella memoria dinamica. L'assegnamento del risultato di *rifMetodo()* a *u2* è quindi significativo.

## 7.10. Visibilità nella classe dei membri di un oggetto della stessa classe

In una classe sono visibili tutti i membri definiti nella classe stessa, anche se privati. Pertanto sono visibili sia tutti i membri dell'oggetto implicito che tutti i membri di qualunque altro oggetto della stessa classe, di cui è presente un riferimento, come un parametro formale o una variabile locale di un metodo. Per esempio, consideriamo la seguente classe relativa ai numeri complessi:

```
// file Complesso.java  
class Complesso  
{ private double re, im;  
    Complesso()  
    {}  
    Complesso(double x, double y)  
    { re = x; im = y;  
    }  
    Complesso somma(Complesso cc)  
    { Complesso ris = new Complesso();  
      ris.re = re + cc.re;  
      ris.im = im + cc.im;  
    }  
}
```



```

        return ris;
    }
    // ...
    void stampa()
    { Console.scriviStringa(re + "+i" + im);
    }
}

// file ProvaComplesso.java
public class ProvaComplesso
{ public static void main(String[] args)
  { Complesso comp1 = new Complesso(1,2),
    comp2 = new Complesso(3,4), comp3;
    comp3 = comp1.somma(comp2);
    comp3.stampa();
  }
}

```

Nel metodo *somma()* della classe *Complesso* si accede e ai campi privati dell'oggetto implicito, e ai campi privati degli oggetti riferiti da *cc* e da *ris*. In esecuzione si ha il seguente risultato:

**4.0+i6.0**

## 7.11. Metodo *main()*

Fra i vari membri di una classe, può essere presente il metodo *main()* con la seguente intestazione:

```
public static void main ( String[] args )
```

Il comando di esecuzione (*Java nomeClasse*) deve specificare il nome di una classe che possiede il metodo *main()* con l'intestazione precedente, e l'esecuzione inizia appunto da tale metodo. La classe individuata costituisce la classe iniziale, e l'esecuzione prosegue fino a che il suo metodo *main()* non termina.

Per una determinata classe, detta appunto classe principale, l'esecuzione di un tal metodo *main()* comporta l'esecuzione di tutto il programma, con la eventuale creazione di oggetti di altre classi (o della stessa classe principale) e l'invocazione di metodi applicati agli oggetti stessi.

In genere, una qualunque classe può avere un metodo *main()* siffatto: la sua esecuzione serve unicamente a mettere a punto la classe stessa, con eliminazione di eventuali errori logici presenti nei metodi. Solo l'esecuzione del metodo *main()* della classe principale serve ad eseguire il programma.

Notare che in una classe potrebbero essere presenti metodi *main()* con una intestazione diversa dalla precedente (meccanismo di overloading), e in questo caso si comporterebbero come metodi ordinari. Nel seguito non prenderemo mai in considerazione tale ipotesi.

## 7.12. Array di variabili classe

Un oggetto array può avere come elementi, oltre che variabili di un tipo primitivo, anche variabili classe (riferimenti). Per esempio, si può scrivere:

```
class Elem
{ // ...
}

class ProvaEl
{ public static void main(String[] args)
  { Elem[] arrayElem = new Elem[10];
    // ...
  }
}
```

Gli elementi dell'oggetto array assumono il valore iniziale default *null*.

La creazione di un oggetto array avviene tramite l'operatore *new*, e non comporta l'invocazione di nessun costruttore definito nella classe a cui

appartengono gli elementi dell'oggetto array stesso.

Gli oggetti riferiti dagli elementi dell'oggetto array devono essere esplicitamente creati, utilizzando un costruttore (anche quello *default*) della classe a cui gli elementi stessi appartengono. Occorre non confondere la creazione di un oggetto array con la creazione di oggetti riferiti dai suoi elementi, anche se in entrambi i casi viene utilizzato l'operatore *new*.

Un array di variabili classe può essere inizializzato in una delle due forme previste nel Capitolo 6 (inizializzazione della variabile array (*a2*) o inizializzazione dell'oggetto array (riferito da *a3*)), ovvero può essere prima creato l'oggetto array (riferito da *a4*) e quindi creati i suoi elementi, come chiarito dal seguente esempio:

```
class El1
{ int a;
  El1(int n)
  { a = n;
  }
  // ...
}

class ProvaEl1
{ public static void main(String[] args)
  { El1[] a1 = null;
    // ...
    El1[] a2 = a1;
    El1[] a3 = { new El1(2), new El1(0), new El1(5) };
    // ...
    El1[] a4 = new El1[10];
    for (int i = 0; i < a4.length; i++)
      a4[i] = new El1(i);
    // ...
    for (int i = 0; i < a4.length; i++)
      Console.scriviStr(a4[i].a + " ");
    Console.nuovaLinea(); // ...
  }
}
```

In esecuzione abbiamo il seguente risultato:

```
0 1 2 3 4 5 6 7 8 9
```

### 7.12.1. Esempio: perimetro di un poligono

Come ulteriore esempio, si riporta un programma comprendente una classe *Poligono* con un metodo *perimetro()*, dove un vertice del poligono è un oggetto di classe *Punto* (la classe *Punto* è composta da due coordinate reali *x* e *y*). Il programma contiene il metodo privato *lung()*, che serve a calcolare la lunghezza del segmento compreso tra due vertici consecutivi.

```
// file Perimetro.java
class Poligono
{ private class Punto
  { double x; double y; }
  private int nvertici = 0;
  private Punto[] polig;
  private double lung(Punto p1, Punto p2)
  { double dx = p2.x - p1.x;
    double dy = p2.y - p1.y;
    return Math.sqrt(dx*dx + dy*dy);
  }
  public Poligono(int n, double[] xx, double[] yy)
  { int i;
    nvertici = n;
    polig = new Punto[nvertici];
    for (i = 0; i < nvertici; i++)
    { polig[i] = new Punto();
      polig[i].x = xx[i];
      polig[i].y = yy[i];
    }
  }
  public double perimetro()
  { int i, j; double perim = 0;
    for (i = 0; i < nvertici; i++)
    { j = (i + 1) % nvertici;
      perim += lung(polig[i], polig[j]);
    }
    return perim;
  }
}

public class Perimetro
{ public static void main(String[] args)
  { int i, nn; double pp; double[] ax, ay;
```

```

    Poligono pol;
    Console.scriviStringa
        ("Scrivi il numero di vertici:");
    nn = Console.leggiIntero();
    ax = new double[nn]; ay = new double[nn];
    Console.scriviStringa
        ("Scrivi le coordinate x e y dei vertici:");
    for (i = 0; i < nn; i++)
    { ax[i] = Console.leggiReale();
      ay[i] = Console.leggiReale();
    }
    pol = new Poligono(nn, ax, ay);
    pp = pol.perimetro();
    Console.scriviStringa("Il perimetro e` " + pp);
}
}

```

Un esempio di esecuzione è il seguente:

```

Scrivi il numero di vertici:
5
Scrivi le coordinate x e y dei vertici:
1 3.5
2.1 1.5
3 1.5
3 4
3.8 2.6
Il perimetro e` 10.236082225732924

```

### 7.13. Organizzazione di classi in package

Un gruppo di classi può essere racchiuso in un *package*, utilizzando, come prima istruzione di ogni file sorgente che ne fa parte la definizione:

```
package nome-package ;
```

Le classi di un package devono essere tutte contenute in cartelle di nome *nome-package*, tipicamente in un'unica cartella con quel nome. Al di fuori

di un package, le classi vengono individuate con un nome completo, costituito dal nome della classe preceduto, con la notazione a punto, dal nome del package di cui fa parte (*nome-package.nome-classe*). In tal modo si evitano conflitti sui nomi, in quanto si possono avere classi con lo stesso identificatore purché appartenenti a package diversi. All'interno di un package, invece, le classi, oltre che col loro nome completo, possono essere individuate col loro nome semplice (il nome del package può essere omissso).

In un package, oltre a file contenenti classi, possono anche esservi altri package (sottopackage), e in questo caso il nome del package annidato deve includere, con la notazione a punto, il nome del package esterno.

Una classe facente parte di un package è visibile al di fuori del package stesso (quindi, anche in un sottopackage), attraverso il suo nome completo, ma solo se possiede il modificatore di accesso *public*. Analogamente, un membro di una classe pubblica è visibile al di fuori del package solo se possiede il modificatore di accesso *public*.

Le finalità dei package possono quindi essere così riassunte:

- raggruppare le classi all'interno di unità logiche;
- evitare il conflitto tra nomi di classi;
- regolamentare l'accesso a classi e a membri con maggior finezza.

Come detto nel Capitolo 1, le classi facenti parte delle Java API sono raggruppate in package.

Le classi che esplicitamente non fanno parte di alcun package, appartengono in realtà al cosiddetto *package implicito*.

A titolo di esempio, supponiamo di avere un programma costituito da un package *lav*, e che in esso siano contenuti, oltre alle classi *Prova* e *Uno*, i sottopackage *lav1* e *lav2*, contenenti ciascuno un file di nome *Cla.java*:

```
// cartella lav, file Prova.java
package lav;
public class Prova
{ public static void main(String[] args)
  { lav.Uno p1 = new lav.Uno();
    // oppure Uno = new Uno();
    lav.lav1.Cla p2 = new lav.lav1.Cla();
    lav.lav2.Cla p3 = new lav.lav2.Cla();
    p1.xu = 1;
  }
}
```

```
        p2.xd = 2;
        p3.xt = 3;
        // ...
    }
}

// cartella lav, file Uno.java
package lav;
public class Uno
{ public int xu;
  // ...
}

// cartella lav/lav1, file Cla.java
package lav.lav1;
public class Cla
{ public int xd;
  void faid()
  { lav.lav2.Cla d1 = new lav.lav2.Cla();
    d1.fait();
    // ...
  }
}

// cartella lav/lav2, file Cla.java
package lav.lav2;
public class Cla
{ public int xt;
  public void fait()
  { lav.Uno t1 = new lav.Uno();
    t1.xu = 10;
  }
}
```

Supponiamo che il direttorio corrente sia quello che contiene la cartella *lav*. Il comando di compilazione, che deve indicare il percorso, il nome e l'estensione del file contenente la classe principale, è fatto nel modo seguente:

```
javac lav/Prova.java
```

In questo caso non occorre nessuna opzione *classpath*, in quanto il nome completo delle classi secondarie (indicato esplicitamente o espresso col solo nome semplice per classi dello stesso package) consente al compilatore di individuare il percorso (relativo rispetto al direttorio corrente) dei file che le contengono.

Il comando di esecuzione viene emesso (dal direttorio corrente) specificando il nome completo della classe principale:

```
java lav.Prova  
(oppure, in molti ambienti, anche java lav/Prova )
```

In genere, nei comandi di compilazione e di esecuzione occorre specificare, con opzioni *classpath*, i percorsi (relativi al direttorio corrente o assoluti) che consentono di individuare le cartelle dei package: notare che tali opzioni non devono comprendere i nomi di tali cartelle.

### 7.13.1. Comando *import*

Con riferimento a quanto detto nel Capitolo 1, le singole classi di un package o tutte le classi pubbliche di un package (non i sottopackage) possono essere importate, rispettivamente, con le istruzioni:

```
import nome_package.nome_classe;  
import nome_package.*;
```

In tal modo, le classi importate possono venir individuate col loro nome semplice invece che col loro nome completo (l'utilizzo del nome semplice non è ovviamente possibile nel caso di classi con lo stesso nome). Notare che in ogni classe di un package vengono automaticamente importate tutte le altre classi dello stesso package, ed è per questa ragione che all'interno di uno stesso package si possono usare nomi semplici.

A titolo di esempio, riportiamo il seguente programma:

```
// cartella pack, file Prova.java  
package pack;  
import pack.pack1.*;  
import pack.pack2.*;  
public class Prova
```



```
{ public static void main(String[] args)
{   Uno p1; Due p2;
    // ...
}
}

// cartella pack/pack1, file Uno.java
package pack.pack1;
import pack.pack2.*;
public class Uno
{   public void faiu()
    {   Due uu = new Due();
        uu.faid();
        // ...
    }
}

// cartella pack/pack2, file Due.java
package pack.pack2;
import pack.pack1.*;
public class Due
{   public void faid()
    {   Uno dd = new Uno();
        dd.faiu();
        // ...
    }
}
```

### 7.13.2. Riepilogo su package e protezione

Come visto in precedenza, i modificatori di accesso per la protezione riguardano sia le classi che i membri delle classi.

Per le classi sono possibili solo due livelli di protezione:

- package*: (nessun modificatore): il nome della classe è visibile solo nello stesso package;
- public**: il nome della classe è visibile nello stesso e in altri package.

Per i membri, sono invece possibili quattro livelli di protezione:

- private:*** il nome del membro è visibile solo nella classe in cui è definito;
- package:*** (nessun modificatore): il nome del membro è visibile, oltre che nella classe in cui è definito, anche nelle altre classi dello stesso package;
- protected:*** il nome del membro è visibile, oltre che nella classe in cui è definito, anche nelle altre classi dello stesso package e nelle *sottoclassi* dello stesso o di altro package.
- public:*** il nome del membro è visibile, oltre che nella classe in cui è definito, anche nelle altre classi dello stesso o di altro package.

Per quello che riguarda il modificatore *protected* e le sottoclassi, si veda il Capitolo 10.

### 7.13.3. Package e classe *Console*

Per programmi organizzati in package, è stato predisposto un package *IngressoUscita* contenente la classe *Console* (Capitolo 12), prelevabile dall'indirizzo Internet:

*<http://www2.ing.unipi.it/LibroJava>*

Per l'utilizzo di tale package occorre 1) ricopiarlo nella cartella in cui si sviluppa il programma e 2) in ogni file del programma usare il comando `import IngressoUscita.*`.

## 8. Liste

### 8.1. Forma di una lista

Una tipica utilizzazione di classi con soli campi dati si ha nelle cosiddette *liste*, formate da elementi dello stesso tipo collegati a catena, la cui lunghezza varia dinamicamente.

Ogni elemento è un oggetto classe di un tipo *Elemento*, costituito solo da campi dati: uno o più membri contenenti informazioni, e un membro riferimento contenente l'indirizzo dell'elemento successivo. Il primo elemento è indirizzato da una variabile riferimento (riferimento della lista), e il membro riferimento dell'ultimo elemento contiene il valore *null*.

Il tipo *Lista* è anch'esso una classe, contenente un campo dati di tipo *Elemento* (che rappresenta il riferimento della lista) e alcuni metodi per manipolare la lista stessa.

In figura 8.1 è rappresentata una lista di tre elementi, ognuno dei quali è un oggetto di tipo *Elemento* con un membro *inf* contenente informazioni e un membro *pun* che è un riferimento di tipo *Elemento* (ricordare che una classe può avere un membro costituito da un riferimento dello stesso tipo classe). I singoli elementi della lista vengono inseriti e estratti a tempo di esecuzione, ottenendo una catena che cresce o decresce dinamicamente a seconda delle specifiche necessità. La lista è costituita da un oggetto di tipo *Lista* contenente un riferimento del primo elemento (*rpl*), riferimento che assume valore *null* quando la lista è vuota (non contiene elementi).

Per semplicità, nel seguito considereremo elementi la cui informazione sia costituita da un intero.

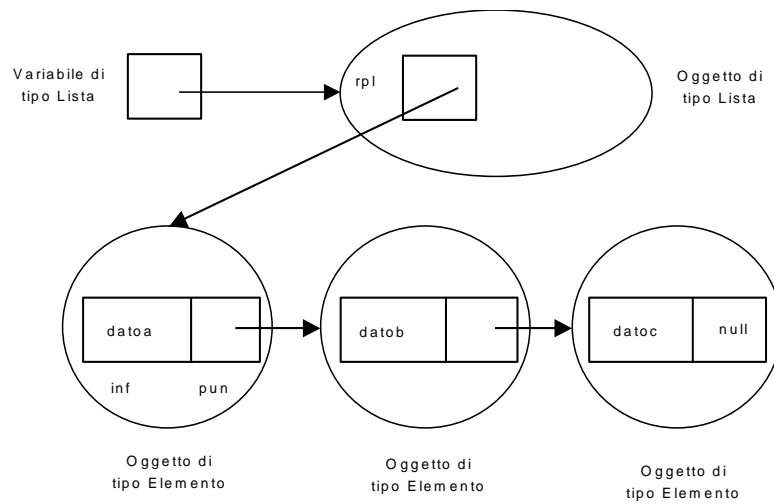


Figura 8.1. Organizzazione di una lista

Lo schema di programma a cui ci si riferisce è il seguente:

```
// file ProgLista.java
class Lista
{ private class Elemento
  { int inf;
    Elemento pun;
  }
  private Elemento rpl;
  boolean empty()
  { if (rpl == null) return true;
    return false;
  }
  void insRegolaIns1(int s)
  { // ...
  }
  // ...
  int estRegolaEst1()
  { // ...
  }
  // ...
}
```

```
void stampa()
{ if (rpl == null)
    Console.scriviStringa("Lista vuota");
  else
  { Elemento rr = rpl;
    Console.scriviStringa("Elementi:");
    while (rr != null)
    { Console.scriviInt(rr.inf); rr = rr.pun; }
    Console.nuovaLinea();
  }
}

public class ProgLista
{ public static void main(String[] args)
  { Lista ll;
    // ...
  }
}
```

Gli inserimenti e le estrazioni di dati nella/dalla lista avvengono in accordo a determinate regole, e vengono effettuati dai vari metodi *insRegolaIns()* ed *estRegolaEst()*. Il riferimento della lista *rpl* viene ad assumere un nuovo valore se l'inserimento o l'estrazione interessa il primo elemento della lista, mentre il suo valore non cambia se la modifica della lista interessa gli altri elementi.

Il metodo *stampa()* effettua la stampa dei contenuti dei membri informazione dei vari elementi della lista, utilizzando un riferimento *rr* che, partendo dal primo elemento, li scorre in sequenza.

## 8.2. Operazioni sulle liste

Di seguito vengono riportati alcuni metodi che realizzano le più comuni operazioni sulle liste (le operazioni di estrazione non possono avvenire se la lista è vuota). Le figure riportate sono semplificate, ed evidenziano le modifiche che debbono subire i riferimenti in relazione alle varie operazioni.

### 8.2.1. Inserimento in testa a una lista

Il seguente metodo inserisce un nuovo elemento in testa alla lista puntata da *rpl*, con il valore *s* nel membro informazione dell'elemento inserito, e aggiorna il valore di *rpl* (Fig. 8.2):

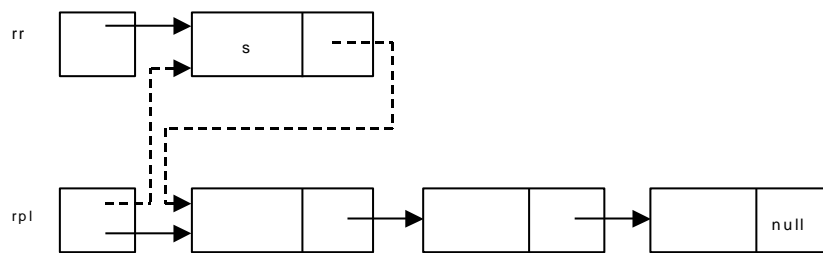


Figura 8.2. Inserimento di un elemento in testa a una lista

```
void insTesta(int s)
{ Elemento rr = new Elemento();
  rr.inf = s; rr.pun = rpl;
  rpl = rr;
}
```

### 8.2.2. Estrazione dalla testa di una lista

Il seguente metodo estrae l'elemento di testa dalla lista puntata da *rpl*, se questa non è vuota. Nel caso in cui l'estrazione è possibile, il metodo assegna ad *s* il contenuto del membro informazione di tale elemento e aggiorna il valore di *rpl* (Fig. 8.3):

```
int estTesta()
{ int s = 0;
  if (rpl!=null)
  { s = rpl.inf;
    rpl = rpl.pun;
  }
  return s;
}
```

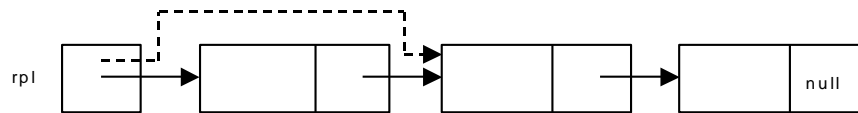


Figura 8.3. Estrazione di un elemento dalla testa di una lista

### 8.2.3. Inserimento in fondo a una lista

Il seguente metodo inserisce un nuovo elemento in fondo alla lista puntata da *rpl*, con il valore *s* nel membro informazione dell'elemento inserito, e aggiorna il valore di *rpl* se la lista è inizialmente vuota (Fig. 8.4):

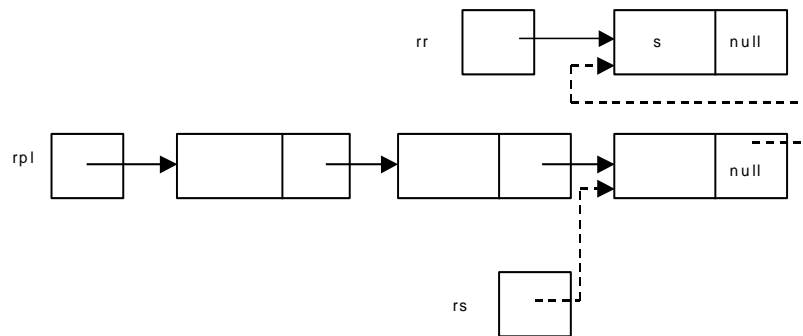


Figura 8.4. Inserimento di un elemento in fondo a una lista

```
void insFondo(int s)
{ Elemento rr = new Elemento();
  rr.inf = s; rr.pun = null;
  if (rpl == null) rpl = rr;
  else
  { Elemento rs = rpl;
    while (rs.pun!=null) rs = rs.pun;
    rs.pun = rr;
  }
}
```

Nel caso di lista vuota, *rpl* deve riferire il nuovo elemento. Nel caso di lista non vuota, prima di effettuare l'inserimento è necessario scorrere la lista stessa fino a riferire l'ultimo elemento (nel membro *pun* di tale elemento deve essere ricopiato il valore del riferimento all'elemento da inserire).

#### 8.2.4. Estrazione dal fondo di una lista

Il seguente metodo estrae l'ultimo elemento dalla lista puntata da *rpl* (Fig. 8.5). Se la lista non è vuota e contiene un solo elemento, *s* assume il valore del membro informazione di tale elemento, e *rpl* il valore *null*. Se la lista contiene più di un elemento, occorre preliminarmente scorrere la lista: a tale scopo vengono utilizzati due riferimenti, *rs* ed *rr*, che avanzano distanziati di un elemento: lo scorrimento si arresta quando *rr* riferisce l'ultimo elemento ed *rs* il penultimo, del quale occorre modificare il campo *pun*.

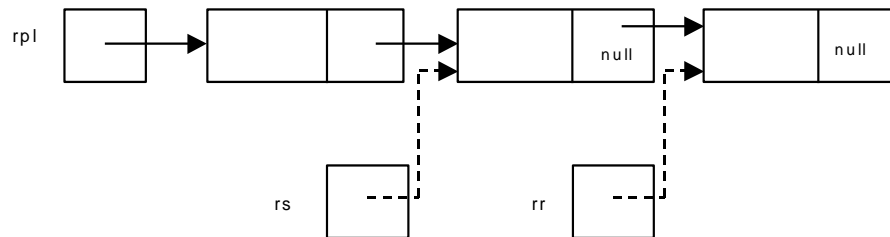


Figura 8.5. Estrazione di un elemento dal fondo di una lista

```
int estFondo()
{ int s = 0;
  if (rpl!=null)
  { if (rpl.pun == null)
    { s = rpl.inf; rpl = null; }
    else
    { Elemento rr = rpl, rs = null;
      while (rr.pun!=null) { rs = rr; rr = rr.pun; }
      s = rr.inf; rs.pun = null;
    }
  }
```



```
    }  
    return s;  
}
```

### 8.2.5. Inserimento in una lista ordinata

Il seguente metodo inserisce un nuovo elemento nella lista ordinata (in senso crescente) puntata da *rpl*, in modo da mantenere l'ordinamento. Il membro informazione del nuovo elemento viene posto uguale ad *s*, e l'inserimento viene effettuato prima dell'eventuale elemento nel cui membro informazione vi sia una quantità maggiore o uguale ad *s*:

```
void insOrd(int s)  
{ Elemento rs = rpl, rt = null;  
  while (rs != null && rs.inf < s)  
  { rt = rs; rs = rs.pun; }  
  Elemento rr = new Elemento();  
  rr.inf = s; rr.pun = rs;  
  // controlla se l'inserimento avviene in testa  
  if (rs == rpl) rpl = rr; else rt.pun = rr;  
}
```

### 8.2.6. Estrazione di un elemento dato

Il seguente metodo verifica se nella lista puntata da *rpl* esiste un elemento nel cui membro informazione è contenuto il valore *s*, e in caso affermativo estrae l'elemento dalla lista:

```
boolean estElem(int s)  
{ Elemento rs = rpl, rt = null;  
  while (rs != null && rs.inf != s)  
  { rt = rs; rs = rs.pun; }  
  // controlla se la lista è stata scorsa per intero  
  if (rs == null) return false;  
  // controlla se viene estratto il primo elemento  
  if (rs == rpl) rpl = rs.pun;  
  else rt.pun = rs.pun;  
  return true;  
}
```

### 8.2.7. Programmi riepilogativi

Il seguente programma gestisce una lista con inserimenti in testa ed estrazioni dalla testa:

```
// file ProgLista1.java
class Lista
{ private class Elemento
  { int inf;
    Elemento pun;
  }
  private Elemento rpl;
  boolean empty()
  { if (rpl == null) return true;
    return false;
  }
  void insTesta(int s)
  { Elemento rr = new Elemento();
    rr.inf = s; rr.pun = rpl;
    rpl = rr;
  }
  int estTesta()
  { int s = 0;
    if (rpl!=null)
    { s = rpl.inf;
      rpl = rpl.pun;
    }
    return s;
  }
  void stampa()
  { if (rpl == null)
    Console.scriviStringa("Lista vuota");
    else
    { Elemento rr = rpl;
      Console.scriviStringa("Elementi:");
      while (rr != null)
      { Console.scriviInt(rr.inf); rr = rr.pun; }
      Console.nuovaLinea();
    }
  }
}
```

```
public class ProgLista1
{ public static void main(String[] args)
  { int n, m, i; Lista ll = new Lista();
    Console.scriviStringa
      ("Scrivi quanti numeri vuoi inserire:");
    m = Console.leggiIntero();
    Console.scriviStringa
      ("Scrivi " + m + " numeri:");
    for (i = 0; i<m; i++)
    { n = Console.leggiIntero();
      ll.insTesta(n);
      Console.scriviStringa("Inserito " + n);
    }
    Console.scriviStringa
      ("Scrivi quanti numeri vuoi estrarre");
    m = Console.leggiIntero();
    for (i = 0; i<m; i++)
      if (!ll.empty())
      { n = ll.estTesta();
        Console.scriviStringa("Estratto " + n);
      }
      else Console.scriviStringa
        ("Nessuna estrazione");
    Console.scriviStringa("Contenuto della lista:");
    ll.stampa();
  }
}
```

Un esempio di esecuzione è il seguente:

```
Scrivi quanti numeri vuoi inserire:
5
Scrivi 5 numeri:
1 2 3 4 5
Inserito 1
Inserito 2
Inserito 3
Inserito 4
Inserito 5
Scrivi quanti numeri vuoi estrarre:
2
Estratto 5
Estratto 4
```

Contenuto della lista:

Elementi:

3 2 1

Il seguente programma gestisce una lista con inserimenti in fondo ed estrazioni dal fondo:

```
// file ProgLista2.java
class Lista
{ private class Elemento
  { int inf;
    Elemento pun;
  }
  private Elemento rpl;
  boolean empty()
  { if (rpl == null) return true;
    return false;
  }
  void insFondo(int s)
  { Elemento rr = new Elemento();
    rr.inf = s; rr.pun = null;
    if (rpl == null) rpl = rr;
    else
    { Elemento rs = rpl;
      while (rs.pun!=null) rs = rs.pun;
      rs.pun = rr;
    }
  }
  int estFondo()
  { int s = 0;
    if (rpl!=null)
    { if (rpl.pun == null)
      { s = rpl.inf; rpl = null; }
      else
      { Elemento rr = rpl, rs = null;
        while (rr.pun!=null) { rs = rr; rr = rr.pun; }
        s = rr.inf; rs.pun = null;
      }
    }
    return s;
  }
  void stampa()
  { if (rpl == null)
```

```

        Console.scriviStringa("Lista vuota");
    else
    { Elemento rr = rpl;
      Console.scriviStringa("Elementi:");
      while (rr != null)
      { Console.scriviInt(rr.inf); rr = rr.pun; }
      Console.nuovaLinea();
    }
}

public class ProgLista2
{ public static void main(String[] args)
  { int n, m, i; Lista ll = new Lista();
    Console.scriviStringa
      ("Scrivi quanti numeri vuoi inserire:");
    m = Console.leggiIntero();
    Console.scriviStringa
      ("Scrivi " + m + " numeri:");
    for (i = 0; i<m; i++)
    { n = Console.leggiIntero();
      ll.insFondo(n);
      Console.scriviStringa("Inserito " + n);
    }
    Console.scriviStringa
      ("Scrivi quanti numeri vuoi estrarre");
    m = Console.leggiIntero();
    for (i = 0; i<m; i++)
      if (!ll.empty())
      { n = ll.estFondo();
        Console.scriviStringa("Estratto " + n);
      }
      else Console.scriviStringa
        ("Nessuna estrazione");
    Console.scriviStringa("Contenuto della lista:");
    ll.stampa();
  }
}

```

Un esempio di esecuzione è il seguente:

```

Scrivi quanti numeri vuoi inserire:
3

```

```

Scrivi 3 numeri:
1 2 3
Inserito 1
Inserito 2
Inserito 3
Scrivi quanti numeri vuoi estrarre:
5
Estratto 3
Estratto 2
Estratto 1
Nessuna estrazione
Nessuna estrazione
Contenuto della lista:
Lista vuota

```

Il seguente programma gestisce una lista ordinata con inserimenti ordinati ed estrazioni di elementi con informazione specificata:

```

// file ProgLista3.java
class Lista
{ private class Elemento
  { int inf;
    Elemento pun;
  }
  private Elemento rpl;
  boolean empty()
  { if (rpl == null) return true;
    return false;
  }
  void insOrd(int s)
  { Elemento rs = rpl, rt = null;
    while (rs != null && rs.inf < s)
    { rt = rs; rs = rs.pun; }
    Elemento rr = new Elemento();
    rr.inf = s; rr.pun = rs;
    // controlla se l'inserimento avviene in testa
    if (rs == rpl) rpl = rr; else rt.pun = rr;
  }
  boolean estElem(int s)
  { Elemento rs = rpl, rt = null;
    while (rs != null && rs.inf != s)
    { rt = rs; rs = rs.pun; }
    // controlla se la lista è stata scorsa per intero

```

```
        if (rs == null) return false;
        // controlla se viene estratto il primo elemento
        if (rs == rpl) rpl = rs.pun;
            else rt.pun = rs.pun;
        return true;
    }
    void stampa()
    { if (rpl == null)
        Console.scriviStringa("Lista vuota");
      else
      { Elemento rr = rpl;
        Console.scriviStringa("Elementi:");
        while (rr != null)
        { Console.scriviInt(rr.inf); rr = rr.pun; }
        Console.nuovaLinea();
      }
    }
}

public class ProgLista3
{ public static void main(String[] args)
  { int n, m, i; Lista ll = new Lista();
    Console.scriviStringa
      ("Scrivi quanti numeri vuoi inserire");
    m = Console.leggiIntero();
    Console.scriviStringa
      ("Scrivi " + m + " numeri:");
    for (i = 0; i < m; i++)
    { n = Console.leggiIntero();
      ll.insOrd(n);
    }
    Console.scriviStringa("Contenuto della lista:");
    ll.stampa();
    Console.scriviStringa("Estrazione di 5 numeri:");
    for (i=0; i<5; i++)
    { Console.scriviStringa("Numero da estrarre:");
      m = Console.leggiIntero();
      if (ll.estElem(m))
        Console.scriviStringa("Estratto " + m);
      else Console.scriviStringa("Non presente");
    }
    Console.scriviStringa("Contenuto della lista:");
```

```
        ll.stampa();  
    }  
}
```

Un esempio di esecuzione è il seguente:

**Scrivi quanti numeri vuoi inserire:**

5

**Scrivi 5 numeri**

7 3 5 5 1

**Contenuto della lista:**

**Elementi:**

1 3 5 5 7

**Estrazione di 5 numeri**

**Numero da estrarre:**

3

**Estratto 3**

**Numero da estrarre:**

2

**Non presente**

**Numero da estrarre:**

8

**Non presente**

**Numero da estrarre:**

7

**Estratto 7**

**Numero da estrarre:**

5

**Estratto 5**

**Contenuto della lista:**

**Elementi:**

1 5

### 8.3. Liste con riferimento dell'ultimo elemento

È possibile avere liste con più di un riferimento di elementi, come un



riferimento al primo elemento, sia *rpl* e uno all'ultimo, sia *rul*. Una lista così fatta presenta vantaggi per l'operazione di inserimento in fondo alla lista, poiché non richiede preliminari scorrimenti.

La manipolazione di una lista con un riferimento dell'ultimo elemento può essere effettuata con metodi analoghi a quelli visti in precedenza. Per esempio, il primo metodo riportato di seguito estrae un elemento dalla testa della lista, e il secondo inserisce un nuovo elemento in fondo a alla lista:

```
void insFondo1(int s)
{ Elemento rr = new Elemento();
  rr.inf = s; rr.pun = null;
  if (rpl == null) { rpl = rr; rul = rr; }
  else
  { rul.pun = rr; rul = rr; }
}

int estTesta1()
{ int s = 0;
  if (rpl!=null)
  { s = rpl.inf;
    rpl = rpl.pun;
    if (rpl == null) rul = null;
  }
  return s;
}
```

Il seguente programma utilizza i metodi precedenti, costruendo anzitutto una lista, ed effettuando prima un certo numero di inserzioni (in fondo) e successivamente un certo numero di estrazioni (dalla testa):

```
// file ProgListaN.java
class ListaN
{ private class Elemento
  { int inf;
    Elemento pun;
  }
  private Elemento rpl, rul;
  boolean empty()
  { if (rpl == null) return true;
    return false;
  }
}
```

```

    }
    void insFondol(int s)
    { Elemento rr = new Elemento();
      rr.inf = s; rr.pun = null;
      if (rpl == null) { rpl = rr; rul = rr; }
      else
      { rul.pun = rr; rul = rr; }
    }
    int estTestal()
    { int s = 0;
      if (rpl!=null)
      { s = rpl.inf;
        rpl = rpl.pun;
        if (rpl == null) rul = null;
      }
      return s;
    }
    void stampa()
    { if (rpl == null)
      Console.scriviStringa("Lista vuota");
      else
      { Elemento rr = rpl;
        Console.scriviStringa("Elementi:");
        while (rr != null)
        { Console.scriviInt(rr.inf); rr = rr.pun; }
        Console.nuovaLinea();
      }
    }
  }
}

public class ProgListaN
{ public static void main(String[] args)
  { int n, m, i; ListaN ll = new ListaN();
    Console.scriviStringa
      ("Scrivi quanti numeri vuoi inserire:");
    m = Console.leggiIntero();
    Console.scriviStringa
      ("Scrivi " + m + " numeri:");
    for (i = 0; i<m; i++)
    { n = Console.leggiIntero();
      ll.insFondol(n);
      Console.scriviStringa("Inserito " + n);
    }
  }
}

```

```
        Console.scriviStringa
            ("Scrivi quanti numeri vuoi estrarre");
m = Console.leggiIntero();
for (i = 0; i<m; i++)
    if (!ll.empty())
    { n = ll.estTesta1();
      Console.scriviStringa("Estratto " + n);
    }
    else Console.scriviStringa
        ("Nessuna estrazione");
Console.scriviStringa("Contenuto della lista:");
ll.stampa();
    }
}
```

Un esempio di esecuzione è il seguente:

```
Scrivi quanti numeri vuoi inserire:
4
Scrivi 4 numeri:
1 2 3 4
Inserito 1
Inserito 2
Inserito 3
Inserito 4
Scrivi quanti numeri vuoi estrarre:
3
Estratto 1
Estratto 2
Estratto 3
Contenuto della lista:
Elementi:
4
```

## 8.4. Liste complesse

E' possibile gestire liste i cui elementi contengano più di un membro

riferimento. Si possono, per esempio, avere liste doppie, in cui ogni elemento riferisce sia il precedente che il successivo, con la conseguente possibilità di scorrimento nei due sensi (vedi Fig. 8.6). Il tipo degli elementi può quindi essere definito nel seguente modo:

```
class ElementoD
{ int inf;
  ElementoD prec;
  ElementoD succ;
}
```

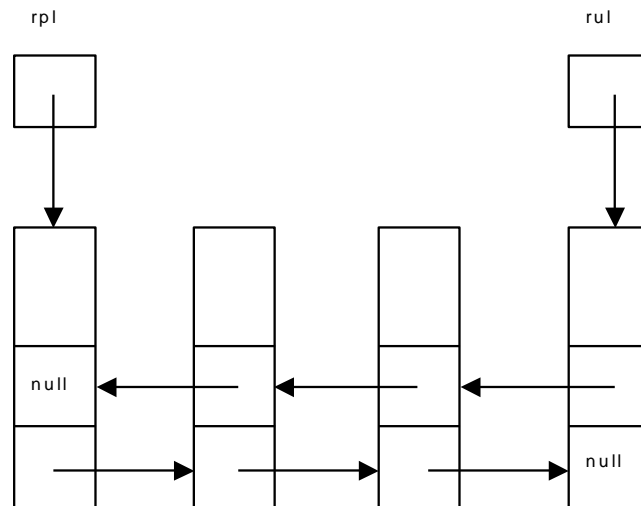


Figura 8.6: Una lista doppia

Con questo tipo di elementi, si possono avere liste con due riferimenti, al primo e all'ultimo elemento, come mostrato in Fig. 8.6.

Si possono anche avere liste ramificate, i cui elementi riferiscono intere liste, con possibilità di rappresentare in modo naturale i grafi. Notare che è comunque opportuno che tutti gli elementi di una lista siano classi dello stesso tipo, per rendere agevoli le operazioni di scorrimento.

## 8.5. Realizzazione di pile con liste

Una pila può anche essere realizzata utilizzando una lista. La pila viene a coincidere con un puntatore alla lista, in cui le operazioni di inserimento e di estrazione vengono effettuate dalla testa. Volendo mantenere inalterate le intestazioni dei metodi della classe *PilaArray* vista nel Capitolo 7, possiamo scrivere la seguente classe *PilaList*:

```
// file PilaList.java
class PilaList
{ private class Elemento
  { int inf; Elemento pun;
  }
  private int quanti, maxElem;
  private Elemento rpl;
  PilaList(int n)
  { maxElem = n; }
  boolean empty()
  { if (quanti == 0) return true;
    return false;
  }
  boolean full()
  { if (quanti == maxElem) return true;
    return false;
  }
  void push(int s)
  { if (quanti < maxElem)
    { quanti++; Elemento rr = new Elemento();
      rr.inf = s; rr.pun = rpl; rpl = rr;
    }
  }
  int pop()
  { int s = 0;
    if (quanti > 0)
    { quanti--;
      s = rpl.inf; rpl = rpl.pun;
    }
    return s;
  }
  void stampa()
```

```

    { if (empty()) Console.scriviStringa("Pila vuota");
      else
      { Console.scriviStringa
        ("Numero di elementi: " + quanti);
        Elemento rr = rpl;
        Console.scriviStringa("Valori:");
        while (rr != null)
        { Console.scriviInt(rr.inf); rr = rr.pun; }
        Console.nuovaLinea();
      }
    }
  }
}

```

La classe *ProvaPilaList* può essere fatta in modo analogo alla classe *ProvaPilaArray* vista nel Capitolo 7.

## 8.6. Realizzazione di code con liste

Una coda può anche essere realizzata utilizzando una lista. Le operazioni di inserimento e di estrazione devono essere effettuate da parti opposte: per evitare scorrimenti, conviene 1) considerare una lista con un puntatore ausiliario all'ultimo elemento, e 2) effettuare inserimenti in fondo ed estrazioni in testa. Volendo mantenere inalterate le intestazioni dei metodi della classe *CodaaArray* vista nel Capitolo 7, possiamo scrivere la seguente classe *CodaList*:

```

// file CodaList.java
class CodaList
{ private class Elemento
  { int inf; Elemento pun;
  }
  private int quanti, maxElem;
  private Elemento rpl, rul;
  CodaList(int n)
  { maxElem = n; }
  boolean empty()
  { if (quanti == 0) return true;

```

```
        return false;
    }
    boolean full()
    { if(quantiti == maxElem) return true;
      return false;
    }
    void enqueue(int s)
    { if (quantiti < maxElem)
      { quantiti++;
        Elemento rr = new Elemento();
        rr.inf = s; rr.pun = null;
        if (rpl == null) { rpl = rr; rul = rr; }
        else
        { rul.pun = rr; rul = rr; }
      }
    }
    int dequeue()
    { int s = 0;
      if (quantiti > 0)
      { quantiti--;
        s = rpl.inf; rpl = rpl.pun;
        if (rpl == null) rul = null;
      }
      return s;
    }
    void stampa()
    { if (empty()) Console.scriviStringa("Pila vuota");
      else
      { Console.scriviStringa
        ("Numero di elementi: " + quantiti);
        Elemento rr = rpl;
        Console.scriviStringa("Valori:");
        while (rr != null)
        { Console.scriviInt(rr.inf); rr = rr.pun; }
        Console.nuovaLinea();
      }
    }
}
```

## 9. Altre proprietà delle classi

### 9.1. Riferimento *this*

All'interno di una classe la parola chiave *this* rappresenta il riferimento dell'oggetto implicito (istanza generica della classe). Pertanto, se *v* e *t* sono variabili istanza, e *m()* e *n()* metodi, nella classe si può scrivere:

<b>this.t = this.v;</b>	assegna alla variabile <i>t</i> dell'oggetto implicito il valore della variabile <i>v</i> dello stesso oggetto
<b>this.m();</b>	invoca il metodo <i>m()</i> applicato all'oggetto implicito
<b>this.n(this);</b>	invoca il metodo <i>n()</i> applicato all'oggetto implicito con argomento attuale costituito dal riferimento dell'oggetto implicito
<b>return this;</b>	restituisce (un metodo) il riferimento dell'oggetto implicito

In molti casi *this* può essere omissso (il significato rimane chiaro): per esempio, nei primi due casi precedenti si può semplicemente scrivere:

<b>t = v;</b>	assegna alla variabile <i>t</i> dell'oggetto implicito il valore della variabile <i>v</i> dello stesso oggetto
<b>m();</b>	invoca il metodo <i>m()</i> applicato all'oggetto implicito

Tuttavia, in alcune circostanze l'uso di *this* si rende necessario, come nelle seguenti:



- se l'oggetto implicito è argomento attuale di un metodo o valore di ritorno di un metodo;
- se in un metodo si deve utilizzare una variabile istanza che ha lo stesso nome di un parametro formale o di una variabile locale.

Il riferimento *this* può essere considerato a tutti gli effetti un membro della classe: a prescindere dalla indicazione esplicita del programmatore, le variabili istanza, i costruttori e i metodi dell'oggetto implicito vengono sempre individuati, all'interno della classe, tramite tale riferimento.

A titolo di esempio, si consideri il seguente caso:

```
class MiaClasse
{ int i;
  MiaClasse(int a)
  { this.i = a;      // si puo` anche scrivere i = a
    //
  }
  void fai(MiaClasse e)
  { /* ... */ }
  MiaClasse elabora()
  { int i;
    // ...
    i = 10;          // variabile locale
    this.i = 20;      // variabile istanza
    this.fai(this);  // si puo` anche scrivere fai(this)
    // ...
    return this;
  }
}
```

Quando viene creato un oggetto classe (tramite l'operatore *new*), il riferimento *this* viene istanziato al pari degli altri membri definiti esplicitamente nella classe, e la copia di *this* viene inizializzata per prima, con l'indirizzo restituito dall'operatore *new*. Vengono quindi valutati eventuali inizializzatori e viene invocato un costruttore: successivamente possono venir applicati all'oggetto eventuali metodi in accordo a quanto stabilito dal programma.

Consideriamo, per esempio, la classe *UtilizzaMiaClasse* che fa uso di variabili e oggetti della precedente classe *MiaClasse*:

```

class UtilizzaMiaClasse
{ public static void main(String[] args)
  { MiaClasse e1 = new MiaClasse(3);
    MiaClasse e2 = e1.elabora();
    // ...
  }
}

```

Quando viene creato il primo oggetto, la copia di *this* assume il valore restituito dall'operatore *new*, valore che viene memorizzato anche in *e1*. Il costruttore opera quindi sulla copia della variabile *i* riferita da *this*, come pure il metodo *elabora()* applicato all'oggetto riferito da *e1* utilizza variabili e metodi dell'oggetto riferito da *this* (e quindi da *e1*) (Fig. 9.1).

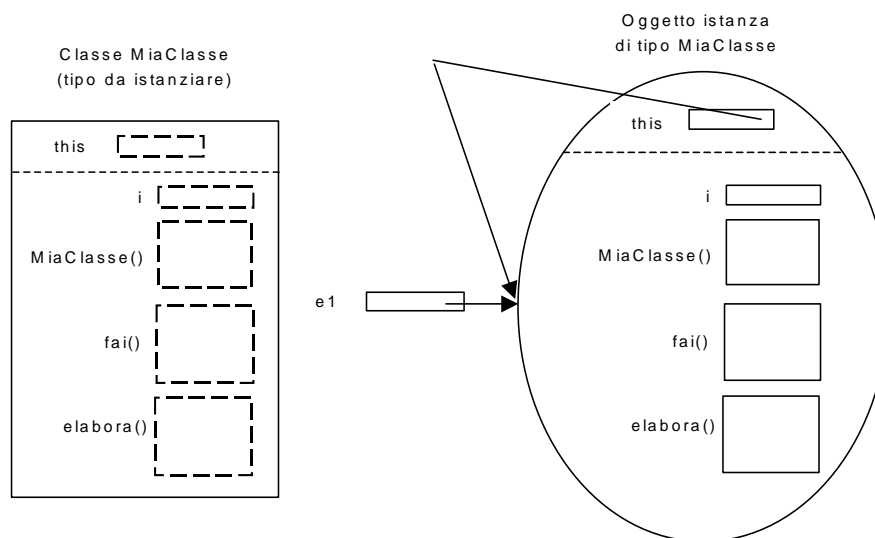


Figura 9.1. Valore di *this* nell'oggetto riferito da *e1*.

## 9.2. Variabili statiche

Una variabile definita nella classe (non una variabile locale) può avere

il modificatore *static*: in questo caso, non esiste una copia della variabile diversa per ogni oggetto classe, ma una copia unica nella classe. Ogni oggetto classe, accanto a una propria copia di variabili istanza (non statiche), accede in condivisione anche alle variabili statiche definite nella classe. Quindi le variabili statiche appartengono alla classe, non richiedono la presenza di oggetti della classe e sono condivise da tutti gli eventuali oggetti della classe stessa.

Le variabili statiche vengono inizializzate quando viene incontrata la definizione di classe (non quando viene creato un oggetto classe): l'espressione che ne determina il valore iniziale può utilizzare solo costanti e membri statici.

All'interno della classe, una variabile statica viene individuata con un identificatore completo comprendente l'identificatore della classe: il programmatore, tuttavia, individua comunemente una variabile statica indicando il suo identificatore semplice (tranne i casi di omonimia con un parametro formale o una variabile locale di un costruttore o di un metodo). Meno naturalmente, il programmatore può individuare una tale variabile anche tramite il riferimento *this* dell'oggetto implicito, o tramite qualunque altro riferimento di un oggetto della stessa classe.

Fuori della classe, una variabile statica (se non è privata) viene individuata con un identificatore completo comprendente l'identificatore della classe: il programmatore può anche individuare una tale variabile tramite un qualunque riferimento di un oggetto della stessa classe.

Quanto detto in precedenza è illustrato dal seguente esempio (vedi anche la Figura 9.2):

```
class Uno
{ static int i;
  int j;
  Uno()
  { Uno.i++;          // anche i++ oppure this.i++
    this.j++;         // anche j++
  };
  void fai(Uno u)
  { int i;
    i = 5;             // i locale
    Uno.i = 6;         // i di classe
    u.i = 7;           // anche Uno.i oppure this.i
    u.j = 8;
```

```

    // ...
}
}

class Due
{
    Uno alfa = new Uno(), beta = new Uno();
    // Uno.i e` la stessa per i due oggetti e vale 2
    // essa si individua anche con alfa.i e beta.i
    // alfa.j e beta.j sono diverse per i due oggetti
    // e valgono entrambe 1
}

```

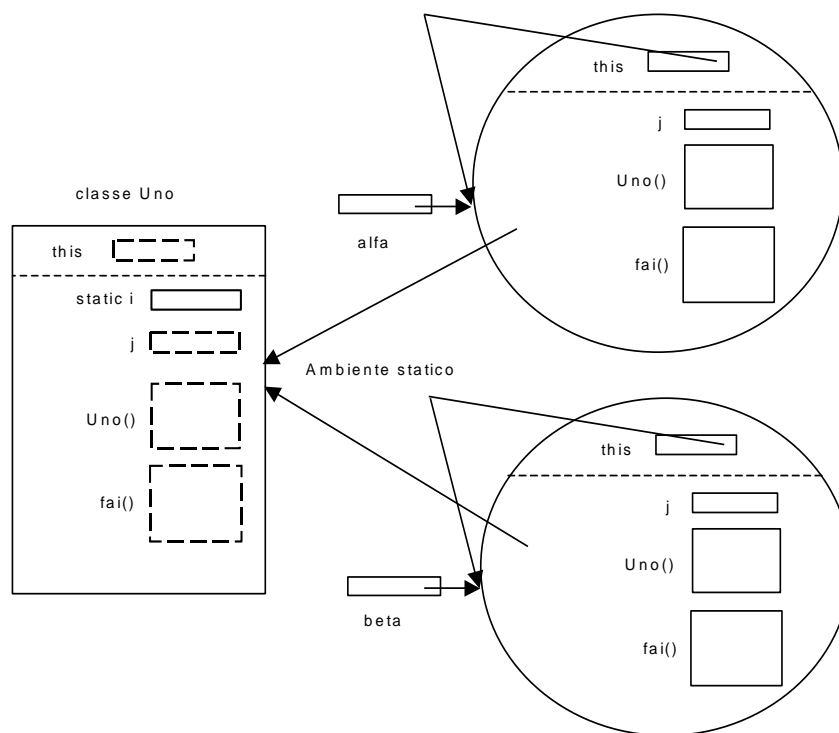


Figura 9.2. Variabile *i* statica (tratto unito nella classe)

Le variabili statiche contengono dati globali della classe, e possono venir

utilizzate per scambiare informazioni tra metodi della stessa classe o fra oggetti della stessa classe.

### 9.3. Metodi statici

Anche un metodo può avere il modificatore *static*: in questo caso non può far uso del riferimento *this*.

Un metodo statico, quindi, può utilizzare i) le variabili statiche della classe, e ii) le variabili statiche e non statiche di un oggetto di cui possiede il riferimento (questo non può essere una variabile istanza, ma una variabile statica, un parametro formale o una variabile locale). Come detto nel paragrafo 9.2, in caso di omonimia fra una variabile statica e un parametro formale o una variabile locale del metodo, la variabile statica si individua tramite un identificatore completo comprendente il nome della classe, e non facendo uso del riferimento *this* che il metodo non può utilizzare.

Analogamente, un metodo statico può utilizzare i) i metodi statici della classe, e ii) i metodi statici e non statici applicati a un oggetto di cui possiede il riferimento (questo non può essere una variabile istanza, ma una variabile statica, un parametro formale o una variabile locale).

Fuori della classe, un metodo statico (se non è privato) può essere richiamato tramite un identificatore completo (identificatore della classe e identificatore del metodo), e può anche essere applicato a un oggetto della stessa classe, analogamente a un metodo non statico.

Quindi i metodi statici appartengono alla classe, nel senso che non richiedono la presenza di oggetti della classe e eseguono le stesse elaborazioni qualunque sia l'oggetto della classe stessa a cui sono applicati.

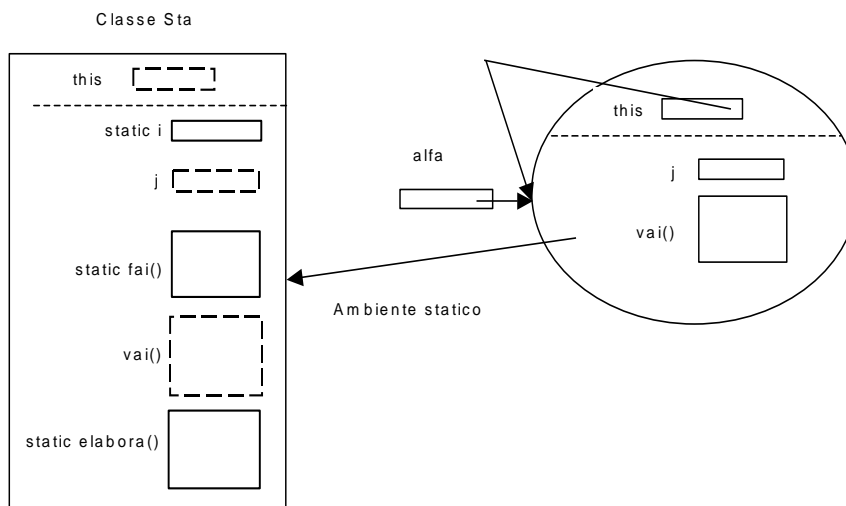
Quanto detto in precedenza è illustrato dal seguente esempio (vedi anche la Figura 9.3):

```
class Sta
{ static int i;
  int j;
  static void fai()
```

```

    { // ...
    };
    void vai()
    { // ...
    };
    static void elabora(Uno u)
    { i++;
      // j++;           errore: equivalente a this.j++;
      u.j++;
      fai();
      // vai();         errore: equivalente a this.vai();
      u.vai();
    }
}

```

Figura 9.3. Variabile *i* statica (tratto unito nella classe)

```

public class ProvaSta
{ Sta beta = new Sta();
  public static void main(String[] args)
  { Sta alfa = new Sta();
    Sta.fai();    // oppure alfa.fai();
  }
}

```

```

        alfa.vai();
        // ...
        // beta.fai();    errore
        // beta.vai();    errore
        // il metodo statico main() non puo`
        // utilizzare la variabile istanza beta
    }
}

```

In una classe *Classe*, il metodo *main()*, se presente, è statico. Esso viene eseguito, per mezzo del comando *java Classe*, senza che sia richiesta la preliminare creazione di alcun oggetto della classe stessa.

Notare che le classi, come la classe *Math*, che hanno solo variabili statiche e metodi statici, vengono in realtà concepite per mettere a disposizione degli utenti dati e funzioni di utilità.

## 9.4. Classi membro

Come visto nel Capitolo 7, una classe (classe esterna) può avere come membro un'altra classe (classe annidata), come nel seguente esempio:

```

class ClasseEsterna
{ // ...
    class ClasseAnnidata
    {
        // ...
    }
    // ...
}

```

Una classe annidata può avere tutti i modificatori di accesso previsti per i membri. Se una classe annidata non è definita statica, allora prende anche il nome di classe *interna*. La relazione di classe annidata è quindi una proprietà testuale, mentre quella di classe interna (come vedremo nel sottoparagrafo seguente) è una proprietà funzionale.

### 9.4.1. Classi interne

Una classe interna ha accesso, oltre che a tutti i suoi membri, anche a tutti i membri della classe esterna (anche ai membri privati), mentre una classe esterna può accedere ai membri non privati di una classe interna solo tramite un oggetto della classe interna stessa. Il seguente esempio chiarisce quanto appena detto:

```
class ClasseEsterna
{ private int i;
  // ...
  class ClasseInterna
  { int j = 5;
    // ...
    void fai()
    { i++; j++;
      // ...
    }
  }
  void elabora()
  { ClasseInterna ci = new ClasseInterna();
    ci.j = 10;
    // ...
  }
}
```

Una classe interna possiede, oltre al proprio riferimento *this*, anche il riferimento *this* della classe esterna (riferimento avente nome completo *ClasseEsterna.this*). In caso di omonimia fra nomi di membri, nella classe interna, un nome definito nella classe interna stessa copre quello definito nella classe esterna: tuttavia, anche quest'ultimo è visibile tramite il riferimento *ClasseEsterna.this*, come nel seguente esempio:

```
class ClasseEsterna
{ int i;
  int j = 3;
  // ...
  class ClasseInterna
  { int i = 5;
    // ...
    void fai()
  }
}
```



```

    { i++;                // i di ClasseInterna
      j++;                // j di ClasseEsterna
      this.i++;           // i di ClasseInterna
      ClasseEsterna.this.i++; // i di ClasseEsterna
      // ClasseEsterna.i++;   errore
      // nella classe esterna,
      // i e` una variabile non statica
      // e non può essere individuata da ClasseEsterna
    }
  }
  ClasseInterna ci = new ClasseInterna();
  void elabora()
  { ClasseInterna cl = new ClasseInterna();
    cl.fai();        // fai() di ClasseInterna
    cl.i = 10;       // i di ClasseInterna
  }
}

```

Le classi interne possono avere più livelli di annidamento: in ogni caso, esse possiedono i riferimenti della classe appena più esterna, sia *ClasseEsterna* (*ClasseEsterna.this*), di quella ancora più esterna, sia *ClasseEEsterna* (*ClasseEEsterna.ClasseEsterna.this*), e così via.

Occorre precisare che una classe interna, possedendo il riferimento *ClasseEsterna.this* (che ha un valore solo nelle istanze della classe esterna), , può essere istanziata solo in una istanza della classe esterna stessa, e non può avere membri statici. Una istanza della classe esterna non comporta, peraltro, nessuna istanza della classe interna (una classe è un tipo), ma oggetti della classe interna vanno esplicitamente creati

Una classe interna costituisce quindi un tipo di oggetti dipendente dallo specifico oggetto della classe esterna, e può essere utilizzato esclusivamente in oggetti della classe esterna stessa.

Nella classe esterna, le istanze della classe interna vengono sempre ottenute utilizzando l'*operatore qualificato this.new* (*this* può essere omesso dal programmatore, ma ha comunque effetto): il riferimento *this* dell'oggetto creato assume il valore restituito dall'operatore *this.new*, mentre il riferimento *ClasseEsterna.this* dell'oggetto creato assume il valore che ha *this* nell'operatore qualificato. Il seguente esempio mette in evidenza alcune delle azioni legate alla creazione di oggetti della classe esterna e della classe interna:

```

class ClasseEsterna
{ int a;
  int b = a+1;
  class ClasseInterna
  { int c = ++b;
    ClasseInterna(int n)
    { c = n;
      Console.scriviInt(a);
      Console.scriviInt(b);
      Console.scriviIntero(c);
    }
  }
  ClasseInterna ci = this.new ClasseInterna(5);
  // anche ClasseInterna ci = new ClasseInterna(5);
  ClasseEsterna(int n)
  { a = n;
    Console.scriviInt(a);
    Console.scriviInt(b);
    Console.scriviIntero(ci.c);
  }
  void fai()
  { ClasseInterna cci = this.new ClasseInterna(7);
    // anche ClasseInterna cci = new ClasseInterna(7);
  }
}

public class ProvaInterna
{ public static void main(String[] args)
  { ClasseEsterna ce = new ClasseEsterna(10);
    ce.fai();
  }
}

```

Quando nel metodo *main()* viene creato l'oggetto riferito da *ce* (vedi anche la Fig. 9.4), *ce.a* assume valore iniziale 0, *ce.b* valore iniziale 1 e *ce.ci* un riferimento di tipo *ClasseInterna*: la valutazione dell'espressione che determina il valore iniziale di *ce.ci* comporta la creazione di un oggetto di tipo *ClasseInterna*: esso possiede il riferimento *this* (che viene memorizzato anche in *ce.ci*) e il riferimento *ClasseEsterna.this* costituito dal valore di *ce*. La creazione di tale oggetto comporta la valutazione dell'espressione che costituisce il valore iniziale di *this.c* (valore 2), con incremento di *ClasseEsterna.this.b* (il suo valore diviene 2) e l'esecuzione

del costruttore *ClasseInterna*(5) (assegnamento a *this.c* del valore 5 e stampa di *ClasseEsterna.this.a*, *ClasseEsterna.this.b* e *this.c* (*this* è quello dell'oggetto riferito da *ce.ci*). Viene quindi richiamato il costruttore *ClasseEsterna*(10), che fa assumere ad *this.a* il valore 10 (*this* è quello dell'oggetto riferito da *ce*), con la stampa di *this.a*, *this.b* e *this.ci.c*. L'esecuzione nel metodo *main()* di *ce.fai()* (vedi anche la Fig. 9.5) produce la creazione di un oggetto di tipo *ClasseInterna* riferito dalla variabile locale di *fai()* *cci*, con il valore di *ClasseEsterna.this.a* invariato, di *ClasseEsterna.this.b* incrementato (portato a 3) e di *this.c* uguale a 3 (*this* è quello dell'oggetto riferito da *cci*): il costruttore *ClasseInterna*(7) fa assumere a *this.c* il valore 7 (con la stampa di *ClasseEsterna.this.a*, *ClasseEsterna.this.b* e *this.c*). In esecuzione abbiamo pertanto:

```
0 2 5
10 2 5
10 3 7
```

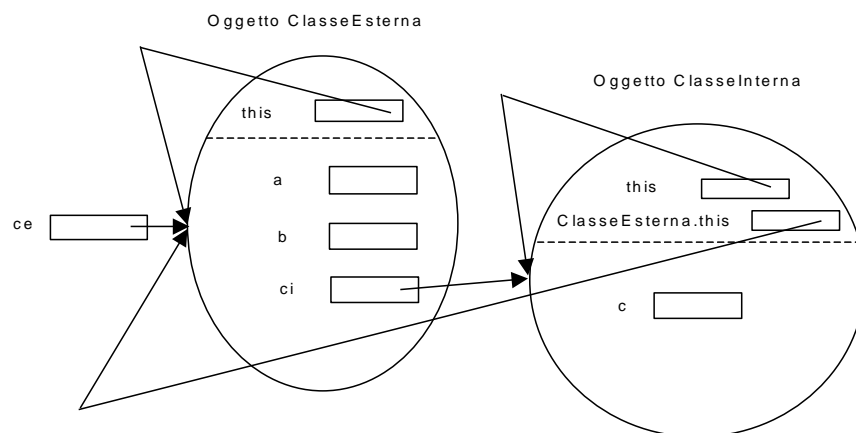


Figura 9.4. Creazione e inizializzazione di un oggetto riferito da *ce*.

Notare che al di fuori della classe esterna, un oggetto appartenente alla classe interna (se questa non è privata) può essere creato solo utilizzando un oggetto della classe esterna. L'esempio precedente può essere esteso nel seguente modo:

```

public class ProvaInternal
{
    public static void main(String[] args)
    {
        ClasseEsterna ce = new ClasseEsterna(10);
        // ...
        ClasseEsterna.ClasseInterna cci =
            ce.new ClasseInterna(6);
        // ...
    }
}

```

Nell'esempio precedente, *this* ha il valore memorizzato anche in *cci*, e *ClasseEsterna.this* il valore memorizzato anche in *ce*.

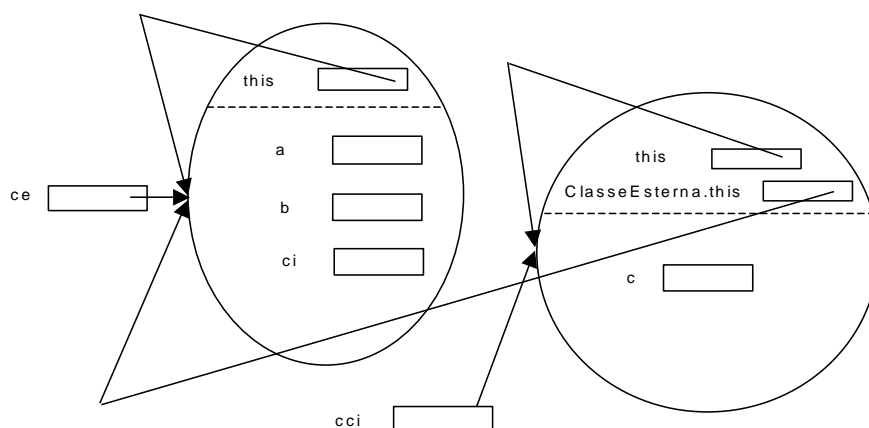


Figura 9.5. Creazione di un oggetto riferito dalla variabile locale *cci*.

### 9.4.2. Classi annidate statiche

Una classe annidata può essere definita *statica*: in questo caso non può riferire i membri non statici dell'oggetto implicito appartenente alla classe esterna (non possiede il riferimento *ClasseEsterna.this*), ferma restando la possibilità di utilizzare i membri statici e non statici di oggetti della classe esterna di cui abbia il riferimento. Nella classe annidata, i membri statici della classe esterna vengono individuati con un identificatore completo comprendente l'identificatore della classe esterna, ma il programmatore li può anche individuare col loro identificatore (tranne i casi di omonimia).

Nella classe esterna, la creazione di oggetti della classe annidata statica non implica l'uso dell'operatore qualificato *this.new*, ma del semplice operatore *new*.

Una classe annidata statica è un tipo definito nella classe esterna, indipendente dallo specifico oggetto della classe esterna stessa. Una istanza di una classe annidata statica non richiede un'istanza della classe esterna.

Al di fuori della classe esterna, una classe annidata statica (se non è privata) è visibile con un identificatore qualificato, comprendente l'identificatore della classe esterna, come nel seguente esempio:

```
class ClasseEsterna
{ static int i;
  int j;
  static class ClasseAnnidata
  { int n;
    // ...
    void fun(ClasseEsterna ce, ClasseAnnidata ca)
    { ClasseEsterna.i++;      // oppure i++;
      this.n++;              // oppure n++;
      ce.j++; ca.n++;
    }
  }
  ClasseAnnidata ca = new ClasseAnnidata();
  // non richiede l'uso di this.new
}

public class ProvaAnnidata
{ public static void main(String[] args)
  { ClasseEsterna cce = new ClasseEsterna();
    ClasseEsterna.ClasseAnnidata
      cca1 = new ClasseEsterna.ClasseAnnidata(),
      cca2 = new ClasseEsterna.ClasseAnnidata();
    cca1.fun(cce, cca2);
    // ...
  }
}
```

Il tipo di una variabile statica definita in una classe, se costituito da una classe annidata, richiede che questa sia statica (il tipo di una variabile statica non può dipendere da un oggetto istanza). Similmente, il tipo di una

variabile definita in un metodo statico, se costituito da una classe annidata, richiede che la classe stessa sia statica. Pertanto, vale quanto riportato nel seguente esempio:

```
class ClasseEsterna
{ class ClasseInterna
  { // ...
  }
  static class ClasseAnnidata
  { // ...
  }
  static ClasseAnnidata ca = new ClasseAnnidata();
  // e` errato scrivere:
  // static ClasseInterna ci = new ClasseInterna();
  static void fai()
  { ClasseAnnidata cca = new ClasseAnnidata();
    // e` errato scrivere:
    // ClasseInterna cci = new ClasseInterna();
    // ...
  }
  static void vai()
  { fai(); // anche ClasseEsterna.fai();
    // e` errato scrivere this.fai();
    // ...
  }
}
```

Le classi annidate statiche possono avere a loro volta membri statici. Più classi annidate statiche possono essere racchiuse all'interno di una stessa classe esterna, e hanno un comportamento che può essere assimilato a quello dei package.

## 9.5. Proprietà dei membri statici

I membri di una classe che possono essere dichiarati statici sono i

campi dati, i metodi e le classi annidate. La caratteristica basilare di un membro statico è quella di non richiedere nessun istanza della classe, e di essere indipendente dalla specifica istanza della classe stessa.

I campi dati statici di una classe sono propri della classe, e sono condivisi da tutti gli eventuali oggetti di quella classe. I metodi statici non possiedono il riferimento *this*, e quindi compiono azioni completamente definite nella classe e indipendenti dall'eventuale oggetto di quella classe a cui sono applicati. Le classi annidate statiche non possedendo il riferimento *ClasseEsterna.this*, e costituiscono tipi completamente definiti dalla classe esterna.

La classe costituente il programma, cui si è fatto riferimento nei Capitoli dal 4 al 6, non è stata istanziata, e i membri sono stati definiti tutti statici: i tipi sono completamente specificati nella classe, i campi dati sono variabili comuni, e i metodi, non essendo applicati a nessun oggetto, sono semplici funzioni. Si può ottenere in tal modo una metodologia di programmazione detta *procedurale*.

Come esempio, consideriamo il seguente programma, con una funzione che scambia i valori interi contenuti negli oggetti riferiti dai parametri:

```
// file ScambiaN.java
public class ScambiaN
{ static class Num
  { int x; }
  static void scambia(Num a, Num b)
  { int c;
    c = a.x; a.x = b.x; b.x = c;
  }
  public static void main(String[] args)
  { Num n1 = new Num(), n2 = new Num();
    Console.scriviStringa("Scrivi 2 interi:");
    n1.x = Console.leggiIntero();
    n2.x = Console.leggiIntero();
    scambia(n1, n2);
    // scambia a.x e b.x, ossia n1.x e n2.x
    Console.scriviStringa
      ("Risultato di scambia: " + n1.x + ' ' + n2.x);
  }
}
```

## 9.6. Classi locali

In un metodo possono essere definite classi locali, e in questo caso non possono avere nessun modificatore di accesso (quindi non possono essere statiche). Le classi locali possono utilizzare le variabili locali del metodo, nonché le variabili istanza e di classe appartenenti alla classe in cui il metodo è definito, anche se sono private. Per esempio, si può avere:

```
class MiaClasse
{ static int i = 5;
  private int j = 10;
  MiaClasse()
  { /* ... */ }
  void fun()
  { class Locale
    { private int n = i, m = j;
      // ...
    }
    Locale cl = new Locale();
    // ...
  }
}
```

## 9.7. Classi wrapper

Le classi *wrapper* sono predefinite nel linguaggio e servono a racchiudere in oggetti i valori di un tipo primitivo. Esse sono le seguenti (fanno parte del package *java.lang*):

- *Boolean*
- *Byte*
- *Character*
- *Integer*
- *Long*
- *Float*



- *Double*

Per ognuna di tali classi, è previsto un costruttore che consente di creare un oggetto classe a partire da un valore di un tipo primitivo, e un metodo per trasformare un oggetto classe in un valore di un tipo primitivo. Per esempio, per la classe *Integer* si ha:

```
Integer ( int i )  
int intValue ()
```

Analogamente, per ogni classe *wrapper* esiste un costruttore che consente di creare un oggetto classe a partire da una stringa (questa deve costituire una rappresentazione testuale valida di un valore del corrispondente tipo primitivo), e un metodo per trasformare un oggetto classe in una stringa. Per esempio, per la classe *Integer* si ha:

```
Integer ( String s )  
String toString ()
```

Infine, per ogni classe *wrapper* esistono due metodi statici che consentono di trasformare una stringa in un valore primitivo o in un oggetto classe, rispettivamente. Per esempio, per la classe *Integer* si ha:

```
static int parseInt ( String s )  
static Integer valueOf ( String s )
```

### 9.7.1. Autoboxing

Nello spirito di semplificare la gestione di alcune strutture dati (tipicamente le *collezioni*, che verranno esaminate nel volume II), avviene automaticamente la conversione di un valore di un tipo primitivo in un oggetto della corrispondente classe *wrapper* (*boxing*) e viceversa (*unboxing*): questo meccanismo (detto di *autoboxing*) viene applicato sia nella corrispondenza tra parametri formali e argomenti attuali di costruttori e metodi, sia nell'utilizzo di operatori. Per esempio, si può scrivere:

```
class Esempio  
{ Integer wi = 5;
```

```
int i;  
Esempio(int ii)  
{ wi = ii;  
  i = wi+3;  
}  
// ...  
}
```

Il meccanismo di autoboxing, nascondendo trasformazioni concettualmente rilevanti, va utilizzato con cautela.

## 9.8. Tipi enumerazione e classi

Un tipo enumerazione è in realtà un caso particolare di classe, che non può peraltro essere istanziata esplicitamente. Tale classe, se è annidata, è implicitamente statica.

Gli enumeratori sono campi dati implicitamente costanti, pubblici e statici, e sono costituiti da riferimenti di oggetti della classe che si sta definendo, opportunamente inizializzati.

Un tipo enumerazione, oltre agli enumeratori, può prevedere altri membri, come campi dati, costruttori e metodi.

Un enumeratore è un identificatore con eventuali argomenti attuali: l'identificatore rappresenta un riferimento, il cui valore è quello restituito dall'operatore *new* che invoca un costruttore con gli argomenti attuali specificati dall'enumeratore.

A titolo di esempio, possiamo scrivere:

```
// file Romani.java  
enum Romani  
{ X(10), L(50), C(100), CC(100);  
  private int val;  
  private Romani(int n)  
  { val = n; }  
  public int valore()  
  { return val; }  
}
```

Il tipo enumerazione *Romani* è equivalente alla seguente classe (ricordare che per i tipi *enum* esistono 1) il metodo *name()* che restituisce come stringa il nome di un enumeratore, e 2) il metodo statico *values()* che restituisce il riferimento di un'array i cui elementi sono gli enumeratori:

```
class Romani
{ public static final Romani X = new Romani(10),
                                L = new Romani(50),
                                C = new Romani(100),
                                M = new Romani(1000);

  private int val;
  private Romani(int n)
  { val = n; }
  public int valore()
  { return val; }
  public String name()
  { // ...
  }
  public static Romani[] values()
  { // ...
  }
}
```

Utilizzando il precedente tipo *Romani* (enumerazione) possiamo quindi scrivere la seguente classe principale:

```
// file ProvaEnum.java
import java.util.*;
class ProvaEnum
{ public static void main(String[] args)
  { Romani[] a = Romani.values();
    Romani[] b = Romani.values();
    Console.scriviBooleano(a == b);
    Console.scriviBooleano(Arrays.equals(a, b));
    Console.scriviBooleano(a[1] == b[1]);
    for (Romani r : a)
      Console.scriviIntero(r.valore());
    for (Romani r : a)
      Console.scriviStringa(r.name());
  }
}
```

In esecuzione abbiamo:

```
false
true
true
10
50
100
1000
X
L
C
M
```

## 9.9. Memoria statica, automatica, dinamica

Come risulta da quanto detto nei capitoli precedenti, ogni classe che costituisce il programma può prevedere variabili statiche e variabili istanza. Le variabili statiche vengono allocate permanentemente (per tutto il tempo di esecuzione del programma) in memoria statica, mentre le variabili istanza vengono allocate in memoria dinamica quando viene creato un oggetto classe e deallocate quando quell'oggetto classe viene distrutto dal garbage-collector. Le variabili, siano esse statiche o istanza, possono essere sia di tipo primitivo che derivato.

I metodi di una classe possono prevedere parametri e variabili locali: questi vengono allocati in memoria automatica quando inizia l'esecuzione di un metodo, e deallocati quando l'esecuzione stessa termina. La memoria automatica viene gestita con la regola della pila. Anche i parametri e le variabili locali possono essere sia di tipo primitivo che derivato.

Il codice esiste in genere in copia unica (in memoria statica), anche quando appartiene a un metodo non statico o a una classe annidata non statica (interna). Un metodo non statico, anche se concettualmente viene a far parte di ogni oggetto classe, in realtà esiste in copia unica e prevede un parametro aggiuntivo (rispetto a quelli indicati dal programmatore) che, quando il metodo viene eseguito su un oggetto, viene inizializzato con il valore di *this* di quell'oggetto. Analogamente, una classe interna, anche se

concettualmente viene a far parte di ogni oggetto classe, in realtà esiste in copia unica: i metodi della classe interna prevedono due parametri aggiuntivi, i quali, quando il metodo viene eseguito su un oggetto della classe interna stessa, vengono inizializzati con i valore di *this* e di *classeEsterna.this* di quell'oggetto.

## 10. Derivazione

### 10.1. Gerarchie di classi

In Java, le classi sono organizzate in gerarchie. Una classe può avere una *superclasse*, ossia una classe che la precede immediatamente nella gerarchia, e in questo caso si dice classe *derivata* o *sottoclasse*. È possibile che una superclasse abbia più sottoclassi, ma non è consentito che una sottoclasse abbia più superclassi (non è prevista la derivazione multipla). Ogni gerarchia di classi ha una classe *radice*, e nella gerarchia il rapporto superclasse-sottoclasse viene iterato. Per esempio, con riferimento alle figure geometriche si può avere una semplice gerarchia illustrata in Fig. 10.1.

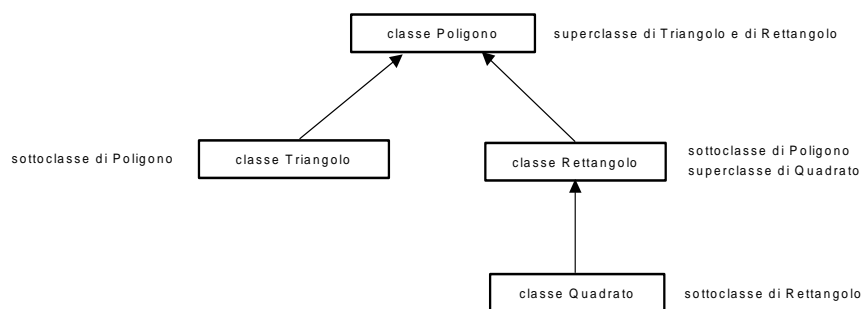


Figura 10.1. Gerarchia di classi rappresentanti figure geometriche

Una classe derivata eredita tutti i membri della sua superclasse, ed estende la superclasse stessa mediante membri aggiuntivi: essa possiede quindi due *sezioni*, una sezione superclasse e una sezione sottoclasse: la sezione superclasse è una copia del corpo della superclasse, e il relativo codice non deve essere riscritto (*riusabilità* del codice).

Sintatticamente, una superclasse e una sua sottoclasse si definiscono come nel seguente esempio:

```
class SuperClasse
{ int i;
  // ...
}

class SottoClasse extends SuperClasse
{ double d;
  // ...
}
```

Nell'esempio, la sottoclasse *SottoClasse* ha una sezione superclasse in cui è presente la variabile intera *i* e una sezione sottoclasse in cui è presente la variabile reale *d*.

Anche un oggetto sottoclasse ha due sezioni: un'istanza della sezione sottoclasse e un'istanza della sezione superclasse (ha anche tutti i membri di un oggetto superclasse). Un riferimento di un oggetto sottoclasse consente di accedere a tutti i membri dell'oggetto, a qualunque sezione appartengano, con le uniche restrizioni imposte dai modificatori di accesso dei membri (vedi sottoparagrafo 10.1.2).

Una variabile riferimento di un oggetto superclasse può assumere come valore anche il riferimento di un oggetto sottoclasse. Con questo assegnamento il riferimento, pur non subendo modifiche nel valore, cambia tipo (da riferimento di un oggetto sottoclasse a riferimento di un oggetto superclasse), e il nuovo tipo consente di accedere solo alla sezione superclasse dell'oggetto sottoclasse riferito (Fig. 10.2).

Il contrario non è ovviamente consentito: se una variabile riferimento di un oggetto sottoclasse potesse assumere come valore il riferimento di un oggetto superclasse, tale variabile consentirebbe di riferire anche la sezione sottoclasse dell'oggetto superclasse stesso, sezione peraltro non presente.

Tuttavia, l'assegnamento a una variabile sottoclasse del valore di una variabile superclasse è possibile, ma occorre utilizzare una conversione esplicita di tipo: è necessario peraltro che il programmatore sia sicuro che (con assegnamenti precedenti) tale variabile superclasse riferisca veramente un oggetto sottoclasse: se così non è, quando si tenta di accedere alla istanza della sezione sottoclasse viene generato un errore (a tempo di esecuzione).

Quanto detto trova estensione anche per una intera catena di derivazione.

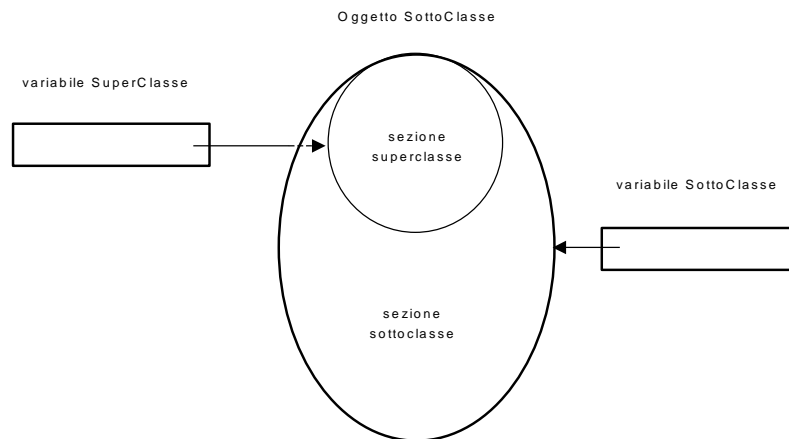


Figura 10.2. Variabili riferimento di superclasse e di sottoclasse

Per illustrare i concetti precedenti, consideriamo il seguente esempio:

```
class SuperClasse1
{ int i;
  // ...
}

class SottoClasse1 extends SuperClasse1
{ double d;
  // ...
}

public class ProvaDer1
```



```

{ public static void main(String[] args)
    { SuperClass1 spa = new SuperClass1();
      SuperClass1 spb = new SuperClass1();
      SottoClass1 sta = new SottoClass1();
      SottoClass1 stb = new SottoClass1();
      // ...
      sta.i = 5;
      sta.d = 5;
      // ...
      spa = sta;
      spa.i = 10;
      //spa.d = 10;    accesso non consentito
      //stb = spb;    assegnamento non consentito
      // ...
      spb = sta;
      stb = (SottoClass1) spb;
      stb.i = 15;
      stb.d = 15;
      // ...
    }
}

```

### 10.1.1. Costruttori

Un costruttore di una sottoclasse, per prima cosa invoca comunque un costruttore della superclasse. L'invocazione può essere fatta esplicitamente, utilizzando la parola chiave *super* con la specifica, fra parentesi tonde, dei parametri attuali richiesti dallo specifico costruttore della superclasse stessa. Se nella superclasse è presente un costruttore senza parametri, la sua invocazione può essere omessa e avviene implicitamente. Per esempio, si può avere:

```

class SuperClasse2
{ int i;
  public SuperClasse2()
  { i = 0; }
  public SuperClasse2(int a)
  { i = a; }
  // ...
}

```

```
class SottoClasse2 extends SuperClasse2
{ double d;
  public SottoClasse2(double b)
  { // chiamata implicita di SuperClasse2()
    d = b;
  }
  public SottoClasse2(int a, double b)
  { super(a);
    // chiamata esplicita di SuperClasse2(a)
    d = b;
  }
  // ...
}
```

### 10.1.2. Modificatori di accesso

Nel rispetto della regola generale, in una sottoclasse non è consentito riferire i membri definiti nella superclasse che hanno livello di protezione *private*, mentre è consentito riferirli 1) sempre se questi hanno livello di protezione *public*, 2) solo quando la sottoclasse appartiene allo stesso package se questi hanno livello di protezione *package*.

Come accennato nel Capitolo 7, in una superclasse è possibile definire un membro con livello di protezione *protected*: in questo caso il membro può essere riferito dalle sottoclassi della classe stessa, a prescindere dal package a cui appartengono, nonché dalle altre classi dello stesso package.

## 10.2. Overloading e overriding

In una sottoclasse, si può aggiungere una nuova versione di un metodo per mezzo del meccanismo di *overloading*, ossia si può utilizzare per il metodo lo stesso identificatore, cono parametri diversi in numero e/o tipo e/o ordine, come nel seguente esempio:

```
class SuperClasse3
{ // ...
```

```
    public void elabora(int i)
    { /* ... */ }
}

class SottoClasse3 extends SuperClasse3
{ private int n = 1; private double m = 2;
  // ...
  public void elabora(int i, double d)
  { int a = 0;
    elabora(n);
    // ...
    if (i == 5) elabora(n, m);
    // ...
  }
}

public class ProvaDer3
{ public static void main(String[] args)
  { int ii = 10; double dd = 20;
    SottoClasse3 st = new SottoClasse3();
    // ...
    st.elabora(ii);
    // ...
    st.elabora(ii, dd);
    // ...
  }
}
```

Come ulteriore proprietà, in una sottoclasse un qualunque membro della superclasse si può ridefinire per mezzo del cosiddetto meccanismo di *overriding*: se si tratta di una variabile, utilizzando lo stesso identificatore, se si tratta di un metodo la stessa *firma*, ossia lo stesso identificatore e lo stesso numero, tipo e ordine dei parametri formali (non confondere ridefinizione con sovrapposizione).

All'interno della sottoclasse, il membro ridefinito nasconde quello originario, ma questo si può ancora individuare attraverso il riferimento *super*.

Più precisamente, in una classe derivata la parola chiave *super* rappresenta il riferimento della sezione superclasse del generico oggetto sottoclasse, e può essere usato ogni qualvolta si renda necessario, come per

risolvere eventuali omonimie fra membri della superclasse e membri della sottoclasse. Il seguente esempio illustra quanto sopra esposto:

```
class SuperClasse4
{ int i;
  public SuperClasse4(int a)
  { i = a; }
  // ...
}

class SottoClasse4 extends SuperClasse4
{ int i;
  double d;
  public SottoClasse4()
  { super(5);
    i = 10;
    d = 20;
  }
  public void elabora()
  { int i = 15;
    super.i++; Console.scriviIntero(super.i);
    this.i++; Console.scriviIntero(this.i);
    i++; Console.scriviIntero(i);
    d++; Console.scriviReale(d);
    // ...
  }
}

public class ProvaDer4
{ public static void main(String[] args)
  { SottoClasse4 ss = new SottoClasse4();
    ss.elabora();
  }
}
```

In esecuzione, si ha il seguente risultato:

```
6
11
16
21.0.
```

In una catena di derivazione, il riferimento *super* consente in realtà di individuare la prima variabile con quel nome o il primo metodo con quella *firma* che si incontra procedendo all'indietro nella catena di derivazione, variabile o metodo definiti per la prima volta o ridefiniti.

Consideriamo un membro di una superclasse ridefinito in una sottoclasse: al di fuori della superclasse e della sottoclasse, il membro originario si può riferire (se i modificatori di accesso lo consentono), attraverso una variabile riferimento di un oggetto superclasse, e il membro ridefinito attraverso una variabile riferimento di un oggetto sottoclasse (il membro ridefinito si può anche riferire attraverso una variabile riferimento di un oggetto superclasse, in virtù del fenomeno del polimorfismo, come chiarito nel Paragrafo 10.4). A titolo di esempio, si può avere:

```
class SuperClasse5
{ // ...
    public void elabora()
    { /* ... */ }
}

class SottoClasse5 extends SuperClasse5
{ // ...
    public void elabora()
    { int a = 0;
      super.elabora();
      // elabora() di SuperClasse5
      // ...
      if ( a != 0 ) elabora();
      // elabora di SottoClasse5
      // ...
    }
}

public class ProvaDer5
{ public static void main(String[] args)
  { SuperClasse5 sp = new SuperClasse5();
    SottoClasse5 st = new SottoClasse5();
    // ...
    sp.elabora(); // elabora() di SuperClasse
    st.elabora(); // elabora() di SottoClasse
  }
}
```

### 10.2.1. Esempio con figure geometriche

L'esempio seguente si riferisce alla gerarchia di classi riportata in Fig. 10.1. In tale gerarchia, le classi *Triangolo* e *Rettangolo* possiedono tutti i membri definiti nella classe *Poligono*, e la classe *Quadrato* quelli definiti nelle classi *Rettangolo* e *Poligono*. Nella classe *Poligono* è definito il metodo *perimetro()*, mentre il metodo *area()* è definito nelle sue sottoclassi *Triangolo* e *Rettangolo*. Inoltre, nella classe *Rettangolo* è ridefinito il metodo *perimetro()*. Nella classe *Quadrato*, sottoclasse di *Rettangolo*, sono ridefiniti i metodi *perimetro()* e *area()*. L'esempio è il seguente:

```
class Punto
{ protected double x; protected double y;
  public Punto(double m, double n)
  { x = m; y = n; };
}

class Poligono
{ protected int nvertici; protected Punto[] pol;
  protected double lung (Punto p1, Punto p2)
  { double dx = p2.x - p1.x; double dy = p2.y - p1.y;
    return Math.sqrt(dx*dx + dy*dy);
  }
  public Poligono(Punto[] v, int n)
  { nvertici = n;
    pol = new Punto[nvertici];
    for (int i = 0; i < nvertici; i++)
      { pol[i] = new Punto(v[i].x, v[i].y); }
  }
  public double perimetro()
  { int j; double perim = 0;
    for (int i = 0; i < nvertici; i++)
      { j = (i + 1) % nvertici;
        perim += lung(pol[i], pol[j]);
      }
    return perim;
  }
}

class Triangolo extends Poligono
{ public Triangolo(Punto[] v) { super(v, 3); }
```

```

    public double area()
    { double p = perimetro()/2;
      double a = lung (pol[1],pol[0]);
      double b = lung (pol[2],pol[1]);
      double c = lung (pol[0],pol[2]);
      return Math.sqrt(p*(p-a)*(p-b)*(p-c));
    }
}

class Rettangolo extends Poligono
{ public Rettangolo(Punto v[]) { super(v, 4); }
  public double perimetro()
  { double base = lung(pol[1], pol[0]);
    double altezza = lung(pol[2], pol[1]);
    return ((base + altezza)*2);
  }
  public double area()
  { double base = lung(pol[1], pol[0]);
    double altezza = lung(pol[2], pol[1]);
    return (base * altezza);
  }
}

class Quadrato extends Rettangolo
{ public Quadrato(Punto v[]) { super(v); }
  public double perimetro()
  { double lato = lung(pol[1], pol[0]);
    return (lato * 4);
  }
  public double area()
  { double lato = lung(pol[1], pol[0]);
    return (lato * lato);
  }
}

public class ProvaGeom
{ public static void main (String[] args)
  { int n; Punto[] v = new Punto[4];
    Console.scriviStringa
      ("Scrivi le coordinate di un triangolo");
    for (int i = 0; i < 3; i++)
    { v[i] = new Punto
      (Console.leggiReale(),

```

```

        Console.leggiReale());
    }
    Triangolo tri= new Triangolo(v);
    Console.scriviStringa( "Triangolo: " +
        tri.perimetro() + '\t' + tri.area() );
    Console.scriviStringa
        ("Scrivi le coordinate di un rettangolo");
    for (int i = 0; i < 4; i++)
    { v[i] = new Punto
        (Console.leggiReale(),
        Console.leggiReale());
    }
    Rettangolo ret= new Rettangolo(v);
    Console.scriviStringa( "Rettangolo: " +
        ret.perimetro() + '\t' + ret.area() );
    Console.scriviStringa
        ("Scrivi le coordinate di un quadrato");
    for (int i = 0; i < 4; i++)
    { v[i] = new Punto
        (Console.leggiReale(),
        Console.leggiReale());
    }
    Quadrato qua= new Quadrato(v);
    Console.scriviStringa( "Quadrato: " +
        qua.perimetro() + '\t' + qua.area() );
}
}

```

### 10.3. Metodologie orientate agli oggetti e derivazione

Nelle metodologie orientate agli oggetti, l'astrazione sui dati costituisce il principio basilare e viene realizzata in Java tramite le classi. Inoltre, la possibilità che un oggetto possa avere al suo interno altri oggetti (relazioni del genere “ha un”) viene ottenuta in Java con variabili riferimento all'interno delle classi.

Un altro pilastro fondamentale nelle metodologie orientate agli oggetti consiste nella possibilità di realizzare relazioni fra tipo e sottotipi: tali relazioni sono del genere “è un”, nel senso che un oggetto di un sottotipo è



anche un oggetto del tipo (un rettangolo è anche un poligono, un quadrato è anche un rettangolo e un poligono), mentre non è vero il viceversa (un poligono non è un triangolo). In Java, un tipo è rappresentato da una superclasse, e i suoi sottotipi dalle sottoclassi. Una sottoclasse ha una sezione superclasse, e un riferimento di un oggetto sottoclasse consente di riferire tutti i membri dell'oggetto definiti sia nella sezione sottoclasse che nella sezione superclasse. Un riferimento di una superclasse può assumere come valore quello di un riferimento di una sottoclasse, ma non è possibile in genere fare il viceversa.

## 10.4. Polimorfismo

Come detto nel paragrafo 10.1, una variabile superclasse 1) può assumere come valore il riferimento di un oggetto superclasse consentendo di individuarne i membri, e 2) può assumere come valore il riferimento di un oggetto sottoclasse consentendo però di individuarne i membri limitatamente alla sezione superclasse. Supponiamo che nella sottoclasse venga ridefinito un membro originariamente definito nella superclasse, e che si realizzi quanto espresso al precedente punto 2): il membro che viene individuato non è quello originario, ma quello ridefinito.

Pertanto, utilizzando le ridefinizioni, con un unico comando che utilizza una variabile superclasse, viene riferito un membro specifico per ogni oggetto classe appartenente a una catena di derivazione, membro che può essere diverso da oggetto a oggetto (dipende dal tipo dell'oggetto riferito). Questo fenomeno è noto come *polimorfismo*.

Per illustrare quanto detto, consideriamo il seguente esempio (vedi anche la Figura 10.3):

```
class ClaBase
{ public void fun()
  { Console.scriviStringa("fun di ClaBase"); }
}
```

```

class ClaDeriv1 extends ClaBase
{ public void fun()
  { Console.scriviStringa("fun di ClaDeriv1");
    // nella sezione ClaBase di un oggetto ClaDeriv1,
    // fun() e` quella ridefinita in ClaDeriv1
  }
  // ...
}

```

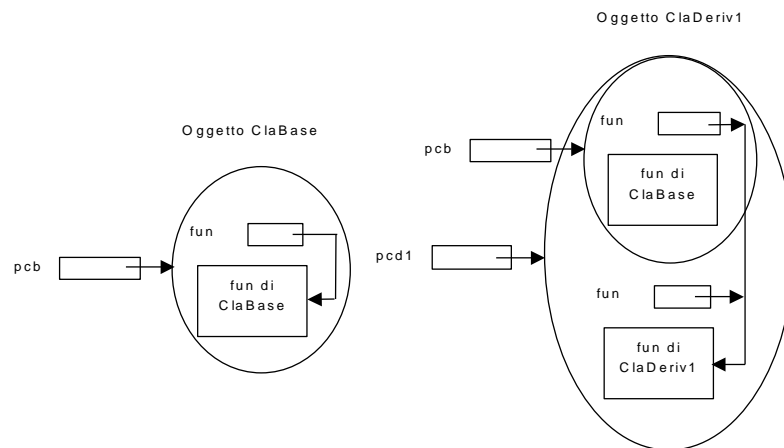


Figura 10.3. Illustrazione grafica di polimorfismo: oggetti *ClaBase* e *ClaDeriv1*

```

class ClaDeriv2 extends ClaDeriv1
{ public void fun()
  { Console.scriviStringa("fun di ClaDeriv2");
    // nella sezione ClaDeriv1 di un oggetto ClaDeriv2
    // e nella
    // sottosezione ClaBase di un oggetto ClaDeriv2
    // fun() e` quella ridefinita in ClaDeriv2
  }
  // ...
}

public class ProvaPolimorf
{ public static void main(String[] args)
  { ClaBase pcb = new ClaBase();
    ClaDeriv1 pcd1 = new ClaDeriv1();
  }
}

```

```

        ClaDeriv2 pcd2 = new ClaDeriv2();
        pcb.fun();           // fun() di ClaBase
        pcb = pcd1; pcb.fun(); // fun() di ClaDeriv1
        pcb = pcd2; pcb.fun(); // fun() di ClaDeriv2
        pcd1.fun();          // fun() di ClaDeriv1
        pcd1 = pcd2; pcd1.fun(); // fun() di ClaDeriv2 }
    }

```

#### 10.4.1. Ridefinizioni e modificatori di accesso

Un membro di una superclasse con modificatore di accesso *private* non è visibile alle sottoclassi: pertanto in una sottoclasse non può essere ridefinito, ma solo definito nuovamente.

Un membro di una superclasse con modificatore *package*, *protected* o *public* può essere invece ridefinito in una sottoclasse (nel primo caso solo in una sottoclasse dello stesso package), anche con un modificatore di accesso diverso, purché meno restrittivo (l'ordine di minor restrizione è quello precedente).

Per chiarire quanto detto, consideriamo il seguente esempio:

```

class ClaBs
{ private void fun1()
  { Console.scriviStringa("fun1 di ClaBs"); }
  public void fun2()
  { Console.scriviStringa("fun2 di ClaBs"); }
  // ...
}

class ClaDrv extends ClaBs
{ public void fun1() // nuova definizione
  { Console.scriviStringa("fun1 di ClaDrv"); }
  public void fun2() // ridefinizione
  { Console.scriviStringa("fun2 di ClaDrv"); }
  // ...
}

public class ProvaAccesso
{ public static void main(String[] args)
  { ClaBs pcb = new ClaBs();
    ClaDrv pcd = new ClaDrv();

```

```

        // pcb.fun1()           errore: fun1() privata
        pcb.fun2();             // fun2() di ClaBs
        pcd.fun1();             // fun1() di ClaDrv
        pcd.fun2();             // fun2() di ClaDrv
        pcb = pcd;
        //pcb.fun1();           errore
        pcb.fun2();             // fun2() di ClaDrv
    }
}

```

#### 10.4.2. Considerazioni sul modificatore *public*

Come visto nel Capitolo 7 (sottoparagrafo 7.13.2), una classe può essere definita pubblica (risulta visibile agli altri package), e un metodo può essere definito pubblico (risulta visibile agli altri package). Definire metodi pubblici in una classe non pubblica può sembrare privo di senso: questo però può essere utile nel caso della derivazione, come nel seguente esempio:

```

// package a, file Clab.java
package a;
public class Clab
{ public void m()
  { /* ... */ }
}

// package a, file Cla1.java
package a;
class Cla1 extends Clab
{ public void m()
  { /* ... */ }
}

// package a, file Cla2.java
package a;
public class Cla2
{ public Cla1 istanza()
  { return new Cla1(); }
}

// package b, file Prova.java

```

```
package b;
public class Prova
{ public static void main(String[] args)
  { a.Cla2 v2 = new a.Cla2();
    //a.Cla1 v1 = new a.Cla1();
    // errore: Cla1 non e' public
    a.Cla1 v = v2.istanza();
    v.m(); // m() di Cla1, classe non visibile
  }
}
```

## 10.5. Classe *Object*

Tutte le classi derivano, direttamente o indirettamente, dalla classe *Object* (package *java.lang*), che costituisce quindi la radice di una intera catena di derivazione.

Una qualunque classe eredita i metodi definiti nella classe *Object*, alcuni di carattere generale, altri specifici per la sincronizzazione fra *thread* (Capitolo 15). Alcuni metodi di carattere generale e di comune utilizzo sono i seguenti:

***protected Object clone ()***

crea una nuova istanza della classe, e ne inizializza i campi dati con quelli dell'oggetto corrente

***public int hashCode ()***

restituisce il codice interno (indirizzo codificato) dell'oggetto corrente

***public Class getClass ()***

restituisce un riferimento di tipo *Class*, che individua la classe a cui appartiene (dinamicamente) l'oggetto corrente; il nome di tale classe si ottiene (come stringa) col metodo *getName()* della classe *Class*

***protected void finalize ()***

non effettua alcuna azione: viene invocato dal Garbage Collector quando non ci sono più riferimenti dell'oggetto

***public boolean equals ( Object obj )***

restituisce *true* o *false*, rispettivamente, se i riferimenti dell'oggetto corrente e dell'oggetto parametro sono uguali o diversi

***public String toString ()***

restituisce la seguente stringa che rappresenta testualmente l'oggetto corrente (l'intero restituito da *hashCode()* viene rappresentato in esadecimale col metodo statico *toHexString()* della classe *Integer*):

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

I metodi precedenti possono venir ridefiniti, specialmente gli ultimi due.

## 10.6. Classi e metodi *final*

Una classe può essere definita facendo uso del modificatore *final*: in questo caso non può avere sottoclassi.

Una prima ragione può essere semplicemente progettuale, nel senso che nella programmazione a oggetti non si prevedono sottoclassi di una certa classe.

Il divieto di definire sottoclassi può anche essere dettato da motivi di sicurezza: una classe non può essere sostituita da una sua sottoclasse, aggiungendo al codice esistente altro codice, che potrebbe risultare potenzialmente non affidabile.

Anche un metodo può essere definito facendo uso del modificatore *final*: in questo caso il metodo non può essere ridefinito dalle sottoclassi. Per esempio, nella classe *Object*, i metodi per la sincronizzazione dei *thread* (Capitolo 15) sono *final*.

## 10.7. Classi e metodi astratti

Una classe può essere definita astratta (modificatore *abstract*): in

questo caso essa rappresenta un concetto astratto e non può essere istanziata (non possono essere definiti oggetti di quella classe, ma eventualmente solo variabili riferimento). Per esempio, si può avere:

```
abstract class OggettoGrafico
{ protected int x, y;
  public void sposta (int newX, int newY)
  { /* ... */ }
}
```

Anche un metodo può essere definito astratto: in questo caso possiede la sola intestazione e non ha corpo: si tratta quindi di una *dichiarazione* e non di una *definizione*. Un metodo astratto può far solo parte di una classe astratta (non è vero il viceversa). Per esempio, si può avere:

```
abstract class OggettoGrafico
{ protected int x, y;
  public void sposta (int newX, int newY)
  { /* ... */ }
  protected abstract void disegna();
  // ...
}
```

Una classe astratta viene usata come superclasse, per specificare campi dati e metodi di tutte le sottoclassi. Le sottoclassi o sono a loro volta classi astratte, o devono implementare tutti i metodi astratti della superclasse. Per esempio, si può avere:

```
class Cerchio extends OggettoGrafico
{ public void disegna()
  { /* ... */ }
}
class Rettangolo extends OggettoGrafico
{ public void disegna()
  { /* ... */ }
}
```

Nelle sottoclassi *Cerchio* e *Rettangolo* sono presenti le variabili *x*, *y* e i metodi *sposta()* e *disegna()*.

### 10.7.1. Classi astratte e tipi di dato astratti

Come detto nel Capitolo 7, un tipo di dato è *astratto* se i valori e le operazioni prescindono dalle modalità di realizzazione interna degli elementi.

Un tipo di dato astratto può essere realizzato con una classe astratta, nella quale vengono definiti i membri che non dipendono dall'implementazione, e solo dichiarati i metodi che dipendono dall'implementazione (metodi astratti). Vengono poi definite tante classi derivate dalla classe astratta quante sono le possibili modalità realizzative, ciascuna delle quali contiene, tra l'altro, la definizione dei metodi solo dichiarati nella classe astratta.

Un riferimento della classe astratta può assumere come valore quello restituito dall'operatore *new* applicato a una qualunque delle classi derivate, e tramite questo si possono richiamare tutte le operazioni corrispondenti a metodi definiti o solo dichiarati nella classe astratta.

Per chiarire quanto detto, consideriamo la seguente classe *Pila* relativa a un tipo di dato astratto pila:

```
abstract class Pila
{ protected int quanti;
  protected int maxElem;
  Pila(int n)
  { maxElem = n; }
  boolean empty()
  { return (quanti == 0); }
  boolean full()
  { return (quanti == maxElem); }
  abstract void push(int s);
  abstract int pop();
  abstract void stampa();
}
```

Definiamo quindi la seguente sottoclasse *PilaAr* relativa al tipo pila realizzato con array:

```
class PilaAr extends Pila
{ private int top;
  private int[] vett;
  PilaAr(int n)
```



```

    { super(n);
      top = -1; vett = new int[maxElem];
    }
    void push(int s)
    { if (quanti < maxElem)
      { quanti++; vett[++top] = s; }
    }
    int pop()
    { int s = 0;
      if (quanti > 0)
      { quanti--; s = vett[top--]; }
      return s;
    }
    void stampa()
    { if (empty()) Console.scriviStringa("Pila vuota");
      else
      { Console.scriviStringa
        ("Numero di elementi: " + (quanti));
        Console.scriviStringa("Valori:");
        for(int i=0; i<=top; i++)
          Console.scriviInt(vett[i]);
        Console.nuovaLinea();
      }
    }
  }
}

```

Definiamo infine la seguente sottoclasse *PilaLi* relativa al tipo pila realizzato con liste:

```

class PilaLi extends Pila
{ private class Elemento
  { int inf; Elemento pun;
  }
  private Elemento rpl;
  PilaLi(int n)
  { super(n); }
  void push(int s)
  { if (quanti < maxElem)
    { quanti++; Elemento rr = new Elemento();
      rr.inf = s; rr.pun = rpl; rpl = rr;
    }
  }
  int pop()

```

```

    { int s = 0;
      if (quanti > 0)
      { quanti--;
        s = rpl.inf; rpl = rpl.pun;
      }
      return s;
    }
    void stampa()
    { if (empty()) Console.scriviStringa("Pila vuota");
      else
      { Console.scriviStringa
        ("Numero di elementi: " + quanti);
        Elemento rr = rpl;
        Console.scriviStringa("Valori:");
        while (rr != null)
        { Console.scriviInt(rr.inf); rr = rr.pun; }
        Console.nuovaLinea();
      }
    }
  }
}

```

Possiamo quindi definire una classe di prova, nella quale si utilizza un riferimento di tipo *Pila*, destinato a contenere l'indirizzo o di un oggetto *PilaAr* o di un oggetto *PilaLi*:

```

public class ProvaPila
{ public static void main(String[] args)
  { int n, m, i;
    Console.scriviStringa
      ("Scrivi la dimensione della pila:");
    m = Console.leggiIntero();
    Pila pp = new PilaAr(m);
    // Pila pp = new PilaLi(m);
    Console.scriviStringa
      ("Scrivi quanti elementi vuoi inserire:");
    m = Console.leggiIntero();
    Console.scriviStringa
      ("Scrivi " + m + " numeri:");
    for (i = 0; i < m; i++)
    { n = Console.leggiIntero();
      if (!pp.full())
      { pp.push(n);
        Console.scriviStringa("Inserito " + n);
      }
    }
  }
}

```

```

    }
    else Console.scriviStringa
        ("Inserimento impossibile");
}
Console.scriviStringa
    ("Scrivi quanti elementi vuoi estrarre");
m = Console.leggiIntero();
for (i = 0; i<m; i++)
    if (!pp.empty())
    { n = pp.pop();
      Console.scriviStringa("Estratto " + n);
    }
    else Console.scriviStringa
        ("Estrazione impossibile");
Console.scriviStringa("Contenuto della pila:");
pp.stampa();
}
}

```

## 10.8. Interfacce

Un'interfaccia rappresenta un protocollo di comportamento. Essa ha la stessa struttura di una classe, ma è costituita solo da definizioni di costanti e da dichiarazioni di metodi (intestazioni senza implementazione). Come le classi, anche le interfacce godono del meccanismo di derivazione: non esiste però una radice per la gerarchia delle interfacce, come esiste per la gerarchia delle classi. Un esempio di interfaccia è il seguente:

```

public interface SuperInt
{ final int a = 10;
  // ...
}

public interface Int extends SuperInt
{ String str = "Ciao";
  // ...
  void fai(int n);
  // ...
}

```

I membri di un'interfaccia sono implicitamente pubblici. In aggiunta, le costanti sono anche statiche: per queste, i modificatori *final*, *public* e *static*, se non presenti, sono impliciti. Inoltre, i metodi sono anche astratti: per questi, i modificatori *public* e *abstract*, se non presenti, sono impliciti. Le interfacce non sono implicitamente pubbliche.

Una classe, oltre a poter avere una superclasse (una sola), può implementare una o più interfacce (parola chiave *implements* seguita dai nomi delle interfacce separati da virgola). In questo caso eredita tutte le costanti definite in tali interfacce, e, se non usa il modificatore *abstract*, deve provvedere a definire tutti i metodi (come metodi pubblici) dichiarati nelle interfacce stesse. Per esempio, si può avere:

```
class SuperCla
{ // ...
}

class Cla extends SuperCla implements Int
{ // ...
    private void elabora()
    { // ...
        Console.scriviStringa(str);
    }
    public void fai(int n)
    { /* ... */ }
    // ...
}
```

Le costanti definite in una interfaccia sono visibili anche nelle classi che non implementano quell'interfaccia, attraverso il loro nome completo. Per esempio, si può avere:

```
interface In
{ final String str = "Ciao";
  // ...
}

class UnaCla
{ // ...
    void elabora()
    { // ...
        Console.scriviStringa(In.str);
    }
}
```

```

    }
    // ...
}

```

### 10.8.1. Riferimenti di un tipo interfaccia

Il linguaggio consente di definire variabili di un tipo interfaccia (riferimenti), e di assegnare a queste riferimenti di oggetti che implementano l'interfaccia stessa, come nel seguente caso:

```

interface Inn
{ /* ... */ }

class Cla implements Inn
{ /* ... */ }

public class ProvaInt
{ public static void main(String[] args)
  { Cla cc = new Cla();
    Inn ii = cc;
    // ...
  }
}

```

Attraverso una variabile di tipo interfaccia, si possono riferire i membri di un oggetto classe che implementa quella interfaccia, non tutti ma solo quelli che compaiono nell'interfaccia stessa. Inoltre, anche per le interfacce vale il principio del polimorfismo. Quanto detto è illustrato dal seguente esempio:

```

interface Interf
{ void m();
  // ...
}

class Cla1 implements Interf
{ public void n()
  { Console.scriviStringa("Metodo n()"); }
  public void m()

```

```
        { Console.scriviStringa("Cla1"); }
        // ...
    }

class Cla2 implements Interf
{ public void m()
  { Console.scriviStringa("Cla2"); }
  // ...
}

public class ProvaInterf
{ public static void main(String[] args)
  { Cla1 ccl = new Cla1();
    ccl.n();          // stampa Metodo n()
    Interf in = new Cla1();
    //in.n();         errore: n() non dichiarato in Interf
    in.m();           // stampa Cla1
    in = new Cla2();
    in.m();           // stampa Cla2
    // ...
  }
}
```

## 11. Eccezioni

### 11.1. Lancio, cattura e gestione delle eccezioni

Una eccezione è un evento anomalo (non grave) che si verifica in fase di esecuzione di un programma. Essa viene:

- *lanciata* (o sollevata) quando si verifica l'evento;
- *catturata* e *gestita*, oppure propagata al chiamante.

Il lancio di una eccezione avviene in un blocco di elaborazione, mentre la cattura avviene in un blocco di gestione: il blocco di elaborazione e il blocco di gestione sono distinti.

In Java, le eccezioni sono oggetti della classe *Exception* o di classi derivate da *Exception*. La classe *Exception* deriva dalla classe *Throwable* che, come tutte le classi, deriva a sua volta dalla classe *Object*.

La classe *Exception*, definita nel package *java.lang*, contiene come campo dati una variabile stringa utilizzata per memorizzare un messaggio. Tale classe prevede un costruttore senza parametri (che costruisce un'eccezione con messaggio vuoto) e un costruttore con un parametro formale *msg* di tipo *String* (che costruisce un'eccezione con il messaggio *msg*). La classe *Exception* prevede inoltre il metodo *getMessage()* (ereditato dalla classe *Throwable*) che restituisce il messaggio memorizzato.

Per compiere elaborazioni che coinvolgono eccezioni si usa l'istruzione *try-catch*:

*try-catch-instruction*

```

    try block try-rest
try-rest
    catch-part
    finally-part
    catch-part finally-part
catch-part
    catch-clause
    catch-part catch-clause
catch-clause
    catch ( formal-parameter ) block
finally-part
    finally block

```

Il blocco che compare dopo la parola chiave *try* (brevemente, blocco *try*) costituisce il blocco di elaborazione, mentre il blocco che compare in ogni clausola *catch* (brevemente, blocco *catch*) costituisce un blocco di gestione. Il parametro formale di ogni clausola *catch* è di un tipo classe, e specifica il tipo di eccezione catturata. Al momento, supponiamo che vi sia un'unica clausola *catch* e che non sia presente la parte *finally*.

Il lancio di un'eccezione avviene nel blocco *try* per mezzo dell'istruzione *throw*, che specifica (mediante un'espressione) un oggetto eccezione (esistente o creato al momento): il controllo passa allora dal blocco *try* alla parte *catch*: se l'eccezione viene catturata (il parametro formale della clausola *catch* è dello stesso tipo classe a cui appartiene l'eccezione lanciata) viene eseguito il relativo blocco *catch* e l'esecuzione prosegue in sequenza, altrimenti il controllo passa al chiamante (vedi paragrafo successivo).

Per illustrare quanto detto, consideriamo il seguente esempio

```

class ClaEcc1
{ void ff()
{ Exception ec = new Exception("EC1");
  int i = Console.leggiIntero();
  try
  { if (i == 1) throw ec;
    throw new Exception("ECn");
  }
  catch(Exception e)
  { Console.scriviStringa(e.getMessage()); }
  Console.scriviStringa("Prosecuzione funzione");
}
}

```



```
public class ProvaClaEcc1
{   public static void main (String[] args)
    {   ClaEcc1 ec = new ClaEcc1(); ec.ff();
        Console.scriviStringa("Prosecuzione programma");
    }
}
```

Quando il programma esegue il metodo *ff()* di *ec*, l'eccezione lanciata viene comunque catturata (qualunque eccezione appartiene alla classe *Exception*), e viene quindi scritto il messaggio in essa contenuto. Si ha poi la prosecuzione del metodo fino al suo termine, quindi la prosecuzione del programma fino al suo termine.

## 11.2. Propagazione delle eccezioni

Un'eccezione, se in un metodo viene sollevata e non catturata, viene trasmessa al chiamante. Se, andando all'indietro, tale eccezione non viene catturata neanche dal metodo *main()*, il programma termina in modo anomalo.

Supponiamo al momento di avere a che fare con eccezioni che devono essere necessariamente gestite (per maggiori dettagli, si veda il paragrafo 11.6). Quando un metodo non cattura tutte le possibili eccezioni che possono essere sollevate, deve darne indicazione utilizzando la parola chiave *throws* seguita dal nome delle classi a cui appartengono le eccezioni non catturate. Il seguente esempio chiarisce quanto illustrato:

```
class NuovaEcc extends Exception
{   NuovaEcc(String s)
    {   super(s); }
    // ...
}

class ClaEcc2
{   void mioMetodo() throws Exception
    {   int i= Console.leggiIntero();
        try
```

```

        {   if (i == 1) throw new NuovaEcc("aa");
            throw new Exception("bb");
            //...
        }
        catch (NuovaEcc ne)
        {   Console.scriviStringa(ne.getMessage());   }
    }
}

public class ProvaClaEcc2
{   public static void main(String[] args)
    {   ClaEcc2 ec = new ClaEcc2();
        try
        {   ec.mioMetodo();   }
        catch (Exception e)
        {   Console.scriviStringa(e.getMessage());   }
    }
}

```

Il metodo *main()* definito nella classe *ProvaClaEcc2* potrebbe non gestire eccezioni ed essere fatto nel seguente modo:

```

public class ProvaClaEcc2
{   public static void main(String[] args)
                                throws Exception
    {   ClaEcc2 ec = new ClaEcc2();
        ec.mioMetodo();
        // ...
        Console.scriviStringa("Terminazione naturale");
    }
}

```

Se il programma viene eseguito specificando in ingresso un numero diverso da 1, l'eccezione lanciata da *mioMetodo()* non viene catturata neppure dal *main()*, e il programma termina forzatamente con un messaggio del tipo (supponiamo che l'intero programma sia memorizzato nel file *ProvaClaEcc2.java*):

```

Exception in thread "main" java.lang.Exception
at ClaEcc2.mioMetodo (ProvaClaEcc2.java: 12)
at ProvaClaEcc2.main(ProvaClaEcc2.java: 24)

```

### 11.3. Gerarchia delle eccezioni e catture multiple

Come detto nel paragrafo 11.1, le eccezioni sono organizzate in una gerarchia, in quanto possono appartenere alla classe *Exception* o a classi da essa derivate. Per le eccezioni vale quindi la relazione tipo-sottotipi: ogni eccezione, oltre che appartenere al proprio sottotipo, appartiene anche ai tipi superiori nella gerarchia.

L'istruzione *try-catch* prevede la possibilità di avere più clausole *catch* poste una di seguito all'altra, con argomento formale di tipo diverso, nel seguente modo:

```
try
{ /* istruzioni di elaborazione */ }
catch(tipo-eccezione nome-eccezione)
{ /* istruzioni di gestione */ }
// ...
catch(tipo-eccezione nome-eccezione)
{ /* istruzioni di gestione */ }
```

Nel caso in cui nel blocco *try* venga sollevata un'eccezione, vengono scorse in sequenza le clausole *catch* ed eseguita quella (eventuale) che ha un argomento formale del tipo a cui appartiene l'eccezione sollevata. Pertanto, per avere discriminazione, gli argomenti formali delle clausole *catch* devono rispettare l'ordine sottotipi-tipo.

A titolo di esempio, supponiamo di trattare la gerarchia di eccezioni riportata in Fig. 11.1:

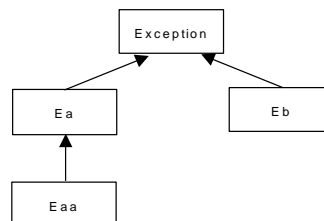


Figura 11.1. Gerarchia di eccezioni

```

class Ea extends Exception
{ /* ... */ }

class Eb extends Exception
{ /* ... */ }

class Eaa extends Ea
{ /* ... */ }

class ClaEcc3
{ void mioMetodo() throws Exception
  { int i= Console.leggiIntero();
    try
    { if(i == 1) throw new Ea();
      if(i == 2) throw new Eb();
      if(i == 3) throw new Eaa();
      throw new Exception();
    }
    catch (Eaa ne)
    { Console.scriviStringa("Tre"); }
    catch (Ea ne)
    { Console.scriviStringa("Uno"); }
    catch (Eb ne)
    { Console.scriviStringa("Due"); }
  }
}

public class ProvaClaEcc3
{ public static void main(String[] args)
  { ClaEcc3 ec = new ClaEcc3();
    try
    { ec.mioMetodo(); }
    catch (Exception e)
    { Console.scriviStringa("Propagata"); }
  }
}

```

Nella classe *ClaEcc3*, metodo *mioMetodo()*, vengono catturate, nell'ordine, eccezioni di tipo *Eaa*, *Ea* ed *Eb*: se avessimo invertito l'ordine fra *Eaa* ed *Ea*, non avremmo più potuto effettuare discriminazioni fra questi due tipi, in quanto il tipo *Ea* comprende anche il sottotipo *Eaa*. Le eccezioni di tipo *Exception* non vengono invece catturate, e vengono propagate al metodo *main()* della classe *ProvaClaEcc3*.

## 11.4. Blocco *finally*

L'istruzione *try-catch* prevede in realtà la possibilità di avere una parte *finally*. La parte *finally* viene comunque eseguita:

- sia nel caso in cui il blocco *try* non sollevi eccezioni
- sia nel caso che una eccezione sollevata venga catturata da una clausola *catch*;
- sia nel caso che una eccezione sollevata non venga catturata, ma propagata al chiamante.

La parte *finally* contiene il cosiddetto codice di pulizia che deve essere comunque eseguito, come la chiusura di un file, ed evita di dover ripetere il codice ovunque sia richiesto. Per esempio, si può avere:

```
class MiaEcc extends Exception
{   MiaEcc(String s)
    {   super(s);   }
}

class ClaEcc4
{   void fun() throws Exception
    {   MiaEcc ec = new MiaEcc("Catturata");
        int i = Console.leggiIntero();
        try
        {   if (i == 1) throw ec;
            throw new Exception("Propagata");
        }
        catch(MiaEcc e)
        {   Console.scriviStringa(e.getMessage()); }
        finally
        {   Console.scriviStringa("Uscita da fun");   }
    }
}
```

```
public class ProvaClaEcc4
{   public static void main (String[] args)
    {   ClaEcc4 ee = new ClaEcc4();
        try
        {   ee.fun();   }
        catch(Exception e)
        {   Console.scriviStringa(e.getMessage()); }
    }
}
```

In esecuzione, a seconda che venga immesso il numero 1 o un altro numero, vengono scritti in sequenza, rispettivamente, o i messaggi *Catturata* e *Uscita da fun*, oppure i messaggi *Uscita da fun* e *Propagata*.

## 11.5. Classi predefinite di eccezioni

Alcune classi di eccezioni sono predefinite nel linguaggio: eccezioni di tali classi vengono sollevate al verificarsi di alcune situazioni, senza che sia richiesto l'uso esplicito di una istruzione *throw*.

Alcune classi predefinite richiedono che le eccezioni debbano essere obbligatoriamente gestite dal programmatore (*checked*) (il compilatore controlla che ciò avvenga), mentre altre classi prevedono la possibilità che le eccezioni possano essere non gestite (*unchecked*): in questo ultimo caso, quando si verificano, vengono propagate all'indietro fino a provocare la terminazione del programma in esecuzione, con un apposito messaggio di errore.

Le eccezioni *unchecked* appartengono alla classe *Error* (è relativa ad errori non controllabili dal programmatore, come l'esaurimento della memoria disponibile) o alla classe *RuntimeException* e alle sue sottoclassi: tali classi riguardano errori numerici (come la divisione per 0), l'indice fuori limite per gli array ( $0 > \text{indice}$ , oppure  $\text{indice} \geq \text{numero-di-elementi}$ ), il valore nullo per un riferimento, eccetera.

Un semplice caso di eccezioni predefinite *unchecked* è il seguente:

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

```
{ String[] nome =  
    {"Luca ", "Marco", "Paolo", "Gianni" };  
    for (int i = 0; i < 5; i++)  
        Console.scriviStringa(nome[i]);  
}
```

In esecuzione, si ha il seguente risultato (4 è il valore raggiunto dall'indice i, 6 è la riga che ha dato luogo all'eccezione):

```
Luca  
Marco  
Paolo  
Gianni  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 4  
at Indice.main(Indice.java:6)
```

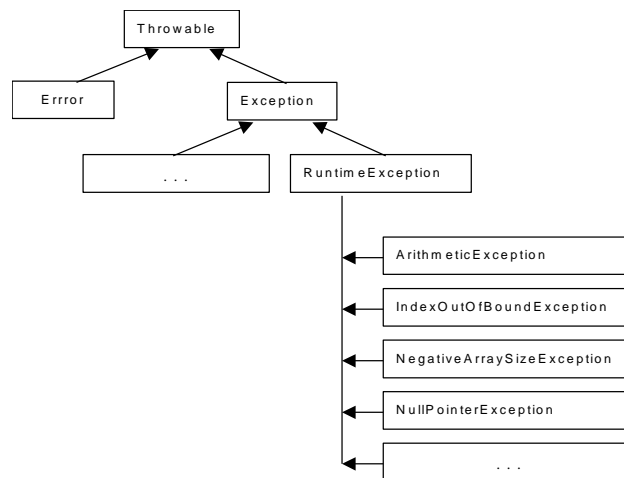


Figura 11.2. Eccezioni predefinite

Le Java API indicano, per ogni metodo di una classe, quali sono le classi a cui appartengono le possibili eccezioni predefinite e *checked*, che possono venir lanciate dal metodo e che non vengono catturate dal metodo

stesso (clausola *throws*): tali classi di eccezione devono essere pertanto esplicitamente gestite dall'utilizzatore.

## 11.6. Classe *RuntimeException*

Le eccezioni appartenenti a una sottoclasse di *RuntimeException* definita dal programmatore, analogamente alle sottoclassi predefinite di *RuntimeException*, possono essere sollevate (tramite l'istruzione *throw*) e non essere esplicitamente gestite: possono quindi non prevedere il costrutto *try-catch* (oppure, pur essendo questo presente, possono non essere catturate da nessuna clausola *catch*), e venir trasmesse al chiamante senza necessità di alcuna indicazione.

La classe *RuntimeException* fa parte del package *java.lang* e ha gli stessi costruttori della classe *Exception*.

Si consideri, per esempio, il seguente programma

```
class RuEcc extends RuntimeException
{   RuEcc(String s)
    {   super(s);   }
}

class ClaEcc5
{   void fun()
    {   RuEcc ec = new RuEcc("Propagata");
        // ...
        throw ec;
        // ...
    }
}

public class ProvaClaEcc5
{   public static void main (String[] args)
    {   ClaEcc5 ee = new ClaEcc5();
        try
        {   ee.fun();   }
        catch(RuntimeException e)
        {   Console.scriviStringa(e.getMessage()); }
    }
```



```
    }  
}
```

In esecuzione, abbiamo la stampa della stringa "**Propagata**". La classe *ProvaClaEcc5* può essere anche essere fatta nel seguente modo:

```
public class ProvaClaEcc5  
{   public static void main (String[] args)  
    {   ClaEcc5 ee = new ClaEcc5();  
        ee.fun();  
    }  
}
```

In questo caso, l'esecuzione produce il seguente risultato:

```
Eception in thread "main" RuEcc: Propagata  
at ClaEcc5.fun(ProvaClaEcc5.java:8)  
at ProvaClaEcc5.main(ProvaClaEcc5.java:18)
```

## 12. Ingresso e uscita

### 12.1. Stream

Un programma Java effettua operazioni di lettura e di scrittura utilizzando *stream*.

Uno stream è una sequenza di caselle che termina con una marca speciale (EOF). A seconda della natura di ogni casella, gli stream si dividono in *stream di caratteri* (ogni casella è in grado di contenere un valore di tipo *char*) e *stream di byte* (ogni casella è in grado di contenere un valore di tipo *byte*). Uno stream può avere un buffer e può essere associato a un dispositivo fisico (come la tastiera e il video) o a un file gestito dal sistema operativo. Una volta definito uno stream ed eseguite su esso operazioni di ingresso/uscita, lo stream può essere chiuso, e questo provoca automaticamente lo svuotamento del suo eventuale buffer: inoltre, se lo stream è associato a un dispositivo fisico o a un file, la chiusura dello stream comporta anche il rilascio del dispositivo o del file.

Ciascuno stream è un oggetto classe, e le classi relative agli stream sono contenute nel package *java.io* (una classe nel package *java.util*). Tali classi derivano dalle seguenti classi astratte:

- *Reader* e *Writer*, per stream di caratteri (o di testo);
- *InputStream* e *OutputStream*, per stream di byte (o binari).

Spesso le operazioni di ingresso/uscita interagiscono con l'utente (uomo): in questo caso un programma effettua letture di caratteri e scritture

di caratteri, utilizzando stream appartenenti a classi derivate da *Reader* o da *Writer*. Tali stream non vengono direttamente associati a dispositivi fisici o a file, perché con le tecniche attuali dispositivi e file sono normalmente composti da caselle costituite da byte e possono essere direttamente associati solo a stream appartenenti a classi derivate da *InputStream* o *OutputStream*. Vengono quindi utilizzati degli stream ponte (per esempio, quelli appartenenti alle classi *InputStreamReader* e *OutputStreamWriter*), che effettuano una conversione da byte a caratteri (lettura) e viceversa (scrittura), con regole di transcodifica che dipendono dalla specifica piattaforma utilizzata (Fig. 12.1). La transcodifica più comune, nota come *utf8*, fa corrispondere alla codifica unicode dei primi 128 caratteri la loro codifica ASCII considerando semplicemente il byte meno significativo della codifica unicode, mentre prevede corrispondenze più complesse negli altri casi (un carattere unicode corrisponde a più di un byte).

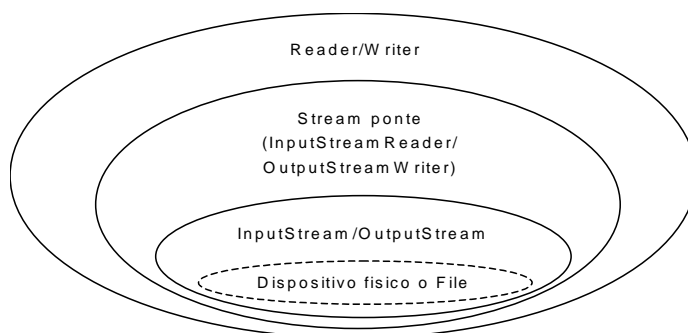


Figura 12.1. Reader/Writer e dispositivi fisici o file

Quando le operazioni di ingresso/uscita non interagiscono con l'utente, le letture e le scritture coinvolgono direttamente i singoli byte della rappresentazione interna di ciascuna informazione: gli stream utilizzati appartengono quindi alle categorie degli *InputStream* e degli *OutputStream* (che prendono anche il nome di stream binari), e la loro associazione a dispositivi fisici o a file avviene direttamente, senza che sia richiesto l'uso di nessun stream ponte. Notare che, nel caso in cui l'informazione sia

costituita da caratteri, invece di un semplice trasferimento su un dispositivo fisico o su un file di una sequenza di due byte in accordo alla codifica unicode, e viceversa, può aversi una transcodifica più efficiente, come la *utf8* menzionata sopra.

Nella trattazione delle operazioni di ingresso/uscita, faremo una schematizzazione che privilegia la simmetria degli stream utilizzati per le due operazioni, a scapito anche dell'efficienza, nella consapevolezza che certi costrutti potrebbero essere semplificati, ma a scapito della chiarezza ed efficacia della presentazione.

## 12.2. Stream di testo

La classe astratta *Reader* prevede, tra gli altri, i seguenti metodi:

***public int read () throws IOException***

legge un carattere e restituisce un intero nel rango 0-16383, oppure -1 se si tenta di leggere la marca di fine stream (l'intero, se non vale -1, rappresenta la codifica unicode del carattere letto).

***public abstract void close () throws IOException***

chiude lo stream.

La classe astratta *Writer* prevede, tra gli altri, i seguenti metodi:

***public void write ( int c ) throws IOException***

scrive il carattere la cui codifica unicode è costituita dai 16 bit meno significativi dell'intero *c*.

***void flush () throws IOException***

svuota l'eventuale buffer associato allo stream.

***void close () throws IOException***

chiude lo stream dopo aver svuotato l'eventuale buffer associato allo stream.

Notare che tutti i metodi precedenti generano eccezioni nel caso di errori di I/O.

Fra le classi derivate da *Reader* e da *Writer*, rispettivamente, le più semplici (che non siano classi astratte) sono *BufferedReader* e *BufferedWriter*. Una loro eventuale associazione a dispositivi fisici o a file richiede una struttura “a cipolla” del tipo di quella illustrata in Fig. 12.1.

Uno stream appartenente alla classe *BufferedReader*, oltre ai metodi previsti dalla classe *Reader*, consente di leggere linee per mezzo del seguente metodo:

***public String readln () throws IOException***

legge caratteri fino a incontrare un carattere di fine linea ('\r', '\n' o entambi) e restituisce la stringa di caratteri letti (senza il carattere di fine linea), o il valore *null* se si tenta di leggere la marca di fine stream.

Uno stream appartenente alla classe *BufferedWriter*, oltre ai metodi previsti dalla classe *Writer*, consente di scrivere un separatore di linea per mezzo del seguente metodo:

***public void newLine () throws IOException***

scrive un separatore di linea (è una proprietà del sistema, in genere il carattere '\n').

Notare che gli stream di uscita con buffer, se associati a un dispositivo fisico o a un file, trasferiscono effettivamente l'informazione da stream a dispositivo o file se il buffer viene svuotato: lo svuotamento si può effettuare esplicitamente (metodo *flush()*), oppure si può ottenere chiudendo lo stream (metodo *close()*).

### 12.3. Stream standard

Gli stream standard sono tre: di *ingresso*, di *uscita* e di *errore*. Lo stream standard di ingresso è *System.in*: esso è derivato da *InputStream* ed è comunemente associato alla tastiera del terminale (a ogni carattere della tastiera corrisponde un byte dello stream): per consentire di correggere

errori di battitura, i caratteri battuti sulla tastiera vengono effettivamente a far parte dello stream *System.in* solo quando viene premuto il tasto “enter”. Lo stream standard di uscita è *System.out*: esso è derivato da *OutputStream* ed è associato comunemente al video del terminale (a ogni byte dello stream corrisponde un carattere su video): per consentire la visualizzazione dei caratteri battuti sulla tastiera, questi vengono automaticamente trasferiti in eco sullo stream standard di uscita (tale stream contiene, quindi, sia i caratteri di uscita che i caratteri eco). Lo stream standard di errore *System.err* ha le stessa struttura dello stream standard di uscita, e comunemente è anch’esso associato al video. Comunemente, gli stream standard si racchiudono in stream ponte, e quest’ultimi in stream con buffer.

### 12.3.1. Esempio

Come esempio di utilizzo degli stream standard, riportiamo il seguente schema di programma:

```
import java.io.*;
class ProgrammaIO
{ public static void main (String[] args)
  throws IOException
  // termina nel caso di eccezioni di IO
  { BufferedReader tast = new BufferedReader
    (new InputStreamReader(System.in));
    BufferedWriter vide = new BufferedWriter
    (new OutputStreamWriter(System.out));
    char cc; int ii;
    // ...
    ii = tast.read();
    cc = (char)ii;
    // ...
    vide.write(cc); vide.flush();
    // ...
  }
}
```

In esecuzione, il programma precedente produce un risultato di questo genere:

**b**  
**b**

## 12.4. Letture e scritture formattate

Le letture e scritture formattate comportano, rispettivamente, la trasformazione della rappresentazione esterna di una certa entità, tipicamente un numero, da una sequenza di caratteri (comunemente cifre con aritmetica in base dieci) in rappresentazione interna (aritmetica in base due), e viceversa. Queste trasformazioni vengono effettuate da stream appartenenti alle classi *Scanner* e *PrintWriter* (tali stream sono anche bufferizzati).

I metodi di queste classi possono generare *eccezioni*, e le azioni intraprese in caso di eccezioni sono specificate dai metodi stessi.

### 12.4.1. Classe *Scanner*

La lettura formattata di valori di un tipo primitivo (escluso il tipo *byte*) o di tipo stringa si effettuano comunemente utilizzando stream appartenenti alla classe *Scanner* (package *java.util*). Uno scanner si ottiene a partire da uno stream di classe *Reader*, in accordo al seguente costruttore:

```
public Scanner ( Reader rr )
```

I metodi previsti dalla classe *Scanner* effettuano le seguenti due azioni:

- prelevano dal reader sottostante una parola o una linea (una parola è una sequenza di caratteri che non siano spazi bianchi: vengono saltati gli eventuali spazi bianchi di testa e l'arresto avviene al primo spazio bianco incontrato, che non viene prelevato);
- trasformano la eventuale parola (con regole che dipendono dal particolare metodo) da sequenza di caratteri in rappresentazione interna (binaria), o la eventuale linea in stringa.

I metodi più utilizzati sono i seguenti:

***public boolean nextBoolean ()***

preleva una parola e la trasforma in un valore booleano.

***public char next().charAt ( 0 )***

preleva una parola e ne considera il primo carattere.

***public int nextInt ()***

preleva una parola e la trasforma in un valore intero (i numeri interi positivi vanno scritti senza il segno '+').

***public double nextDouble ()***

preleva una parola e la trasforma in un valore reale.

***public String next ()***

preleva una parola e la considera una stringa.

***public String nextLine ()***

preleva la parte rimanente della linea attuale e la considera una stringa.

I metodi della classe *Scanner* possono generare eccezioni (di classe *InputMismatchException*, se la successiva parola non è una costante del tipo, di classe *NoSuchElementException* se si è raggiunta la marca di fine stream, di classe *IllegalStateException*, se lo scanner è chiuso): queste, tuttavia, appartengono a classi derivate da *RuntimeException*, per cui non devono essere obbligatoriamente gestite.

Uno scanner può anche essere costruito a partire da uno stream di classe *InputStream* (come *System.in*), e in questo caso ha anche le funzionalità di uno stream ponte.

#### **12.4.2. Classe *PrintWriter***

La scrittura formattata di valori di un tipo primitivo (escluso il tipo *byte*) o di un tipo stringa si effettuano comunemente utilizzando stream appartenenti alla classe *PrintWriter* (package *java.io*). Un *printwriter* si



ottiene a partire da uno stream di classe *Writer*, in accordo al seguente costruttore:

***public PrintWriter ( Writer vwr )***

I metodi previsti dalla classe *PrintWriter* effettuano le seguenti due azioni:

- trasformano un valore di un tipo primitivo o di una stringa (con regole che dipendono dal particolare metodo) da rappresentazione interna (binaria) a sequenza di caratteri;
- trasferiscono tale sequenza di caratteri sul writer sottostante (eventualmente seguita da un fine linea).

I metodi più utilizzati sono i seguenti:

***public void print ( boolean b )    void println ( boolean b )***  
stampa un valore booleano (ed eventualmente un fine linea).

***public void print ( char c )        void println ( char c )***  
stampa un carattere (ed eventualmente un fine linea).

***public void print ( int i )        void println ( int i )***  
stampa un valore intero (ed eventualmente un fine linea).

***public void print ( double d )    void println ( double d )***  
stampa un valore reale (ed eventualmente un fine linea).

***public void print ( String s )    void println ( String s )***  
stampa una stringa (ed eventualmente un fine linea).

***public void println ()***  
stampa un fine linea.

La classe *PrintWriter* è una sottoclasse della classe *Writer*, per cui sono disponibili anche i metodi di quest'ultima.

Un *printwriter* può essere anche costruito a partire da uno stream di classe *OutputStream* (come *System.out* o *System.err*), e in questo caso ha anche le funzionalità di stream ponte.

Inoltre, un `printwriter` può effettuare lo svuotamento automatico del buffer (*autoflush*) quando viene eseguito un metodo *println()*, specificando in fase di costruzione, dopo il nome dello stream argomento attuale, il valore booleano *true*.

### 12.4.3. Esempio

Come esempio di utilizzo delle letture e scritture formattate, riportiamo il seguente schema di programma:

```
import java.io.*; import java.util.*;
class IOFormatto
{ public static void main (String[] args)
  { Scanner tastiera = new Scanner(System.in);
    PrintWriter video = new PrintWriter(System.out);
    // oppure, per avere autoflush:
    //      new PrintWriter(System.out, true);
    boolean bb; char cc; int ii; double dd; String ss;
    // ...
    bb = tastiera.nextBoolean();
    cc = tastiera.next().charAt(0);
    ii = tastiera.nextInt();
    dd = tastiera.nextDouble();
    ss = tastiera.next();
    // ...
    video.print(bb); video.print(' ');
        video.println(cc);
    video.print(ii); video.print(' ');
        video.println(dd);
    video.println(ss);
    video.flush();
    // puo` essere omissso in caso di autoflush
  }
}
```

In esecuzione, il programma produce un risultato di questo genere (i caratteri in corsivo riguardano i dati di ingresso):

```
true a -130 +15.3e+2 ciao
true a
```

```
-130 1530.0  
ciao
```

## 12.5. Utilizzo di file per operazioni formattate

Utilizzando stream appartenenti alle classi *Scanner* e *PrintWriter* si possono effettuare letture formattate da file e scritture formattate su file, in quanto tali stream possono essere costruiti utilizzando file. Più precisamente, un file viene comunemente individuato da un letterale stringa contenente un *pathname*, ossia 1) un eventuale percorso assoluto o relativo alla cartella corrente, e 2) un nome seguito, per alcuni sistemi (Windows) dal carattere '.' e da un'estensione (esempi: "C:/LibroJava/Volume I/miofile.txt" oppure "miofile.txt"). Un file può essere utilizzato per la costruzione di uno stream appartenente alle classi *FileReader* o *FileWriter* (anche con funzionalità di stream ponte), e questo (essendo di classe *Reader* o *Writer*, rispettivamente) può a sua volta dar luogo alla costruzione di uno scanner o di un printwriter.

Quando uno stream si costruisce a partire da un file, possono generarsi eccezioni di IO dovute all'impossibilità di individuare il file stesso. Inoltre, un *filewriter* crea il file se questo non esiste, mentre uno *filereader* presuppone che il file esista già. Infine, un *filewriter* può effettuare scritture anche alla fine del file sottostante (*append*), specificando come secondo argomento attuale del costruttore il valore booleano *true*.

Uno spezzone di programma che utilizza file può essere scritto nel seguente modo:

```
import java.io.*; import java.util.*;  
class FileTxt  
{ public static void main (String[] args)  
    throws IOException  
    { PrintWriter fw = new PrintWriter  
      (new FileWriter("miofile.txt"));  
      // oppure, per avere append:  
      // (new FileWriter("miofile.txt", true));  
      // oppure, per avere anche autoflush  
      // (new FileWriter("miofile.txt", true), true);
```

```
// il file, se non esiste viene creato
Scanner fl =
    new Scanner(new FileReader("miofile.txt"));
// il file deve gia` esistere
boolean b = true; char c = 'a';
int i = -130; double d = +15.3e+2;
String s = "ciao";
// ...
fw.print(b); fw.print(' '); fw.println(c);
fw.print(i); fw.print(' '); fw.println(d);
fw.println(s);
fw.close();
// ...
boolean bb = fl.nextBoolean();
char cc = fl.next().charAt(0);
int ii = fl.nextInt();
double dd = fl.nextDouble();
String ss = fl.next();
fl.close();
// ...
PrintWriter video = new PrintWriter(System.out);
video.println(bb);
video.println(cc);
video.println(ii);
video.println(dd);
video.println(ss);
video.flush();
}
}
```

In esecuzione, il programma precedente produce il seguente risultato:

```
true
a
-130
1530.0
ciao
```

Occorre precisare che, come nell'esempio precedente, può essere utilizzato uno stesso file per ottenere sia un `FileReader` che un `FileWriter`. In questo caso, anche se le letture e le scritture vengono effettuate su stream

diversi, interessano in realtà lo stesso file: le caselle del file coinvolte restano tuttavia indipendenti, e corrispondono a quelle del *filereader* o del *filewriter* (non è il file che ha un proprio indice di casella, ma ciascun stream ne ha uno indipendente).

## 12.6. Stream binari

Come detto nel paragrafo 12.1, tutti gli stream di byte (o stream binari) appartengono a classi derivate dalle classi astratte *InputStream* e *OutputStream*.

I metodi per la lettura da un *InputStream* e per la scrittura in un *OutputStream*, simili a quelli illustrati nel paragrafo 12.2 per le classi astratte *Reader* e *Writer*, sono illustrate di seguito.

### *InputStream*

***public abstract int read () throws IOException***

legge e restituisce il byte successivo (come intero nel rango 0-255), oppure -1 se si tenta di leggere la marca di fine stream.

***public void close () throws IOException***

chiude lo stream.

### *OutputStream*

***public abstract void write ( int b ) throws IOException***

scrive un byte (gli 8 bit meno significativi dell'intero *b*)

***public void flush () throws IOException***

svuota l'eventuale buffer associato allo stream

***public void close () throws IOException***

chiude lo stream, dopo aver svuotato l'eventuale buffer associato allo stream

## 12.7. Classi *DataInputStream* e *DataOutputStream*

Gli stream della classe *DataInputStream* si costruiscono a partire da un inputstream, e prevedono, oltre a quelli visti per gli inputstream, metodi per leggere (senza effettuare conversioni) valori di un tipo primitivo, i più utilizzati dei quali sono:

```
public final boolean readBoolean () throws IOException  
public final byte readByte () throws IOException  
public final char readChar () throws IOException  
public final int readInt () throws IOException  
public final double readDouble () throws IOException
```

Esiste anche un metodo per leggere stringhe, con una transcodifica fra i byte del datainputstream e i caratteri della stringa restituita, effettuata secondo la regola *utf8*:

```
public final String readUTF () throws IOException
```

Gli stream della classe *DataOutputStream* si costruiscono a partire da un outputstream e prevedono, oltre a quelli visti per gli outputstream, metodi per scrivere (senza effettuare conversioni) valori di un tipo primitivo, i più utilizzati dei quali sono:

```
public final void writeBoolean ( boolean b ) throws IOException  
public final void writeByte ( byte be ) throws IOException  
public final void writeChar ( char c ) throws IOException  
public final void writeInt ( int i ) throws IOException  
public final void writeDouble ( double d ) throws IOException
```

Esiste anche un metodo per scrivere stringhe, con una transcodifica fra i caratteri della stringa e i byte del dataoutputstream, effettuata secondo la regola *utf8*:

```
public final void writeUTF ( String s ) throws IOException
```

Tutti i metodi presentati possono generare eccezioni di IO. In particolare, i metodi per la lettura generano eccezioni di classe *EOFException*, sottoclasse di *IOException*, quando si tenta di leggere la marca di fine stream.

Le classi *DataInputStream* e *DataOutputStream* implementano le interfacce *DataInput* e *DataOutput*, rispettivamente, nelle quali sono dichiarati i metodi precedenti.

## 12.8. Utilizzo di file per operazioni binarie

Utilizzando stream appartenenti alle classi *DataInputStream* e *DataOutputStream* si possono effettuare letture binarie da file e scritture binarie su file, in quanto tali stream possono essere costruiti utilizzando file. Più precisamente, un file, comunemente individuato da un letterale stringa contenente un pathname, può dar luogo alla costruzione di uno stream appartenente alle classi *FileInputStream* o *FileOutputStream*, e questo, in quanto di classe *InputStream* e *OutputStream*, rispettivamente, può dar luogo alla costruzione di un *datainputstream* o in un *dataoutputstream*.

Quando uno stream racchiude un file, possono generarsi eccezioni di IO dovute all'impossibilità di individuare il file stesso. Inoltre, un *fileoutputstream* crea il file se questo non esiste, mentre un *fileinputstream* presuppone che il file esista già. Infine, un *fileoutputstream* può effettuare scritture anche alla fine del file sottostante (*append*), specificando dopo l'identificatore del file coinvolto, il valore booleano *true*.

Uno spezzone di programma che utilizza file per operazioni binarie può essere scritto nel seguente modo:

```
import java.io.*; import java.util.*;
class FileBin
{ public static void main (String[] args)
    throws IOException
    { DataOutputStream fos = new DataOutputStream
      (new FileOutputStream("miofile.dat"));
      // oppure, per avere append:
      // (new FileOutputStream("miofile.dat", true));
```

```
// il file, se non esiste viene creato
DataInputStream fis = new DataInputStream
    (new FileInputStream("miofile.dat"));
// il file deve gia` esistere
boolean b = true; byte be =127; char c = 'a';
int i = -130; double d = +15.3e+2;
String s = "ciao";
// ...
fos.writeBoolean(b);
fos.writeByte(be);
fos.writeChar(c);
fos.writeInt(i);
fos.writeDouble(d);
fos.writeUTF(s);
fos.close();
// ...
boolean bb = fis.readBoolean();
byte bbe = fis.readByte();
char cc = fis.readChar();
int ii = fis.readInt();
double dd = fis.readDouble();
String ss = fis.readUTF();
fis.close();
//
PrintWriter video = new PrintWriter(System.out);
video.println(bb);
video.println(bbe);
video.println(cc);
video.println(ii);
video.println(dd);
video.println(ss);
video.flush();
    }
}
```

In esecuzione, il programma precedente produce il seguente risultato:

```
true
127
a
-130
1530.0
ciao
```



## 12.9. Classe *Console*

Come accennato nel Capitolo 4, ai fini di una più agevole gestione delle operazioni di ingresso/uscita utilizzando tastiera e video, senza interessarci del problema delle eccezioni, abbiamo definito sia la classe *Console* che un package *IngressoUscita* contenente tale classe. Nel package *IngressoUscita*, la classe *Console* è fatta nel seguente modo:

```
package IngressoUscita;
import java.util.*;
import java.io.*;
public class Console
{ private static BufferedReader tas = new
  BufferedReader(new InputStreamReader(System.in));
  private static PrintWriter vid = new
    PrintWriter(System.out);
  public static char leggiUnCarattere()
  { try
    { int i; char c;
      i = tas.read();
      if (i == -1) throw new Exception
        ("marca EndStream");
      c = (char)i; return c;
    }
    catch(Exception e)
    { System.out.println(e.getMessage());
      return '\0';
    }
  }
  public static void scriviUnCarattere(char c)
  { vid.write(c); vid.flush(); }
  private static Scanner tastiera =
    new Scanner(System.in);
  private static PrintWriter video =
    new PrintWriter(System.out, true);
  // letture
  public static boolean leggiBooleano()
  { return tastiera.nextBoolean(); }
  public static char leggiCarattere()
  { try
```

```
{ String s = tastiera.next();
    if (s.length() > 1) throw new Exception
        ("troppi caratteri");
    return s.charAt(0);
}
catch(Exception e)
{System.out.println(e.getMessage());
    return '\0';
}
}

public static int leggiIntero()
{ return tastiera.nextInt(); }
public static double leggiReale()
{ return tastiera.nextDouble(); }
public static String leggiStringa()
{ return tastiera.next(); }
public static String leggiLinea()
{ return tastiera.nextLine(); }
// scritture di linee
public static void scriviBooleano(boolean b)
{ video.println(b); }
public static void scriviCarattere(char c)
{ video.println(c); }
public static void scriviIntero(int i)
{ video.println(i); }
public static void scriviReale(double d)
{ video.println(d); }
public static void scriviStringa(String s)
{ video.println(s); }
public static void nuovaLinea()
{ video.println(); }
// scritture di parole + spazio
public static void scriviBool(boolean b)
{ video.print(b); video.print(' ');}
public static void scriviCar(char c)
{ video.print(c); video.print(' ');}
public static void scriviInt(int i)
{ video.print(i); video.print(' ');}
public static void scriviReal(double d)
{ video.print(d); video.print(' ');}
public static void scriviStr(String s)
{ video.print(s); video.print(' ');}
// compatibilita` con le versioni precedenti
```

```

    public static boolean leggiBool()
    { return leggiBooleano(); }
    public static char leggiCar()
    { return leggiCarattere(); }
    public static int leggiInt()
    { return leggiIntero(); }
    public static double leggiReal()
    { return leggiReale(); }
    public static String leggiStr()
    { return leggiStringa(); }
}

```

Utilizzando tale package, lo schema di programma riportato nel sottoparagrafo 12.4.3 assume la seguente forma:

```

import IngressoUscita.*;
class IOFormattato1
{ public static void main (String[] args)
  { boolean bb; char cc; int ii; double dd; String ss;
    // ...
    bb = Console.leggiBooleano();
    cc = Console.leggiCarattere();
    ii = Console.leggiIntero();
    dd = Console.leggiReale();
    ss = Console.leggiStringa();
    // ...
    Console.scriviBool(bb);
                Console.scriviCarattere(cc);
    Console.scriviInt(ii); Console.scriviReale(dd);
    Console.scriviStringa(ss);
    // ...
  }
}

```

## 12.10. Classi *TextFileLettura* e *TextFileScrittura*

In modo analogo e con finalità simili a quelle della classe *Console*, abbiamo definito una classe *TextFileLettura* e una classe *TextFileScrittura*

che consentono di effettuare agevolmente operazioni formattate sui file, e gestiscono le eccezioni che si possono generare per impossibilità di individuare i file stessi. Tali classi possono essere utilizzate per sviluppare programmi con letture/scritture che interessano file di tipo testo, prima di esaminare i meccanismi di ingresso/uscita: per questo sono state anch'esse inserite nel package *IngressoUscita* menzionato nel paragrafo 12.9.

La classe *TextFileLettura* possiede il seguente costruttore:

```
public TextFileLettura nomeFile =  
    new TextFileLettura ( "pathname" );
```

Essa consente di effettuare lettura di parole o di linee e di chiudere lo stream con i seguenti metodi:

```
public boolean nomeFile.leggiBooleano ()  
public char nomeFile.leggiCarattere ()  
public int nomeFile.leggiIntero ()  
public double nomeFile.leggiReale ()  
public String nomeFile.leggiStringa ()  
public String nomeFile.leggiLinea ()  
public void nomeFile.chiudi ()
```

Tale classe è riportata di seguito:

```
package IngressoUscita;  
import java.util.*;  
import java.io.*;  
public class TextFileLettura  
{ private Scanner ingresso;  
  public TextFileLettura(String s)  
  { try  
    { ingresso = new Scanner(new FileReader(s)); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public boolean leggiBooleano()  
  { return ingresso.nextBoolean(); }  
  public char leggiCarattere()  
  { try  
    { String s = ingresso.next();  
      if (s.length() > 1) throw new Exception  
        ("troppi caratteri");
```

```

        return s.charAt(0);
    }
    catch(Exception e)
    { System.out.println(e.getMessage());
      return '\0';
    }
}
public int leggiIntero()
{ return ingresso.nextInt(); }
public double leggiReale()
{ return ingresso.nextDouble(); }
public String leggiStringa()
{ return ingresso.next(); }
public String leggiLinea()
{ return ingresso.nextLine(); }
public void chiudi()
{ ingresso.close(); }
}

```

La classe *TxtFileScrittura* possiede il seguente costruttore:

```

public TxtFileScrittura nomeFile =
        new TxtFileScrittura ( "pathname" );

```

Essa consente di effettuare scritture e di chiudere lo stream con i seguenti metodi:

-) scritture di valori più spazio

```

public void nomeFile.scriviBool ( boolean b )
public void nomeFile.scriviCar ( char c )
public void nomeFile.scriviInt ( int i )
public void nomeFile.scriviReal ( double d )
public void nomeFile.scriviStr ( String s )

```

-) scritture di linee con svuotamento del buffer

```

public void nomeFile.scriviBooleano ( boolean b )
public void nomeFile.scriviCarattere ( char c )
public void nomeFile.scriviIntero ( int i )
public void nomeFile.scriviReale ( double d )
public void nomeFile.scriviStringa ( String s )
public void nomeFile.nuovaLinea ()

```

-) chiusura dello stream

**public void** *nomeFile.chiudi* ()

Tale classe è riportata di seguito:

```
package IngressoUscita;
import java.io.*;
public class TextFileScrittura
{ private PrintWriter uscita;
  public TextFileScrittura(String s)
  { try
    { uscita = new PrintWriter(new FileWriter(s)); }
    catch (IOException e) { e.getMessage(); }
  }
  // scritte di linee
  public void scriviBooleano(boolean b)
  { uscita.println(b); }
  public void scriviCarattere(char c)
  { uscita.println(c); }
  public void scriviIntero(int i)
  { uscita.println(i); }
  public void scriviReale(double d)
  { uscita.println(d); }
  public void scriviStringa(String s)
  { uscita.println(s); }
  public void nuovaLinea()
  { uscita.println(); }
  // scritte di parole + spazio
  public void scriviBool(boolean b)
  { uscita.print(b); uscita.print(' '); }
  public void scriviCar(char c)
  { uscita.print(c); uscita.print(' '); }
  public void scriviInt(int i)
  { uscita.print(i); uscita.print(' '); }
  public void scriviReal(double d)
  { uscita.print(d); uscita.print(' '); }
  public void scriviStr(String s)
  { uscita.print(s); uscita.print(' '); }
  public void chiudi()
  { uscita.close(); }
}
```

Lo spezzone di programma riportato nel paragrafo 12.5 può quindi essere così riscritto:

```
import IngressoUscita.*;
class FileTxt1
{ public static void main (String[] args)
  { TextFileScrittura fs =
    new TextFileScrittura("miofile.txt");
    TextFileLettura fl =
    new TextFileLettura("miofile.txt");
    boolean b = true; char c = 'a';
    int i = -130; double d = 15.3e+2;
    String s = "ciao";
    // ...
    fs.scriviBool(b); fs.scriviCarattere(c);
    fs.scriviInt(i); fs.scriviReale(d);
    fs.scriviStringa(s);
    fs.chiudi();
    // ...
    boolean bb = fl.leggiBooleano();
    char cc = fl.leggiCarattere();
    int ii = fl.leggiIntero();
    double dd = fl.leggiReale();
    String ss = fl.leggiStringa();
    fl.chiudi();
    // ...
    Console.scriviBooleano(bb);
    Console.scriviCarattere(cc);
    Console.scriviIntero(ii);
    Console.scriviReale(dd);
    Console.scriviStringa(ss);
  }
}
```

## 12.11. Classi *BinFileLettura* e *BinFileScrittura*

Analogamente a quanto fatto nel paragrafo precedente, abbiamo definito una classe *BinFileLettura* e una classe *BinFileScrittura* che

consentono di effettuare agevolmente operazioni binarie sui file, e gestiscono le eccezioni che si possono generare per impossibilità di individuare i file stessi e per non congruità dei dati letti o scritti. Tali classi possono essere utilizzate per sviluppare programmi con letture/scritture che interessano file binari, prima di esaminare i meccanismi di ingresso/uscita: per questo sono state anch'esse inserite nel package *IngressoUscita* menzionato nel paragrafo 12.9.

La classe *BinFileLettura* possiede il seguente costruttore:

```
public BinFileLettura nomeFile =  
    new BinFileLettura ( "pathname " ) ;
```

Essa consente di effettuare lettura di valori di tipi primitivi e di chiudere lo stream con i seguenti metodi:

```
public boolean nomeFile.leggiBooleano ()  
public byte nomeFile.leggiByte ()  
public char nomeFile.leggiCarattere()  
public int nomeFile.leggiIntero ()  
public double nomeFile.leggiReale ()  
public String nomeFile.leggiStringaUTF ()  
public void nomeFile.chiudi ()
```

Tale classe è riportata di seguito:

```
package IngressoUscita;  
import java.io.*;  
public class BinFileLettura  
{ private DataInputStream ingresso;  
  public BinFileLettura(String s)  
  { try  
    { ingresso = new DataInputStream  
      (new FileInputStream(s));  
    }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public boolean leggiBooleano()  
  { try { return ingresso.readBoolean(); }  
    catch (IOException e)  
    { e.getMessage(); return false; }  
  }  
}
```



```

public byte leggiByte()
{ try { return ingresso.readByte(); }
  catch (IOException e)
    { e.getMessage(); return 0; }
}
public char leggiCarattere()
{ try { return ingresso.readChar(); }
  catch (IOException e)
    { e.getMessage(); return '\0'; }
}
public int leggiIntero()
{ try { return ingresso.readInt(); }
  catch (IOException e)
    { e.getMessage(); return 0; }
}
public double leggiReale()
{ try { return ingresso.readDouble(); }
  catch (IOException e)
    { e.getMessage(); return 0; }
}
public String leggiStringaUTF()
{ try { return ingresso.readUTF(); }
  catch (IOException e)
    { e.getMessage(); return "\0"; }
}
public void chiudi()
{ try { ingresso.close(); }
  catch (IOException e) { e.getMessage(); }
}
}

```

La classe *BinFileScrittura* possiede il seguente costruttore:

```

public BinFileScrittura nomeFile =
    new BinFileScrittura ( "pathname" );

```

Essa consente di effettuare scritture e di chiudere lo stream con i seguenti metodi:

```

public void nomeFile.scriviBooleano ( boolean b )
public void nomeFile.scriviByte ( byte b )
public void nomeFile.scriviCarattere ( char c )

```

```
public void nomeFile.scriviIntero ( int i )  
public void nomeFile.scriviReale ( double d )  
public void nomeFile.scriviStringaUTF ( String s )  
public void nomeFile.chiudi ()
```

Tale classe è riportata di seguito:

```
package IngressoUscita;  
import java.io.*;  
public class BinFileScrittura  
{ private DataOutputStream uscita;  
  public BinFileScrittura(String s)  
  { try  
    { uscita = new DataOutputStream  
      (new FileOutputStream(s)); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public void scriviBooleano(boolean b)  
  { try { uscita.writeBoolean(b); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public void scriviByte(byte b)  
  { try { uscita.writeByte(b); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public void scriviCarattere(char c)  
  { try { uscita.writeChar(c); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public void scriviIntero(int i)  
  { try { uscita.writeInt(i); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public void scriviReale(double d)  
  { try { uscita.writeDouble(d); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public void scriviStringaUTF(String s)  
  { try { uscita.writeUTF(s); }  
    catch (IOException e) { e.getMessage(); }  
  }  
  public void chiudi()  
  { try { uscita.close(); }  
  }
```

```
        catch (IOException e) { e.getMessage(); }  
    }  
}
```

Lo spezzone di programma riportato nel paragrafo 12.8 può quindi essere così riscritto:

```
import IngressoUscita.*;  
class FileBin1  
{ public static void main (String[] args)  
    { BinFileScrittura fs =  
        new BinFileScrittura("miofile.dat");  
      BinFileLettura fl =  
        new BinFileLettura("miofile.dat");  
      boolean b = true; byte be =127; char c = 'a';  
      int i = -130; double d = +15.3e+2;  
      String s = "ciao";  
      // ...  
      fs.scriviBooleano(b);  
      fs.scriviByte(be);  
      fs.scriviCarattere(c);  
      fs.scriviIntero(i);  
      fs.scriviReale(d);  
      fs.scriviStringaUTF(s);  
      fs.chiudi();  
      // ...  
      boolean bb = fl.leggiBooleano();  
      byte bbe = fl.leggiByte();  
      char cc = fl.leggiCarattere();  
      int ii = fl.leggiIntero();  
      double dd = fl.leggiReale();  
      String ss = fl.leggiStringaUTF();  
      fl.chiudi();  
      //  
      Console.scriviBooleano(bb);  
      Console.scriviIntero(bbe);  
      Console.scriviCarattere(cc);  
      Console.scriviIntero(ii);  
      Console.scriviReale(dd);  
      Console.scriviStringa(ss);  
    }  
}
```

## 12.12. Classe *File*

La classe *File* (package *java.io*) consente di rappresentare come oggetti i pathname che si riferiscono a file o cartelle. La classe prevede (fra gli altri) i seguenti costruttori:

```
public File ( String pathname )  
public File ( File f , String child-pathname )
```

consentono di costruire oggetti *File* a partire da un pathname assoluto o relativo, oppure da un oggetto *File* che rappresenta una cartella e da un pathname relativo alla cartella stessa.

Per le classi *FileReader* e *FileWriter*, come per le classi *FileInputStream* e *FileOutputStream*, esistono costruttori che accettano argomenti di tipo *File*. Per esempio, possiamo avere:

```
import java.io.*;  
class MioFile  
{ public static void main (String[] args)  
    throws IOException  
    { File ff = new File("C:/java");  
      File f1 = new File(ff, "miofile.txt");  
      File f2 = new File(ff, "miofile.dat");  
      // ...  
      FileWriter fw = new FileWriter(f1);  
      FileOutputStream fos = new FileOutputStream(f2);  
      // ...  
      FileReader fr = new FileReader(f1);  
      FileInputStream fis = new FileInputStream(f2);  
      // ...  
    }  
}
```

Fra i numerosi metodi della classe *File*, trovano spesso applicazione quelli che consentono di “navigare” nel file system del sistema su cui si lavora. Per esempio, i seguenti metodi:

```
public String[] list ()  
public File[] listFiles ()
```

consentono di ottenere un array di riferimenti di oggetti *String* o di oggetti *File*, rispettivamente, contenuti nella cartella a cui i metodi si applicano (*null* se nella cartella non ci sono oggetti). In genere gli elementi di tali array sono riferiti a nomi di file o cartelle che rispettano l'ordine alfabetico, anche se non vi è nessuna garanzia in tal senso.

Per conoscere se un oggetto *File* è un file o una cartella, si possono usare i seguenti metodi:

```
public boolean isFile ()  
public boolean isDirectory ()
```

Per ottenere come stringa il nome del file o della cartella corrispondente a un oggetto *File*, o il suo pathname assoluto, si possono usare i seguenti metodi:

```
public String getName ()  
public String getPath ()
```

Utilizzando i metodi precedentemente illustrati, si può fare un esempio per leggere da un file (il primo incontrato) che compare nella cartella *C:/java/Lavoro* (supposto che questa contenga file):

```
import java.io.*;  
class TuoFile  
{ public static void main (String[] args)  
    throws IOException  
    { int i;  
      File f = new File("C:/java/Lav");  
      File[] as = f.listFiles();  
      for (i = 0; i < as.length; i++)  
        if(as[i].isFile()) break;  
      String s = null;  
      if(i != as.length) s = as[i].getPath();  
      else { Console.scriviStringa("Non ci sono file");  
            System.exit(1); }  
      File fa = new File(s);  
      FileReader fl = new FileReader(fa);  
      // ...  
    }  
}
```

## 13. Letture e scritture di oggetti

### 13.1. Scrittura formattata di oggetti

Nella classe *PrintStream* sono previsti i seguenti due metodi per scrivere oggetti:

```
public void print ( Object obj )  
public void println ( Object obj )
```

Tali metodi scrivono la stringa restituita dal metodo statico *valueOf(Object obj)* della classe *String*, che è quella prodotta dal metodo *toString()* della classe a cui appartiene *obj*.

Come sappiamo, una qualunque classe deriva da *Object* ed eredita il metodo *toString()*: se applicato a un oggetto *obj*, tale metodo restituisce la stringa:

```
obj.getClass().getName() + '@' + Integer.toHexString(obj.hashCode())
```

Il metodo *getClass()* della classe *Object* restituisce la classe (tipo *Class*) a cui appartiene l'oggetto, e il metodo *getName()* della classe *Class* restituisce l'identificatore della classe stessa. Il metodo statico *toHexString()* della classe *Integer* trasforma un intero in una stringa costituita da cifre esadecimali, e il metodo *hashCode()* della classe *Object* restituisce un intero dipendente dall'indirizzo dell'oggetto in memoria.

In una classe il metodo *toString()* viene in genere ridefinito, come avviene per le classi wrapper dei tipi primitivi.

## 13.2. Serializzazione

Il meccanismo di *serializzazione* di un oggetto (che comprende sia la serializzazione che la deserializzazione) trasforma lo stato dell'oggetto in una sequenza di byte, che può essere riutilizzata per ricostruire lo stato stesso o lo stato di un altro oggetto della stessa classe (per semplicità, useremo dire che il meccanismo si applica agli oggetti, invece che ai loro stati). Per esempio:

- un oggetto può essere serializzato su un disco dalla esecuzione di un programma, ed essere deserializzato da una nuova esecuzione dello stesso programma (*Lightweight Persistence*);
- un oggetto può essere serializzato da un computer per trasferirlo a un altro, e deserializzato da quest'ultimo, utilizzando le tecniche RMI (*Remote Method Invocation*).

Un oggetto, per essere serializzabile, deve appartenere a una classe che implementa una delle seguenti due interfacce (package *java.io*):

- *Serializable*: la serializzazione avviene automaticamente;
- *Externalizable*: l'oggetto è responsabile della propria serializzazione.

Una classe che implementa l'interfaccia *Serializable* specifica semplicemente che gli oggetti della classe sono serializzabili (l'interfaccia non specifica alcun metodo). Invece, una classe che implementa l'interfaccia *Externalizable* deve definire i metodi *readExternal()* e *writeExternal()* (paragrafo 13.4).

Il meccanismo di serializzazione abilitato per una classe viene automaticamente abilitato anche per le sue sottoclassi (non è vero il contrario), e non coinvolge i campi dati statici.

In una classe serializzabile, un campo può contenere una informazione sensibile (come una *password*) che non è opportuno serializzare: in questo caso occorre dichiarare il campo *private transient*. Nella deserializzazione, un campo dichiarato *private transient* assume valore 0 (o *null*, se trattasi di riferimento).

La serializzazione si ottiene semplicemente *scrivendo* o *leggendo* un oggetto in/da un opportuno stream binario, come illustrato nel sottoparagrafo seguente. Il meccanismo è ricorsivo: se l'oggetto contiene un riferimento a un'altro oggetto, anche questo (che deve essere serializzabile) viene serializzato.

### 13.2.1. Classi *ObjectInputStream* e *ObjectOutputStream*

Per leggere o scrivere oggetti si utilizzano stream appartenenti alle classi *ObjectInputStream* o *ObjectOutputStream* (package *java.io*), derivate rispettivamente da *InputStream* e *OutputStream*.

La classe *ObjectInputStream*, oltre a quello senza parametri, possiede il seguente costruttore:

```
public ObjectInputStream ( InputStream in ) throws IOException
```

Tale classe implementa l'interfaccia *ObjectInput*, che estende l'interfaccia *DataInput*, per cui sono disponibili tutti i metodi previsti da quest'ultima (*readBoolean()*, *readByte()*, *readChar()*, *readInt()*, *readDouble()*, *readUTF()*). Esiste inoltre il seguente metodo:

```
public final Object readObject ()  
    throws ClassNotFoundException , IOException
```

deserializza l'oggetto, ed ha un comportamento ricorsivo: se nella deserializzazione vengono incontrati riferimenti ad altri oggetti, vengono deserializzati gli altri oggetti (che devono essere serializzabili); restituisce un riferimento a *Object*, che va convertito nel tipo dell'oggetto da deserializzare.

La classe *ObjectOutputStream*, oltre a quello senza parametri, possiede il seguente costruttore:



***public ObjectOutputStream ( OutputStream out ) throws IOException***

Tale classe implementa l'interfaccia *ObjectOutput*, che estende l'interfaccia *DataOutput*, per cui sono disponibili tutti i metodi previsti da quest'ultima (*writeBoolean*, *writeByte()*, *writeChar()*, *writeInt()*, *writeDouble*, *writeUTF()*). Esiste inoltre il seguente metodo:

***public final void writeObject ( Object obj ) throws IOException***

serializza l'oggetto, ed ha un comportamento ricorsivo: se nella serializzazione vengono incontrati riferimenti ad altri oggetti, vengono serializzati gli altri oggetti (devono essere serializzabili).

### 13.3. Interfaccia *Serializable*

La serializzazione/deserializzazione di oggetti appartenenti a classi che implementano l'interfaccia *Serializable* avviene automaticamente, e interessa tutti i campi della classe. La deserializzazione richiede solo un riferimento dell'oggetto da deserializzare, e non viene coinvolto alcun costruttore.

Come esempio, riportiamo il seguente programma:

```
import IngressoUscita.*;
import java.io.*;
class MiaClasse implements Serializable
{ private int i;
  private double d;
  public MiaClasse(int a, double b)
  { i = a; d = b; }
  public void stampa()
  { Console.scriviStringa
    ("Valori intero e reale: " + i + " " + d);
  }
}

public class ProvaSer
{ public static void main(String args[])
  { throws ClassNotFoundException, IOException
```

```

{ MiaClasse o1 = new MiaClasse(10, 25), o2;
  ObjectOutputStream oos = new ObjectOutputStream
    (new FileOutputStream("miofile.dat"));
  oos.writeObject(o1); oos.close();
  ObjectInputStream ois = new ObjectInputStream
    (new FileInputStream("miofile.dat"));
  o2 = (MiaClasse)ois.readObject(); ois.close();
  o2.stampa();
}
}

```

In esecuzione abbiamo il seguente risultato:

**Valori intero e reale: 10 25.0**

Il comportamento ricorsivo del meccanismo di serializzazione viene messo in evidenza dal seguente esempio, in cui si scrive e si rilegge una lista (vedi anche la Figura 13.1). Gli elementi hanno nel campo informazione il riferimento di un oggetto contenente un intero.

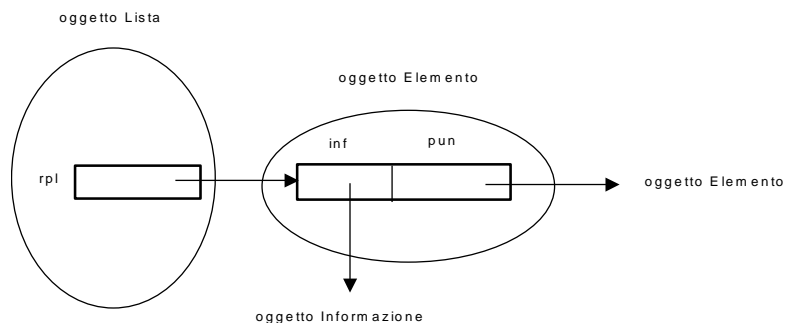


Fig. 13.1. Struttura ricorsiva di una lista

```

import IngressoUscita.*;
import java.io.*;
class Informazione implements Serializable
{ public int a; }

class Lista implements Serializable

```

```

{ private class Elemento implements Serializable
  { Informazione inf; Elemento pun; }
  private Elemento rpl;
  public boolean empty()
  { if (rpl == null) return true;
    return false;
  }
  public void insTesta(Informazione oi)
  { Elemento rr = new Elemento();
    rr.inf = oi; rr.pun = rpl; rpl = rr;
  }
  public Informazione estTesta()
  { Informazione oi = null;
    if (!empty())
      { oi = rpl.inf; rpl = rpl.pun; }
    return oi;
  }
  public void stampa()
  { Elemento rr = rpl;
    if (rpl == null)
      Console.scriviStringa("Lista vuota");
    else while (rr != null)
      { Console.scriviIntero(rr.inf.a); rr = rr.pun; }
  }
}

public class ProvaSerLista
{ public static void main(String args[])
  { throws ClassNotFoundException, IOException
    { int i; Lista li1 = new Lista();
      for (i=0; i<5; i++)
        { Informazione ii = new Informazione();
          ii.a = i; li1.insTesta(ii);
        }
      Console.scriviStringa("Lista li1:");
      li1.stampa();
      ObjectOutputStream oos = new ObjectOutputStream
        (new FileOutputStream("miofile.dat"));
      oos.writeObject(li1); oos.close();
      Lista li2;
      ObjectInputStream ois = new ObjectInputStream
        (new FileInputStream("miofile.dat"));
      li2 = (Lista)ois.readObject(); ois.close();
    }
  }
}

```

```
        Console.scriviStringa("Lista li2:");  
        li2.stampa();  
    }
```

} In esecuzione abbiamo il seguente risultato:

```
Lista li1:  
4  
3  
2  
1  
0  
Lista li2:  
4  
3  
2  
1  
0
```

### 10.3.1. Superclasse serializzabile e non serializzabile

Consideriamo il caso di una classe serializzabile, che sia una sottoclasse di una superclasse.

Se la superclasse è serializzabile, i metodi *readObject()* e *writeObject()* delle classi *ObjectInputStream* e *ObjectOutputStream* vengono applicati prima alla sezione superclasse, quindi alla sezione sottoclasse. Lo stesso meccanismo vale per una intera catena di derivazione, fino alla più alta classe serializzabile.

Se, invece la superclasse non è serializzabile, relativamente ad essa:

1. il metodo *readObject()* della classe *ObjectInputStreeam* richiama semplicemente il costruttore senza argomenti (che deve essere necessariamente presente) della superclasse stessa;
2. il metodo *writeObject()* della classe *ObjectInputStreeam* non esegue alcuna azione.

Consideriamo quindi il seguente esempio:

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

```
class Sclasse
{ private int i;
  public Sclasse()
  { i = 0; }
  public Sclasse(int a)
  { i = a; }
  public void stampa()
  { Console.scriviStringa("Valore intero: " + i); }
}

class Dclasse extends Sclasse implements Serializable
{ private double d;
  public Dclasse(int i, double b)
  { super(i); d = b; }
  public void stampa()
  { super.stampa();
    Console.scriviStringa("Valore reale: " + d);
  }
}

public class ProvaSer1
{ public static void main(String args[])
  throws ClassNotFoundException, IOException
  { Dclasse o1 = new Dclasse(10, 25), o2;
    ObjectOutputStream oos = new ObjectOutputStream
      (new FileOutputStream("miofile.dat"));
    oos.writeObject(o1); oos.close();
    // l'intero i non viene serializzato
    ObjectInputStream ois = new ObjectInputStream
      (new FileInputStream("miofile.dat"));
    o2 = (Dclasse)ois.readObject(); ois.close();
    // viene richiamato per primo il costruttore
    // default della superclasse, che pone i a 0,
    o2.stampa();
  }
}
```

In esecuzione abbiamo il seguente risultato:

```
Valore intero: 0
Valore reale: 25.0
```

## 13.4. Interfaccia *Externalizable*

Come detto all'inizio del paragrafo, una classe che implementa l'interfaccia *Externalizable* è interamente responsabile di serializzare o meno i propri campi dati, e di trattare quelli della propria eventuale superclasse. L'interfaccia *Externalizable* estende l'interfaccia *Serializable*.

I metodi dell'interfaccia *Externalizable*, che devono essere definiti da una classe che la implementa, sono i seguenti:

```
public void readExternal ( ObjectInput in )  
           throws ClassNotFoundException , IOException  
public void writeExternal ( ObjectOutputStream out ) throws IOException
```

Poiché i parametri formali dei metodi precedenti sono le interfacce *ObjectInput* e *ObjectOutput*, gli argomenti attuali possono appartenere alle classi *ObjectInputStream* e *ObjectOutputStream* rispettivamente, le quali implementano tali interfacce.

Si consideri, per esempio, il seguente programma:

```
import IngressoUscita.*;  
import java.io.*;  
class MiaClasse implements Externalizable  
{ private int i;  
  private String s;  
  // la classe String e` serializzabile  
  public MiaClasse()  
  { i = 0; s = null; }  
  public MiaClasse(int a, String st)  
  { i = a; s = st; }  
  public void readExternal(ObjectInput in)  
    throws IOException, ClassNotFoundException  
  { i = in.readInt(); s = (String)in.readObject(); }  
  public void writeExternal(ObjectOutput out)  
    throws IOException  
  { out.writeInt(i); out.writeObject(s); }  
  public void stampa()  
  { Console.scriviStringa  
    ("Intero e stringa: " + i + " " + s);  
  }  
}
```

```

public class ProvaExtern
{
    public static void main(String args[])
        throws IOException, ClassNotFoundException
    {
        MiaClasse o1 = new MiaClasse(10, "Ecco"),
        o2 = new MiaClasse();
        ObjectOutputStream oos = new ObjectOutputStream
            (new FileOutputStream("miofile.dat"));
        o1.writeExternal(oos); oos.close();
        ObjectInputStream ois = new ObjectInputStream
            (new FileInputStream("miofile.dat"));
        o2.readExternal(ois); ois.close();
        o2.stampa();
    }
}

```

In esecuzione abbiamo il seguente risultato:

**Intero e stringa: 10 Ecco**

Notare che, poiché si usano stream appartenenti alle classi *ObjectInputStream* e *ObjectOutputStream*, si possono usare i metodi *readObject()* e *writeObject()* definiti in tali classi. Se l'oggetto da serializzare/deserializzare appartiene a una classe che implementa l'interfaccia *Externalizable*, allora:

- il metodo *writeObject()* richiama il metodo *writeExternal()* dell'argomento;
- il metodo *readObject()* richiama prima il costruttore senza argomenti dell'oggetto a cui deve essere assegnato il risultato prodotto dal metodo (costruttore che deve essere obbligatoriamente presente), poi il metodo *readExternal()* dell'oggetto stesso.

Pertanto, la precedente classe *ProvaExtern* può essere scritta nel seguente modo:

```

public class ProvaExtern1
{
    public static void main(String args[])
        throws ClassNotFoundException, IOException
    {
        MiaClasse o1 = new MiaClasse(10, "Ecco"), o2;
        ObjectOutputStream oos = new ObjectOutputStream

```

```

        (new FileOutputStream("miofile.dat"));
        oos.writeObject(o1); oos.close();
        ObjectInputStream ois = new ObjectInputStream
            (new FileInputStream("miofile.dat"));
        o2 = (MiaClasse)ois.readObject(); ois.close();
        o2.stampa();
    }
}

```

In una catena di derivazione, una sottoclasse che implementa l'interfaccia *Externalizable* ha il completo controllo anche delle variabili definite nelle classi più alte (se tali variabili sono manipolabili), come nel seguente esempio:

```

import IngressoUscita.*;
import java.io.*;
class Sclasse
{ private int i;
  public Sclasse()
  { i = 0; }
  public Sclasse(int a)
  { i = a; }
  public void writeEx(ObjectOutput out)
      throws IOException
  { out.writeInt(i); }
  public void readEx(ObjectInput in)
      throws ClassNotFoundException, IOException
  { i = in.readInt(); }
  public void stampa()
  { Console.scriviStringa("Valore intero: " + i); }
}

class Dclasse extends Sclasse implements Externalizable
{ private double d;
  public Dclasse()
  { super(); d = 0; }
  public Dclasse(int i, double b)
  { super(i); d = b; }
  public void writeExternal(ObjectOutput out)
      throws IOException
  { super.writeEx(out); out.writeDouble(d); }
  public void readExternal(ObjectInput in)

```



```
        throws ClassNotFoundException, IOException
    { super.readEx(in); d = in.readDouble(); }
    public void stampa()
    { super.stampa();
      Console.scriviStringa("Valore reale: " + d);
    }
}

public class ProvaExtern2
{ public static void main(String args[])
  { throws ClassNotFoundException, IOException
    { Dclasse o1 = new Dclasse(10, 20), o2;
      ObjectOutputStream oos = new ObjectOutputStream
        (new FileOutputStream("miofile.dat"));
      oos.writeObject(o1); oos.close();
      ObjectInputStream ois = new ObjectInputStream
        (new FileInputStream("miofile.dat"));
      o2 = (Dclasse)ois.readObject(); ois.close();
      o2.stampa();
    }
  }
}
```

In esecuzione abbiamo il seguente risultato:

```
Valore intero: 10
Valore reale: 20.0
```

## 14. Generici

### 14.1. Classi generiche

In Java una classe, oltre che da un semplice identificatore, può essere individuata anche un nome, costituito da un identificatore seguito da *parametri formali* racchiusi tra parentesi angolari e separati da virgola: in quest'ultimo caso si ha una *classe generica* (formale). I parametri formali sono anch'essi costituiti da identificatori o nomi di classi, e nel caso più semplice, da un unico identificatore di classe (al momento ci riferiremo a questo caso). La visibilità dei parametri è limitata alla classe.

Un parametro formale di una classe generica può essere utilizzato nella classe come identificatore di tipo (per specificare il tipo di una variabile, il tipo di un parametro formale di un costruttore o di un metodo, e il tipo del risultato di un metodo). Lo stesso dicasi per il nome della classe generica.

Al di fuori di una classe generica, il nome della classe stessa può essere utilizzato per specificare un tipo, previa sostituzione dei parametri formali con argomenti attuali (*classe generica attuale*).

Per esempio, possiamo avere:

```
class Casella<T>                // classe generica formale
// T: parametro formale della classe generica
{ private T tt;
  Casella(T aa)
  { tt = aa; }
  T prendi()
```

```

    { return tt; }
    void metti(T aa)
    { tt = aa; }
    void copia(Casella<T> cc)
    { tt = cc.tt;
    }
}

public class ProvaCasella
{ public static void main(String args[])
  { Casella<Integer> gi = // classe generica attuale
    new Casella<Integer>(new Integer(5));
    Casella<Integer> gil = // classe generica attuale
    new Casella<Integer>(new Integer(6));
    Casella<String> gs = // classe generica attuale
    new Casella<String>(new String("Ciao"));
    // ...
    Integer ii = gi.prendi();
    String ss = gs.prendi();
    gi.copia(gil);
    // ...
  }
}

```

Esaminiamo adesso i casi più significativi fra quelli consentiti e non consentiti. In una data classe generica si possono definire variabili (riferimenti) il cui tipo sia un parametro formale (1), mentre non si possono creare oggetti di un tal tipo (non è specificato quali siano i costruttori del parametro formale). Inoltre, si possono definire variabili il cui tipo sia la classe generica stessa (2) e si possono creare oggetti di un tal tipo solo nei metodi (3) (per non avere una eccezione in esecuzione per esaurimento della memoria dinamica). Infine, si possono definire variabili e creare oggetti di una classe che contiene una variabile del tipo di un parametro formale (4).

Per esempio, si può scrivere:

```

class Gen<T>
{ private class Cla
  { T rt;
    // ...
  }
}

```

// 1

```

private T rt; // 1
// private T rtl = new T(); errore
private Gen<T> gt; // 2
// private Gen<T> gtl = new Gen<T>();
// eccezione per esaurimento della memoria dinamica
private Cla ct= new Cla(); // 4
void meto()
{ T rt; // 1
  // rt = new T(); errore
  Gen<T> gt = new Gen<T>(); // 3
  Cla ct = new Cla(); // 4
  // ...
}
}

```

Come esempio di utilizzo di una classe generica, riportiamo la definizione della classe *PilaList<T>*, idonea a rappresentare pile (per mezzo di liste) che memorizzano informazioni di tipo *T*. La definizione della classe è la seguente:

```

class PilaList<T>
{ private class Elemento
  { T inf; Elemento pun; }
  private Elemento rpl;
  boolean empty()
  { return rpl == null; }
  void push(T s)
  { Elemento rr = new Elemento();
    rr.inf = s; rr.pun = rpl; rpl = rr;
  }
  T pop()
  { T s = null;
    if (rpl != null) { s = rpl.inf; rpl = rpl.pun; }
    return s;
  }
}

public class ProvaPilaList
{ public static void main(String[] args)
  { Integer nn; String ss;
    PilaList<Integer> pli = new PilaList<Integer>();
    Integer[] ai =

```

```

        { new Integer(1), new Integer(4), new Integer(8) };
        for (int i = 0; i < 3; i++) pli.push(ai[i]);
        while (!pli.empty())
        { nn = pli.pop();
          Console.WriteLine
            ("Estratto " + nn);
        }
        PilaList<String> pls = new PilaList<String>();
        String[] as = { "Marco", "Carlo", "Luca" };
        for (int i = 0; i < 3; i++) pls.push(as[i]);
        while (!pls.empty())
        { ss = pls.pop();
          Console.WriteLine
            ("Estratto " + ss);
        }
    }
}

```

#### 14.1.1..Classi generiche e relazione tipo-sottotipo

Consideriamo una classe generica *Gen*<*T*> avente un parametro formale. Se *Cla* è superclasse di *ClaD*, non è vero che *Gen*<*Cla*> è superclasse di *Gen*<*ClaD*>, ma le due classi generiche non sono fra loro correlate. Quanto detto può essere esplicitato nel seguente modo:

```

class Gen<T>
{ // ...
}

class Cla
{ // ...
}

class ClaD extends Cla
{ // ...
}

public class ProvaD
{ public static void main(String[] args)
  { Cla cc = new Cla(); ClaD cd = new ClaD();
    Gen<Cla> gc = new Gen<Cla>();
  }
}

```

```

        Gen<ClaD> gd = new Gen<ClaD>();
        // gc = gd;      errore: tipi incompatibili
        // ...
    }
}

```

Naturalmente, una classe generica può avere una sua sottoclasse, in maniera analoga a quanto previsto per le classi normali:

```

class Cla
{ // ...
}

class Gen<T>
{ // ...
}

class GenD<T> extends Gen<T>
{ // ...
}

public class ProvaD
{ public static void main(String[] args)
  { Cla cc = new Cla();
    Gen<Cla> gc = new Gen<Cla>();
    GenD<Cla> gd = new GenD<Cla>();
    gc = gd;
    // ...
  }
}

```

## 14.2. Controllo sui tipi

Oltre a consentire di utilizzare lo stesso codice per rappresentare intere categorie di classi, le classi generiche costituiscono a tutti gli effetti nuovi tipi, e consentono di effettuare maggiori controlli sui tipi stessi a tempo di compilazione. Si consideri, per esempio, il seguente programma:

```

class Casella<T>
{ private T tt;
  Casella(T aa)
  { tt = aa; }
  T prendi()
  { return tt; }
  // ...
}

public class ProvaTipi
{ public static void main(String args[])
  { Casella<Integer> gi =
    new Casella<Integer>(new Integer(5));
    Casella<String> gs =
    new Casella<String>(new String("Ciao"));
    // ...
    // String ii = gi.prendi();    // errore
    String ss = gs.prendi();
    // ...
  }
}

```

In fase di compilazione, viene segnalato un errore (*incompatible types*). Un errore analogo (*inconvertible types*) verrebbe segnalato se usassimo una conversione di tipo, scrivendo:

```
String ii = (String)gi.prendi();    // errore
```

Il programma precedente può essere riscritto usando, invece che una classe generica, una classe normale e sostituendo il parametro classe *T* con la classe *Object*.

```

class CasellaN
{ private Object tt;
  CasellaN(Object aa)
  { tt = aa; }
  Object prendi()
  { return tt; }
  // ...
}

public class ProvaTipiN

```

```
{ public static void main(String args[])
{ CasellaN gi =
    new CasellaN(new Integer(5));
  CasellaN gs =
    new CasellaN(new String("Ciao"));
  // ...
  String ii = (String)gi.prendi(); // *
  // eccezione in esecuzione
  String ss = (String)gs.prendi();
  // ...
}
}
```

Il metodo *prendi()* produce un risultato di tipo *Object*, per cui il suo assegnamento a una variabile di un sottotipo richiede necessariamente una trasformazione di tipo. In fase di compilazione non viene segnalato nessun errore, mentre in fase di esecuzione viene generata un'eccezione (*ClassCastException*) alla riga contrassegnata con \*.

### 14.3. Classi generiche e array

Per quanto riguarda gli array, si possono definire variabili (riferimenti) di un tipo array i cui elementi siano del tipo di una classe generica o di un parametro, ma esiste la restrizione di non poter creare oggetti array i cui elementi siano di tali tipi (o di un tipo contenente una classe generica). Si hanno pertanto le seguenti segnalazioni di errore:

```
class GenArr<T>
{ private class Cla
{ T rt;
  // ...
}
private T[] ra;
// private T[] ra1 = new T[10];           errore
private GenArr<T>[] ga;
// private GenArr<T>[] ga1 = new GenArr<T>[10];
//                                     errore
```



```

private Cla[] ca;
// private Cla[] ca1 = new Cla[10];           errore
void meto()
{ T[] ra;
  // ra = new T10];                           errore
  GenArr<T>[] ga;
  // ga = new GenArr<T>[10];                  errore
  Cla[] ca;
  // ca = new Cla[10];                        errore
  // ...
}
}

public class ProvaGenArr
{ public static void main(String args[])
  { GenArr<Integer>[] aa;
    //aa = new GenArr<Integer>[10];           errore
  }
}

```

La possibilità di creare oggetti array, se consentita, darebbe luogo a possibili errori non rilevabili in fase di compilazione (la comprensione di questo fenomeno si evidenzia con le *collezioni*, che peraltro non sono incluse in questo volume).

Supponiamo di voler realizzare pile generiche utilizzando array. Viste le restrizioni esistenti, si deve ricorrere ad oggetti array i cui elementi siano di tipo *Object*, nel seguente modo:

```

class PilaArray<T>
{ private int top = -1;
  private int max;
  private T[] pp;
  PilaArray(int n)
  { max = n;
    pp = (T[])new Object[max];
    // warning: unchecked operation
  }
  boolean full()
  { return top==max-1;
  }
  boolean empty()
  { return top==-1;
  }
}

```

```

    }
    void push(T tt)
    { if(!full())
      pp[++top] = tt;
    }
    T pop()
    { T tt = null;
      if (!empty()) tt = pp[top--];
      return tt;
    }
}

```

## 14.4. Metodi generici

Oltre che tutta la classe, anche singoli metodi di una classe possono essere metodi generici: in questo caso, il parametro del metodo viene indicato fra parentesi angolari prima della sua specifica (dopo eventuali modificatori). A titolo di esempio, consideriamo il seguente caso:

```

class Casella<T>
{ private T tt;
  Casella(T n)
  { tt = n; }
  T prendi()
  { return tt; }
  void metti(T aa)
  { tt = aa; }
  // ...
}

class Utility
{ // ...
  static <T> void scambia
    (Casella<T> o1, Casella<T> o2)
  { T oo = o1.prendi(); o1.metti(o2.prendi());
    o2.metti(oo);
  }
}

```

```

public class ProvaMetodoGen
{
    public static void main(String[] args)
    {
        Casella<Integer> i1 = new Casella<Integer>
            (new Integer(4));
        Casella<Integer> i2 = new Casella<Integer>
            (new Integer(6));
        Utility.scambia(i1, i2);
        Console.scriviIntero(i1.prendi().intValue());
        Console.scriviIntero(i2.prendi().intValue());
    }
}

```

## 14.5. Classi generiche con parametri vincolati

I parametri formali di una classe generica possono essere soggetti al vincolo di essere sottoclassi di una data classe, come nel seguente esempio:

```

class Casella<T extends Number>
{
    private T tt;
    Casella(T n)
    {
        tt = n;
    }
    // ...
}

public class ProvaVincoli
{
    public static void main(String[] args)
    {
        Casella<Integer> ii = new Casella<Integer>
            (new Integer(4));
        Casella<Double> dd = new Casella<Double>
            (new Double(6.5));
        // Casella<String> ss =
        // new Casella<String>("Ciao");
        // errore: String non e' una sottoclasse di Number
    }
}

```

Notare che la classe *Integer* è una sottoclasse di *Number*, mentre la classe *String* non lo è.

## 14.6. Wildcard

Un argomento attuale di una classe generica può essere, oltre che un nome di classe, anche un cosiddetto simbolo “wildcard”, che nel caso più semplice è il punto interrogativo. Una classe generica con wildcard può costituire il tipo di una variabile (riferimento), ma non può essere istanziata.

Consideriamo una classe generica con un parametro formale *Cl<T>*, e due classi generiche attuali, una con un argomento wildcard *Cl<?>* e una con un argomento classe *Cl<MiaCla>*. *Cl<?>* risulta essere un supertipo di *Cl<MiaCla>*, nel senso che una variabile di tipo *Cl<?>* può assumere come valore quello di una variabile di tipo *Cl<MiaCla>*. Per esempio, si può avere:

```
class Casella<T>
{ private T tt;
  Casella(T aa)
  { tt = aa; }
  // ...
}

public class ProvaW
{ public static void main(String[] args)
  { Casella<?> cw;
    Casella<Integer> ci
      = new Casella<Integer>(new Integer(10));
    Casella<String> cs
      = new Casella<String>(new String("Ciao"));
    cw = ci;
    cw = cs;
    // ...
  }
}
```

Notare che la classe generica *Casella<Object>*, a differenza di *Casella<?>*, non risulta essere superclasse di *Casella<Integer>* e di *Casella<String>*.

Data una classe con wildcard, si possono non solo definire riferimenti di array di tale classe, ma anche creare oggetti array di elementi di tale classe, come nel seguente esempio:

```
class Casella<T>
{ private T tt;
  Casella(T n)
  { tt = n; }
  T prendi()
  { return tt; }
  void metti(T aa)
  { tt = aa; }
  // ...
}

public class ProvaWa
{ public static void main(String[] args)
  { Casella<?>[] cwa = new Casella<?>[10];
    Casella<Integer> ci =
      new Casella<Integer>(new Integer(10));
    Casella<String> cs =
      new Casella<String>(new String("Ciao"));
    cwa[0] = ci; cwa[1] = cs;
    // ...
  }
}
```

Un wildcard, denotando un tipo completamente indefinito, può sempre essere “catturato” da una variabile di tipo *Object*, mentre occorre una conversione di tipo per poter essere catturato da una variabile di altro tipo. Possiamo quindi aggiungere codice alla precedente classe *ProvaWa* nel seguente modo:

```
public class ProvaWa
{ public static void main(String[] args)
  { Casella<?>[] cwa = new Casella<?>[10];
    Casella<Integer> ci =
      new Casella<Integer>(new Integer(10));
```

```

        Casella<String> cs =
            new Casella<String>(new String("Ciao"));
        cwa[0] = ci; cwa[1] = cs;
        Object oo; Integer ii; String ss;
        oo = cwa[0].prendi();
        oo = cwa[1].prendi();
        ii = (Integer)cwa[0].prendi();
        ss = (String)cwa[1].prendi();
        ss = (String)cwa[0].prendi();
        // eccezione in esecuzione
        // ...
    }
}

```

In aggiunta rispetto alla forma di wildcard precedentemente vista, si possono avere classi generiche con un argomento che sia superclasse o sottoclasse di una classe specificata. Per esempio, si può avere:

```

class Cla
{ // ...
}

class Gen<T>
{ // ...
}

class Prova
{ public static void main(String[] args)
  { Gen<?> cg;
    Gen<? super Cla> cs = new Gen<Cla>();
    Gen<? extends Cla> ce = new Gen<Cla>();
    cg = cs;
    cg = ce;
    // ...
  }
}

```

## 15. Thread

### 15.1. Applicazioni Java e thread

Un'applicazione Java (programma, in senso lato) è formata da più *processi*, ciascuno costituito da una porzione di programma eseguita su un nodo della rete da una JVM attivata con il comando *java*. Più processi possono risiedere o sullo stesso nodo (questo deve essere *multiprogrammato* e i processi vengono eseguiti in *concorrenza*), o ciascuno su un nodo diverso (i processi vengono eseguiti in *parallelo*).

Ogni processo può essere costituito da uno o più *thread* (o processi leggeri), che sono flussi sequenziali di esecuzione che condividono lo stesso spazio di memoria (Fig. 15.1).

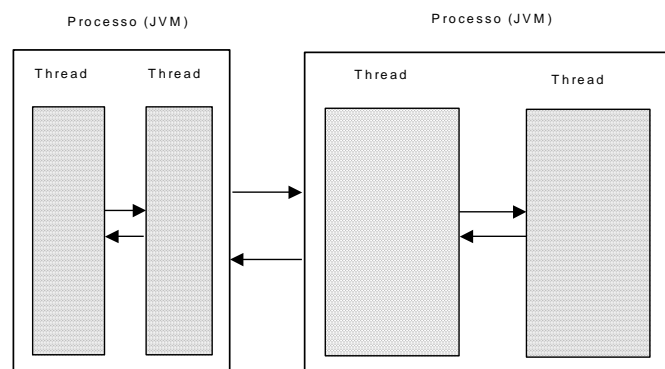


Figura 15.1. Applicazione Java

I thread di uno stesso processo vengono eseguiti in concorrenza, mediante un meccanismo di schedulazione realizzato dalla JVM (se il nodo su cui viene eseguito il processo prevede un solo processore, i thread vengono eseguiti a divisione di tempo, mentre se prevede più processori, alcuni thread possono essere eseguiti in parallelo).

Il linguaggio prevede meccanismi di comunicazione sia fra i thread di un processo che fra processi diversi: mentre le comunicazioni fra thread vengono trattate in questo capitolo, le comunicazioni tra processi fanno parte della gestione della rete e saranno esaminate nel Volume II.

Ogni processo Java ha almeno un *thread principale*, che, nel caso di un programma, effettua le elaborazioni specificate dal metodo statico *main()*, mentre, nel caso di un applet (Volume II), compie azioni indicate dal browser.

In Java, un thread (a parte il thread principale) si ottiene a partire da un oggetto appartenente a una classe derivata dalla classe *Thread*, (package *java.lang*). Invocando un costruttore viene creato un oggetto thread, mentre viene attivato un nuovo thread (flusso sequenziale) invocando il metodo (pubblico) *start()* definito nella classe *Thread*: tale metodo, dopo aver inizializzato il thread, provvede a richiamare il metodo (pubblico) *run()* dell'oggetto thread, che specifica il codice da eseguire. Anche il metodo *run()* è definito nella classe *Thread*, ma non compie alcuna azione, per cui viene comunemente ridefinito nelle sottoclassi a cui i thread appartengono (la ridefinizione deve essere necessariamente pubblica, in quanto la definizione nella classe *Thread* è pubblica).

I thread attivi vengono gestiti da uno schedulatore, che provvede ad assegnarli alle risorse di elaborazione presenti nel nodo: il metodo *start()* provvede a inserire il thread a cui è applicato nell'insieme di quelli gestiti dallo schedulatore e a invocare il metodo *run()*.

Per illustrare quanto detto, consideriamo il seguente esempio:

```
class ClaThread extends Thread
{ // ...
  public void run()
  { /* ... */ }
}

public class ProvaClaThread
{ public static void main(String[] args)
  // viene attivato ed eseguito il thread main
```



```
{  ClaThread uno = new ClaThread(),
    due = new ClaThread();
    // crea gli oggetti thread uno e due
    // ...
    uno.start(); due.start();
    // attiva i thread uno e due,
    // che vengono gestiti dallo schedulatore
    // ed eseguono il metodo run()
    // ..
}
```

## 15.2. Creazione, esecuzione e terminazione di un thread

La creazione di un thread (thread figlio) avviene ad opera di un altro thread (thread padre) che invoca un costruttore appropriato (il thread *main* è l'unico che non ha padre).

La classe *Thread* prevede i seguenti costruttori:

```
public Thread ()  
public Thread ( String id )
```

Il secondo costruttore prevede come argomento un identificatore che costituisce il nome del thread. Tale nome viene restituito dal seguente metodo:

```
public final String getName ()
```

Si può pertanto scrivere:

```
class ClaThread1 extends Thread  
{ public ClaThread1(String id)  
    { super(id);  
      // ...  
      Console.scriviStringa(getName());  
    }  
    // ...  
}
```

Esiste anche la possibilità di creare un thread che ha per argomento un riferimento all'interfaccia *Runnable* (oltre all'argomento che ne specifica il nome), come sarà chiarito nel paragrafo 15.5.

La classe *Thread* prevede il seguente metodo statico:

```
public static Thread currentThread ()
```

Il metodo restituisce il riferimento al thread attualmente in esecuzione (di questo si può conoscere il nome mediante il metodo *getName()*, come nel seguente caso:

```
class QuestaClasse  
{ // ...  
    public void questoMetodo()  
    { // ...  
        Console.scriviStringa  
            (Thread.currentThread().getName());  
    }  
    // ...  
}
```

Tale metodo viene tipicamente usato nelle classi non derivate dalla classe *Thread*, ma che possono essere utilizzate dalle classi derivate dalla classe *Thread* stessa.

I thread si suddividono in due categorie: *daemon thread* e *user thread*: la distinzione avviene quando si invoca il costruttore, che nel primo caso deve richiamare il metodo *setDaemon()* con un argomento di tipo booleano e di valore *true*, e nel secondo caso non richiamare il metodo *setDaemon()* oppure richiamarlo con un argomento booleano di valore *false*. I thread figli di un daemon thread sono anch'essi dei daemon thread.

Un user thread ha vita indipendente dal thread che lo ha creato, nel senso che termina (a meno di eccezioni) quando giunge alla fine il suo metodo *run()*. Il thread *main()* è un user thread.

Un daemon thread termina, se non è già terminato, quando tutti gli user thread sono terminati.

Un processo, per poter terminare, deve attendere che tutti i thread di cui è costituito siano terminati.

Per illustrare quanto detto, riportiamo il seguente esempio. Il thread *main* crea tre daemon thread appartenenti alla classe *ClaThread*, di nome

*uno*, *due* e *tre*, ciascun dei quali, quando è in esecuzione, scrive il suo nome. Anche il thread *main* scrive il suo nome, ma dopo 10 volte termina, e terminano quindi anche i tre thread figli. A questo punto l'intero processo (la JVM che lo esegue) termina.

```
class ClaThread2 extends Thread
{   public ClaThread2(String id)
    {   super(id); setDaemon(true);   }
    public void run()
    {   for(;;) Console.scriviStringa(getName()); }
    // ...
}

public class ProvaClaThread2
{   public static void main(String[] args)
    {   ClaThread2 uno = new ClaThread2("uno"),
        due = new ClaThread2("due"),
        tre = new ClaThread2("tre");

        // ...
        uno.start(); due.start(); tre.start();
        for(int i=0;i<10;i++)
            Console.scriviStringa
                (Thread.currentThread().getName());
    }
}
```

Una tipica esecuzione produce il seguente risultato (la JVM verifica ciclicamente, con certi intervalli di tempo, che siano terminati tutti gli user thread):

```
main
main
main
main
uno
due
tre
...
main
uno
tre
```

Ogni thread ha associata una priorità: questa influenza le scelte dello schedatore, ma non può essere utilizzata per avere garanzie sui tempi di esecuzione dei singoli thread. Per default, ogni thread viene creato con una priorità uguale a quella del thread padre; tuttavia, ogni thread può variarla con il metodo della classe *Thread*:

```
public final void setPriority ( int prio )
```

La priorità è un numero il cui valore va da 1 (priorità minima) a 10 (priorità massima) ed ha valore tipico 5 (quello del thread *main*).

### 15.3. Stati di un thread

Un thread può trovarsi in uno dei seguenti stati:

- **new**: il thread è stato creato, ma non ancora attivato (il metodo *start()* non è stato ancora richiamato);
- **runnable**: il thread è pronto e in qualunque momento può essere mandato in esecuzione dallo schedatore (il metodo *start()* è stato eseguito);
- **blocked**: il thread è bloccato in attesa di un qualche evento;
- **dead**: il thread ha terminato di eseguire il metodo *run()*.

Un thread pronto diviene bloccato (e viceversa) se:

- si addormenta (invocando il metodo *sleep(long milliseconds)*):  
il thread diviene di nuovo pronto dopo il tempo specificato;
- cerca di invocare un metodo su un oggetto gestito in mutua esclusione e occupato da un altro thread:  
il thread diviene di nuovo pronto quando la mutua esclusione sull'oggetto viene rilasciata;
- invoca il metodo *wait()* (sincronizzazione):  
il thread diviene di nuovo pronto in seguito alla esecuzione (da parte di un altro thread) del metodo *notify()* o del metodo *notifyAll()*;
- esegue una operazione di I/O:  
il thread diviene di nuovo pronto alla fine dell'operazione.

Nella classe *Thread* è presente il seguente metodo:

***public boolean isAlive ()***

Tale metodo restituisce *true* o *false* a seconda che il thread a cui si applica non si trova o si trova nello stato *dead*.

## 15.4. Segnali di interruzione

Un thread (sorgente) può inviare un segnale di interruzione a un altro thread (destinatario), eseguendo su questo il metodo della classe *Thread*:

***public void interrupt ()***

L'effetto non è quello di interrompere il thread destinatario, ma di porre ad 1 nel thread destinatario stesso il cosiddetto *flag di interruzione*. Un thread può verificare in qualunque momento lo stato del suo flag di interruzione mediante il metodo della classe *Thread*:

***public boolean isInterrupted ()***

Tale metodo restituisce *true* se il flag di interruzione vale 1 e viceversa: il valore restituito può essere utilizzato come condizione all'interno del metodo *run()*.

Come visto nel paragrafo precedente, un thread si può addormentare per un tempo prefissato invocando il metodo della classe *Thread*:

***public static void sleep ( long millisecondi )***  
***throws InterruptedException***

Se il flag di interruzione vale 1 quando il thread si addormenta, o se viene posto a 1 quando il thread è addormentato, viene lanciata una eccezione di classe *InterruptedException* (classe derivata da *Exception* e contenuta nel package *java.lang*) e il flag di interruzione viene azzerato (tale eccezione va necessariamente gestita). Il metodo *sleep()* è statico, per cui può essere

applicato anche alla classe *Thread* e addormenta il thread attualmente attivo.

In questo primo esempio, il thread *main* manda un segnale di interruzione al thread *uno*, il quale, nel metodo *run()*, esamina il flag di interruzione:

```
class MioThread extends Thread
{   public MioThread(String id, boolean b)
    {   super(id); setDaemon(b);   }
    public void run()
    {   Console.scriviStringa("Inizio " + getName());
        for(;;) if (isInterrupted()) break;
        Console.scriviStringa("Fine " + getName());
    }
}

public class ProvaInterrupt
{   public static void main(String[] args)
    {   String messaggio;
        MioThread uno = new MioThread("Uno", false),
            due = new MioThread("Due", true);
        Console.scriviStringa
            ("Per partire scrivi vai, per terminare alt");
        for (;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("vai")) break;
        }
        uno.start(); due.start();
        for(;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("alt")) break;
        }
        uno.interrupt();
        Console.scriviStringa("Due e` un Daemon Thread");
        // non occorre interrompere il thread due,
        // che termina appena e` terminato il thread main
    }
}
```

Una tipica esecuzione produce il seguente risultato:

**vai**

```

Inizio Uno
Inizio Due
alt
Due e` un daemon thread
Fine Uno

```

Nel secondo esempio, il thread *main* invia un segnale di interruzione ai thread *uno*, *due* e *tre* che ciclicamente si addormentano: questo produce il generarsi di eccezioni predefinite della classe *InterruptedException* e l'esecuzione della parte *catch* del metodo *run()* che ne determina la fine (messaggio tipico delle eccezioni generate: *sleep interrupted*):

```

class TuoThread extends Thread
{ private long intervallo;
  public TuoThread(String id, long time)
  { super(id); intervallo = time; }
  public void run()
  { try
    { for (;;)
      { Console.scriviStringa
        ("sonno " + getName());
        sleep(intervallo);
        Console.scriviStringa
        ("risveglio " + getName());
      }
    }
    catch(InterruptedException e)
    { Console.scriviStringa
      (getName() + " " + e.getMessage());
    }
  }
}

public class ProvaSleep
{ public static void main(String[] args)
  { String messaggio;
    TuoThread uno = new TuoThread("Uno", 3000),
    due = new TuoThread("Due", 3500),
    tre = new TuoThread("Tre", 4000);
    Console.scriviStringa
      ("Per partire scrivi vai, per terminare alt");
    for (;;)

```

```

        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("vai")) break;
        }
        uno.start(); due.start(); tre.start();
        for (;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("alt")) break;
        }
        uno.interrupt(); due.interrupt();
        tre.interrupt();
    }
}

```

Una tipica esecuzione produce il seguente risultato:

```

vai
sonno Uno
sonno Due
sonno Tre
risveglio Uno
sonno Uno
risveglio Due
sonno Due
alt
Uno sleep interrupted
Due sleep interrupted
Tre sleep interrupted

```

Nel terzo esempio, infine, viene illustrata la realizzazione di uno *WatchDog*, che è un thread speciale che fa partire un qualunque altro thread specificato come parametro e lo fa terminare dopo un tempo specificato come altro parametro (il metodo *run()* del thread che costituisce il parametro termina quando il suo flag di interruzione viene posto a 1 dallo *WatchDog*):

```

class SuoThread extends Thread
{   public SuoThread(String id)
    {   super(id);   }
    public void run()
    {   while(!isInterrupted())
        Console.scriviStringa
            ("In esecuzione " + getName());
    }
}

```



```
        // ...
    }
}

class WatchDog extends Thread
{   private SuoThread suot;
    private int intervallo;
    public WatchDog(SuoThread t, int ms)
    {   suot = t; intervallo = ms;
        start();
    }
    public void run()
    {   suot.start();
        try
        {   sleep(intervallo);   }
        catch (InterruptedException e) { }
        suot.interrupt();
    }
}

public class ProvaWatch
{   public static void main(String[] args)
    {   SuoThread uno = new SuoThread("Uno");
        new WatchDog(uno, 10);
        SuoThread due = new SuoThread("Due");
        new WatchDog(due, 20);
    }
}
```

Notare che la creazione di un oggetto (*new WatchDog()*) costituisce un'istruzione espressione (Capitolo 4). Una tipica esecuzione del precedente programma produce il seguente risultato:

```
In esecuzione Uno
In esecuzione Uno
In esecuzione Due
In esecuzione Due
In esecuzione Due
In esecuzione Uno
In esecuzione Uno
In esecuzione Due
In esecuzione Due
In esecuzione Due
```

## 15.5. Interfaccia *Runnable*

Come accennato nel paragrafo 15.2, la classe *Thread* prevede i seguenti due costruttori:

```
Thread ( Runnable ru )
Thread ( Runnable ru , String nome )
```

L'argomento attuale, o il primo argomento attuale, di questi costruttori può essere una qualunque classe che implementa l'interfaccia *Runnable*, la quale prevede il metodo *run()* (anche la classe *Thread* implementa l'interfaccia *Runnable*).

È possibile quindi creare come thread un oggetto appartenente a una classe che non è derivata dalla classe *Thread* (può essere derivata da qualche altra classe), ma che implementa l'interfaccia *Runnable* (la derivazione è solo singola). Per esempio, si può avere:

```
class Cc
{ /* ... */ }

class ClasseRu extends Cc implements Runnable
{ // ...
    public void run()
    { // ...
    }
}

public class ThreadRu
{ public static void main()
  { ClasseRu cr = new ClasseRu();
    Thread tt = new Thread(cr);
    tt.start();
    // viene attivato il thread tt ed eseguito il
    // metodo run() dell'oggetto riferito da cr
    // ...
  }
}
```

## 15.6. Variabili comuni

Lo spazio di memoria comune tra tutti i thread di uno stesso processo comprende, oltre che variabili private di ciascun thread, anche variabili comuni a più thread. Per esempio, più thread appartenenti alla stessa classe hanno come variabili comuni le variabili statiche definite nella classe, oppure, più thread con una variabile riferimento della medesima classe possono individuare un oggetto comune (e quindi con gli stessi campi dati).

Quando più thread accedono alle stesse variabili per modificarne il valore, occorre che tutte le modifiche previste vengano effettuate per intero dai singoli thread, evitando che un thread possa trovare le stesse variabili in uno stato inconsistente. Per esempio, supponiamo che due thread *t1* e *t2* accedano, tramite il riferimento privato *dat*, ad un oggetto comune *dd*, modificandone e il campo intero *alfa* (sommandovi 10) e il campo intero *beta* (incrementandolo): senza precauzioni, il numero di somme di 10 al campo *alfa* e di incrementi del campo *beta* possono essere diversi:

```
class Dati
{   public int alfa;
    public int beta;
}

class UnThread extends Thread
{   private Dati dat;
    public UnThread(String id, Dati d)
    {   super(id); dat = d; }
    // l'argomento attuale corrispondente a d
    // potrebbe essere lo stesso per tutti i thread;
    // quindi, la variabile riferimento dat
    // di ciascun thread potrebbe individuare un
    // oggetto di classe Dati comune a tutti i thread;
    public void run()
    {   int y;
        while(!isInterrupted())
        {   y = dat.alfa + 10;
            Console.scriviStringa
                (getName() + " inizio for");
            for (int i =0; i < 1000000000; i++) ;
        }
    }
}
```

```

        Console.scriviStringa
            (getName() + " fine for");
        dat.alfa = y;
        // ipotesi:
        // dat riferisce un oggetto comune
        // durante il for si ha commutazione
        // y del primo thread vale 10
        // y del secondo thread vale 10
        // dopo i due assegnamenti, dat.alfa vale 10
        dat.beta++;
    }
}

public class ProvaComuni
{
    public static void main(String[] args)
    {
        Dati dd = new Dati(); String messaggio;
        // argomento attuale dd uguale per i due thread
        UnThread t1 = new UnThread("t1", dd);
        UnThread t2 = new UnThread("t2", dd);
        Console.scriviStringa
            ("Per partire scrivi vai, per terminare alt");
        for (;;)
        {
            messaggio = Console.leggiStringa();
            if (messaggio.equals("vai")) break;
        }
        t1.start(); t2.start();
        for (;;)
        {
            messaggio = Console.leggiStringa();
            if (messaggio.equals("alt")) break;
        }
        t1.interrupt(); t2.interrupt();
        while (t1.isAlive() || t2.isAlive()) ;
        Console.scriviStringa("alfa = " + dd.alfa);
        Console.scriviStringa("beta = " + dd.beta);
    }
}

```

Una tipica esecuzione produce il seguente risultato:

```

vai
t1 inizio for
t2 inizio for

```

```
t2 fine for
t2 inizio for
t1 fine for
t1 inizio for
t1 fine for
t1 inizio for
t2 fine for
t2 inizio for
alt
t2 fine for
t1 fine for
alfa = 30
beta = 60
```

## 15.7. Indivisibilità e mutua esclusione

In Java, ogni oggetto possiede un *lock*. Un blocco di istruzioni può essere *sincronizzato* su un oggetto, nel senso che il thread che lo esegue:

- attende finché non acquisisce il lock dell'oggetto (*esegue un lock*);
- esegue il blocco;
- rilascia il lock dell'oggetto (*esegue un unlock*).

Il blocco viene quindi eseguito in modo indivisibile, e più thread che hanno un blocco di istruzioni sincronizzato sullo stesso oggetto eseguono il rispettivo blocco in mutua esclusione. Blocchi sincronizzati su oggetti diversi possono essere eseguiti in concorrenza.

La sincronizzazione di un blocco di istruzioni si ottiene con l'istruzione:

***synchronized*** ( *expression* ) *block*

Il valore dell'espressione, nel caso di sincronizzazione su oggetti, è il riferimento di un oggetto.

L'oggetto di cui un thread acquisisce il lock non viene "chiuso": l'effetto che si ottiene è quello della mutua esclusione fra più thread che si sincronizzano su uno stesso oggetto (magari su quello su cui i thread stessi

operano). Se un thread non si sincronizza opera liberamente su quell'oggetto.

L'esempio del paragrafo precedente può essere riscritto sincronizzando sull'oggetto riferito da *dat* il blocco di istruzioni che vi operano:

```
class Dati
{   public int alfa;
    public int beta;
}

class UnThread1 extends Thread
{   private Dati dat;
    public UnThread1(String id, Dati d)
    {   super(id); dat = d;   }
    public void run()
    {   int y;
        while(!isInterrupted())
            synchronized(dat)
            {   // blocco eseguito in mutua esclusione
                // sull'oggetto riferito da dat
                y = dat.alfa + 10;
                Console.scriviStringa
                    (getName() + " inizio for");
                for (int i =0; i < 1000000000; i++) ;
                Console.scriviStringa
                    (getName() + " fine for");
                dat.alfa = y; dat.beta++;
            }
    }
}

public class ProvaComuni1
{   public static void main(String[] args)
    {   Dati dd = new Dati(); String messaggio;
        UnThread1 t1 = new UnThread1("t1",dd);
        UnThread1 t2 = new UnThread1("t2", dd);
        Console.scriviStringa
            ("Per partire scrivi vai, per terminare alt");
        for (;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("vai")) break;
        }
        t1.start(); t2.start();
    }
}
```

```

        for (;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("alt")) break;
        }
        t1.interrupt(); t2.interrupt();
        while (t1.isAlive() || t2.isAlive()) ;
        Console.scriviStringa("alfa = " + dd.alfa);
        Console.scriviStringa("beta = " + dd.beta);
    }
}

```

In questo modo, il numero di volte in cui e' stato sommato 10 al membro *alfa* dell'oggetto riferito da *dat* e il numero di volte in cui è stato incrementato il membro *beta* dello stesso oggetto sono sempre uguali.

La sincronizzazione può anche riguardare un intero metodo (il corpo di un metodo è un blocco), e può essere fatta sia sull'oggetto istanza implicito che sulla classe a cui il metodo appartiene.

Per sincronizzare un metodo sull'oggetto istanza implicito, occorre usare nell'intestazione del metodo (prima della specifica del tipo del risultato) il modificatore *synchronized*, ed è come se il corpo del metodo fosse costituito dalla seguente unica istruzione:

***synchronized ( this ) block***

Più metodi di una medesima classe sincronizzati sull'oggetto istanza implicito (quindi, su uno stesso oggetto) vengono eseguiti in mutua esclusione su ogni singolo oggetto appartenente alla classe stessa.

Come esempio di mutua esclusione su ogni singolo oggetto istanza, riportiamo un esempio in cui viene banalmente gestito un conto corrente:

```

class Conto
{   private int numPrelievi;
    private int numVersamenti;
    private String ultimaOp;
    private double importo;
    private double totale;
    public synchronized void versa(double somma)
    {   ultimaOp = "Versamento "; importo = somma;
        numVersamenti++; totale += somma;
    }
    // oppure:

```

```

    // public void versa(double somma)
    // {   synchronized(this)
    //     {   ultimaOp = "Versamento "; importo = somma;
    //         numVersamenti++; totale += somma;
    //     }
    // }
    public synchronized boolean preleva(double somma)
    {   if (totale < somma) return false;
        else
        {   ultimaOp = " Prelievo " ; importo = somma;
            numPrelievi++; totale -= somma; return true;
        }
    }
    public synchronized void estratto()
    {   Console.scriviStringa
        ("Numero dei prelievi = "  + numPrelievi);
        Console.scriviStringa
        ("Numero dei versamenti = "  + numVersamenti);
        Console.scriviStringa
        ("Ultima operazione = " + ultimaOp +
         "; Importo = " + importo);
        Console.scriviStringa("Totale = "  + totale);
    }
}

class Versamento extends Thread
{   private Conto cc;
    public Versamento(String id, Conto co)
    {   super(id); cc = co;   }
    public void run()
    {   int n = 200;
        while(!isInterrupted())
        {   // ...
            cc.versa(n);
            Console.scriviStringa
                (getName() + " ha versato " + n);
            for(int i = 0; i < 1000110000; i++) ;
        }
    }
}

class Prelievo extends Thread
{   private Conto cc;

```



```
public Prelievo(String id, Conto co)
{  super(id); cc = co;  }
public void run()
{  int n = 100; // ...
  while(!isInterrupted())
  {  // ...
    if (cc.preleva(n)) Console.scriviStringa
      (getName() + " ha prelevato " + n);
    else Console.scriviStringa
      (getName() + " prelievo non possibile ");
    for (int i = 0; i < 100000000; i++) ;
  }
}

public class ProvaCC
{  public static void main(String[] args)
  {  String messaggio; Conto mioc = new Conto();
    Versamento vv = new Versamento("vv", mioc);
    Prelievo pp = new Prelievo ("pp", mioc);
    Console.scriviStringa
      ("Per partire scrivi vai, per terminare alt");
    for (;;)
    {  messaggio = Console.leggiStringa();
      if (messaggio.equals("vai")) break;
    }
    vv.start(); pp.start();
    for (;;)
    {  messaggio = Console.leggiStringa();
      if (messaggio.equals("alt")) break;
    }
    mioc.estratto();
    vv.interrupt(); pp.interrupt();
  }
}
```

Una tipica esecuzione produce il seguente risultato:

```
vai
vv ha versato 200
pp ha prelevato 100
vv ha versato 200
pp ha prelevato 100
```

```
pp ha prelevato 100
alt
vv ha versato 200
Numero dei prelievi = 3
Numero dei versamenti = 3
Ultima operazione = Versamento ; Importo = 200.0
Totale = 300.0
```

Quando in una classe vi sono variabili statiche, la sincronizzazione su ogni singolo oggetto classe non è significativa, in quanto oggetti diversi hanno accesso alle stesse variabili. Occorre quindi una sincronizzazione sulla classe, e non sull'oggetto classe. Per un metodo della classe, questo si ottiene inserendo nel suo corpo la seguente unica istruzione:

***synchronized ( getClass() ) block***

Il metodo *getClass()* fa parte della classe *Object*, e restituisce la classe a cui appartiene l'oggetto a cui si applica (in questo caso, quello implicito, riferito da *this*).

## 15.8. Comunicazione fra thread

Quando si hanno più thread che operano su variabili comuni, può accadere che un thread non possa procedere fino a quando un altro thread abbia compiuto opportune operazioni su tali variabili: all'interno della mutua esclusione, vi può quindi essere l'esigenza di uno scambio di informazioni, per cui è stato previsto un meccanismo apposito di comunicazione.

Ogni oggetto (appartenente quindi alla classe *Object*) ha associato un insieme di thread bloccati sull'oggetto stesso (*wait-set*). Inoltre, nella classe *Object* sono definiti i metodi *wait()* e *notify()*, che agiscono su tale insieme con le seguenti regole:

***public final void wait () throws InterruptedException***

- deve essere eseguito su un oggetto del quale il thread corrente ha acquisito il lock;
- aggiunge il thread allo wait-set dell'oggetto;
- blocca il thread;
- produce il rilascio del lock sull'oggetto e di eventuali altri lock acquisiti dal thread.

***public final void notify ()***

- deve essere eseguito su un oggetto del quale il thread corrente ha acquisito il lock;
- rimuove uno dei thread dallo wait-set dell'oggetto;
- il thread rimosso:
  1. riesegue automaticamente il lock sull'oggetto (può proseguire solo quando il thread corrente ha eseguito un unlock sull'oggetto);
  2. riesegue automaticamente eventuali altri lock acquisiti nel momento in cui ha eseguito il metodo *wait()*.

Al ritorno dal metodo *wait()*, il thread rimosso si trova esattamente nelle stesse condizioni di quando ha invocato il metodo *wait()* stesso, e riparte dall'istruzione successiva. Se il metodo *wait()* viene eseguito sotto condizione, al ritorno dal metodo occorre verificare nuovamente la condizione ed eventualmente rieseguire il metodo stesso. Per esempio, è corretto scrivere:

```
// ...
synchronized(o1)
{ while(condizione-non-verificata) o1.wait();
  // condizione verificata;
  // altre istruzioni
}
// ...
```

Al ritorno dal metodo *wait()* il ciclo viene ripetuto, e viene eseguito nuovamente l'esame della condizione prevista dall'istruzione *while* stessa.

Il metodo *wait()* possiede due altre forme alternative:

***public final void wait ( long millisecondi )***

```
throws InterruptedException  
public final void wait ( long millisecondi , int nanosecondi )  
throws InterruptedException
```

La differenza è che il thread, trascorso il tempo specificato, se non è stato ancora rimosso, viene rimosso dallo wait-set dell'oggetto.

Anche il metodo *notify()* prevede un'altra forma alternativa:

```
public final void notifyAll ()
```

La differenza è che rimuove tutti i thread dallo wait-set dell'oggetto. Tale forma alternativa viene utilizzata per evitare situazioni di blocco, come verrà illustrato nel sottoparagrafo 15.8.2.

### 15.8.1. Eccezioni prodotte da *wait()* e *notify()*

Il metodo *wait()*, se il thread che lo esegue ha il flag di interruzione ad 1, oppure se tale flag va ad 1 mentre il thread si trova nello wait-set di un oggetto, solleva un'eccezione di classe *InterruptedException* e il flag di interruzione viene azzerato (tale eccezione deve essere obbligatoriamente gestita).

Inoltre, i metodi *wait()* e *notify()* sollevano un'eccezione di classe *IllegalMonitorStateException* (sottoclasse di *RuntimeException*, e quindi unchecked) nel caso in cui non sia stato acquisito il *lock* sull'oggetto.

### 15.8.2. Esempi

In questo primo esempio, viene gestita una casella postale appartenente alla classe *Casella*, che contiene una sola cella appartenente alla classe *Informazione* (questa contenente a sua volta un intero). La classe *Casella* prevede due metodi *synchronized* *get()* e *put()*, che usano *wait()* e *notifyAll()*. Vengono poi previsti thread appartenenti alle classi *Immetti* ed *Estrai*, che utilizzano una casella postale alla quale applicano, rispettivamente, i metodi *put()* e *get()*. Il programma di prova utilizza una casella *miaca*, due thread *i1* e *i2* che vi fanno immissioni e un thread *ee* che vi fa estrazioni.

```
class Informazione
{ public int a; }

class Casella
{ private Informazione cella;
  public synchronized void put(Informazione og)
    throws InterruptedException
  { while (cella != null) wait();
    // blocca il thread sull'oggetto istanza
    cella = og;
    Console.scriviStringa
      (Thread.currentThread().getName() +
        " ha immesso " + og.a);
    notifyAll();
    // rimuove i thread bloccati sull'oggetto istanza
  }
  public synchronized Informazione get()
    throws InterruptedException
  { Informazione og;
    while (cella == null) wait();
    // blocca il thread sull'oggetto istanza
    og = cella; cella = null;
    Console.scriviStringa
      (Thread.currentThread().getName() +
        " ha prelevato " + og.a);
    notifyAll();
    // rimuove i thread bloccati sull'oggetto istanza
    return og;
  }
}

class Metti extends Thread
{ private Casella box;
  public Metti(String id, Casella ca)
  { super(id); box = ca; }
  public void run()
  { Informazione ob; int n = 0;
    try
    { while(!isInterrupted())
      { ob = new Informazione(); ob.a = n; n++;
        box.put(ob);
        // altre operazioni
        for(int i=0; i<100000000; i++);
      }
    }
  }
}
```

```

        }
    }
    catch (InterruptedException e) { }
}

class Prendi extends Thread
{
    private Casella box;
    public Prendi(String id, Casella ca)
    {
        super(id); box = ca;
    }
    public void run()
    {
        Informazione ob;
        try
        {
            while(!isInterrupted())
            {
                ob = box.get();
                // altre operazioni
                for(int i=0; i<100000000; i++);
            }
        }
        catch (InterruptedException e) { }
    }
}

public class ProvaCasella
{
    public static void main(String[] args)
    {
        String messaggio;
        Casella miaca = new Casella();
        Metti m1 = new Metti("m1", miaca);
        Metti m2 = new Metti("m2", miaca);
        Prendi p1 = new Prendi("p1", miaca);
        Console.scriviStringa
            ("Per partire scrivi vai, per terminare alt");
        for (;;)
        {
            messaggio = Console.leggiStringa();
            if (messaggio.equals("vai")) break;
        }
        m1.start(); m2.start(); p1.start();
        for (;;)
        {
            messaggio = Console.leggiStringa();
            if (messaggio.equals("alt")) break;
        }
        m1.interrupt();
        m2.interrupt();
    }
}

```

```

        p1.interrupt();
    }
}

```

Una tipica esecuzione produce il seguente risultato:

```

vai
m1 ha immesso 0
p1 ha prelevato 0
m2 ha immesso 0
p1 ha prelevato 0
m2 ha immesso 1
p1 ha prelevato 1
m1 ha immesso 1
p1 ha prelevato 1
alt

```

Supponiamo di utilizzare, nella classe *Casella*, il metodo *notify()* invece del metodo *notifyAll()*, e ipotizziamo la seguente sequenza di eventi (partendo da casella postale piena):

- *m1* bloccato su *wait()*, *m2* bloccato su *wait()*, *p1* in esecuzione;
- *p1* estrae, sblocca *m1* con *notify()*, poi si blocca su *wait()*;
- *m1* immette, sblocca *m2* con *notify()*, poi si blocca su *wait()*;
- *m2* si blocca su *wait()*.

I tre thread sono bloccati, e si è in una situazione di stallo.

Per verificare sperimentalmente il prodursi di una situazione di stallo in pochi secondi di esecuzione, suggeriamo (oltre alla sostituzione di *notifyAll()* con *notify()* nella classe *Casella*) di fare le seguenti modifiche al programma precedente:

- classi *Metti* e *Prendi*: diminuire il periodo di attesa, sostituendo nella istruzione *for* 100000000 con 1000000;
- classe *ProvaCasella*: sostituirla con la seguente, nella quale vengono creati 10 thread che immettono e 10 thread che prelevano:

```

public class ProvaCasella
{ public static void main(String[] args)
    throws InterruptedException

```

```

{ String messaggio;
  int N = 10;
  Casella miaca = new Casella();
  Prendi[] ap = new Prendi[N];
  Metti[] am = new Metti[N];
  Console.scriviStringa
    ("Per partire scrivi vai, per terminare alt");
  for (;;)
  { messaggio = Console.leggiStringa();
    if (messaggio.equals("vai")) break;
  }
  for(int i=0; i<N; i++)
  { (ap[i] = new Prendi("p"+i, miaca)).start();
    (am[i] = new Metti("m"+i, miaca)).start();
  }
  for (;;)
  { messaggio = Console.leggiStringa();
    if (messaggio.equals("alt")) break;
  }
  for (int i=0; i<10; i++)
  { ap[i].interrupt();
    am[i].interrupt();
  }
}
}

```

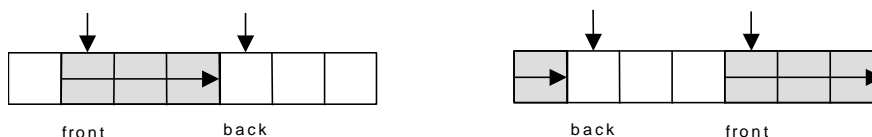


Figura 15.2. Buffer realizzato con un array circolare e due puntatori: due situazioni di parziale riempimento

Nel secondo esempio, viene gestita una casella postale appartenente alla classe *Buffer*, che contiene più celle appartenenti alla classe *Informazione* organizzate a coda (ogni cella è in grado di contenere un riferimento di *Informazione*). La gestione avviene utilizzando un array circolare e due puntatori (indici) *back* e *front*, che rappresentano,



rispettivamente, la posizione per la prossima immissione e quella per il prossimo prelievo (Fig. 15.2). Si ipotizza che il numero di elementi dell'array (contenuto in *maxElem*) sia specifico per ogni oggetto *Buffer*, e che il numero di riferimenti presenti (variabili di tipo *Informazione*) sia contenuto in *quanti*.

Il programma di prova utilizza un buffer *bu*, due thread *im1* e *im2* che vi fanno immissioni e un thread *pe1* che vi fa prelievi.

```
class Informazione
{   public int a;   }

class Buffer
{   private int back, front,
        quanti, maxElem;
    private Informazione[] buffer;
    public Buffer(int n)
    {   maxElem = n;
        buffer = new Informazione[maxElem];
    }
    public synchronized void put(Informazione og)
        throws InterruptedException
    {   while (quanti == maxElem) wait();
        buffer[back] = og; quanti++;
        back = (back+1)%buffer.length;
        Console.scriviStringa
            (Thread.currentThread().getName() +
              " ha immesso " + og.a);
        notifyAll();
    }
    public synchronized Informazione get()
        throws InterruptedException
    {   Informazione og;
        while (quanti == 0) wait();
        og = buffer[front]; quanti--;
        front = (front+1)%buffer.length;
        Console.scriviStringa
            (Thread.currentThread().getName() +
              " ha prelevato " + og.a);
        notifyAll();
        return og;
    }
    public synchronized void stampa()
```

```

    { Console.scriviStringa
      ("Numero elementi " + quanti);
      int k = front;
      for(int i = 0; i < quanti; i++)
      { Console.scriviInt(buffer[k].a);
        k=(k+1)%buffer.length;
      }
      Console.nuovaLinea();
    }
  }

class Immetti extends Thread
{ private Buffer buf;
  public Immetti(String id, Buffer b)
  { super(id); buf = b; }
  public void run()
  { Informazione ob; int n = 0;
    try
    { while(!isInterrupted())
      { ob = new Informazione(); ob.a = n; n++;
        buf.put(ob);
        // altre operazioni
        for(int i=0; i<100000000; i++);
      }
    }
    catch(InterruptedException e) { }
  }
}

class Preleva extends Thread
{ private Buffer buf;
  public Preleva(String id, Buffer b)
  { super(id); buf = b; }
  public void run()
  { Informazione ob;
    try
    { while(!isInterrupted())
      { ob= buf.get();
        // altre operazioni
        for(int i=0; i<100000000; i++);
      }
    }
    catch (InterruptedException e) { }
  }
}

```

```
    }  
}  
  
public class ProvaBuffer  
{ public static void main(String[] args)  
  { String messaggio;  
    Buffer bu = new Buffer(5);  
    Immetti im1 = new Immetti("im1", bu);  
    Immetti im2 = new Immetti("im2", bu);  
    Preleva pe1 = new Preleva("pe1", bu);  
    Console.scriviStringa  
      ("Per partire scrivi vai, per terminare alt");  
    for (;;)   
    { messaggio = Console.leggiStringa();  
      if (messaggio.equals("vai")) break;  
    }  
    im1.start(); im2.start();  
    pe1.start();  
    for (;;)   
    { messaggio = Console.leggiStringa();  
      if (messaggio.equals("alt")) break;  
    }  
    bu.stampa();  
    im1.interrupt(); im2.interrupt();  
    pe1.interrupt();  
  }  
}
```

Una tipica esecuzione produce il seguente risultato:

```
vai  
im1 ha immesso 0  
im2 ha immesso 0  
pe1 ha prelevato 0  
im1 ha immesso 1  
im2 ha immesso 1  
pe1 ha prelevato 0  
im1 ha immesso 2  
im2 ha immesso 2  
pe1 ha prelevato 1  
im1 ha immesso 3  
im2 ha immesso 3  
pe1 ha prelevato 1
```

```

alt
Numero di elementi 4
2 2 3 3

```

Nel terzo esempio, viene considerata una casella postale appartenente alla classe *Coda*, la quale gestisce una lista i cui elementi hanno come campo informazione riferimenti di *Informazione* (Fig. 15.3). La classe *Coda* prevede due riferimenti *testa* e *fondo* al primo e all'ultimo elemento della lista, con inserimenti in fondo alla lista ed estrazioni dalla testa della lista stessa. Si ipotizza che il massimo numero di elementi in lista sia stabilito per ogni oggetto *Coda* (contenuto in *maxElem*), e che il numero di elementi presenti sia contenuti in *quanti*. Il programma di prova utilizza una coda *q*, due thread *in1* e *in2* che vi fanno immissioni e un thread *es1* che vi fa estrazioni.

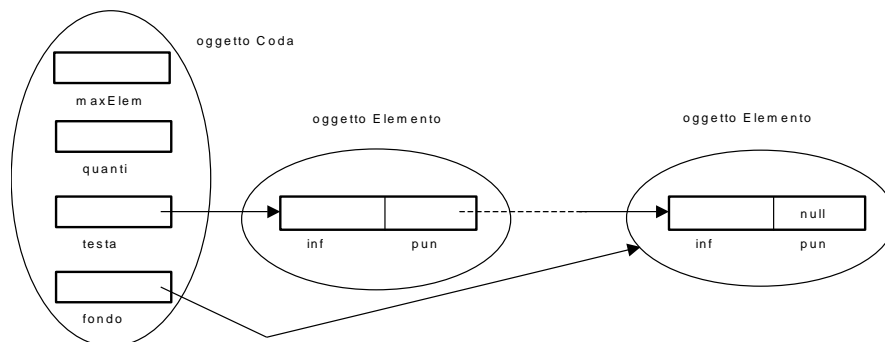


Figura 15.3. Coda realizzata con una lista e che contiene due puntatori, il numero di elementi e il numero massimo di elementi

```

class Informazione
{ public int a; }

class Coda
{ private class Elemento
  { Informazione inf; Elemento pun; }
  private int quanti = 0, maxElem;
  private Elemento testa = null, fondo = null;

```

```

public Coda(int n)
{   maxElem = n;   }
public synchronized void inscoda(Informazione og)
    throws InterruptedException
{   while (quanti == maxElem) wait();
    Elemento p = new Elemento();
    p.inf = og; p.pun = null; quanti++;
    if (testa == null) { testa = p; fondo = p; }
    else { fondo.pun = p; fondo = p; }
    Console.scriviStringa
        (Thread.currentThread().getName() +
        " ha immesso " + og.a);
    notifyAll();
}
public synchronized Informazione estcoda()
    throws InterruptedException
{   Informazione og;
    while (quanti == 0) wait();
    og = testa.inf; testa = testa.pun; quanti--;
    if (testa == null) fondo = null;
    Console.scriviStringa
        (Thread.currentThread().getName() +
        " ha prelevato " + og.a);
    notifyAll();
    return og;
}
public synchronized void stampa()
{   Elemento p = testa;
    if (testa == null)
    { Console.scriviStringa("Coda vuota"); return; }
    Console.scriviStringa
        ("Numero elementi " + quanti);
    while (p != null)
    { Console.scriviInt(p.inf.a); p = p.pun; }
    Console.nuovaLinea();
}
}

class Inserisci extends Thread
{   private Coda c;
    public Inserisci(String id, Coda cc)
    {   super(id); c = cc;   }
    public void run()

```

```

    {   Informazione ob; int n = 0;
        try
        {   while (!isInterrupted())
            {   ob = new Informazione(); ob.a = n; n++;
                c.inscoda(ob);
                // altre operazioni
                for(int i=0; i<100000000; i++);
            }
        }
        catch (InterruptedException e) {}
    }
}

class Estrai extends Thread
{   private Coda c;
    public Estrai(String id, Coda cc)
    {   super(id); c = cc;   }
    public void run()
    {   Informazione ob;
        try
        {   while (!isInterrupted())
            {   ob = c.estcoda();
                // altre operazioni
                for(int i=0; i<100000000; i++);
            }
        }
        catch (InterruptedException e ) {}
    }
}

public class ProvaCoda
{   public static void main(String[] args)
    {   String messaggio;
        Coda q = new Coda(5);
        Inserisci in1 = new Inserisci("In1", q);
        Inserisci in2 = new Inserisci("In2", q);
        Estrai es1 = new Estrai("Es1", q);
        Console.scriviStringa
            ("Per partire scrivi vai, per terminare alt");
        for (;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("vai")) break;
        }
    }
}

```

```

        in1.start(); in2.start(); es1.start();
        for (;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("alt")) break;
        }
        q.stampa();
        in1.interrupt(); in2.interrupt();
        es1.interrupt();
    }
}

```

Il problema di definire una casella postale con più celle può essere risolto introducendo una classe astratta *Box*, e due classi da essa derivate *BoVett* e *BoList* che costituiscono due possibili implementazioni. La classe astratta, oltre al costruttore, prevede campi dati *quanti* e *maxElem*, e i metodi astratti *put()*, *get()* e *stampa()*. Un metodo astratto non può avere il modificatore *synchronized* (sincronizzazione sull'oggetto corrente), in quanto non si possono avere oggetti di una classe astratta: un metodo in una classe derivata è ridefinito per overriding se la sua intestazione differisce da quella del corrispondente metodo astratto solo per il modificatore *synchronized*.

Le due classi che rappresentano thread per l'immissione (*Inserisci*) e l'estrazione (*Estrai*) sono le stesse nei due casi. Il programma è il seguente:

```

class Informazione
{   public int a;   }

abstract class Box
{   protected int quanti, maxElem;
    public Box(int n)
    {   maxElem = n;   }
    public abstract void put(Informazione og)
        throws InterruptedException;
    public abstract Informazione get()
        throws InterruptedException;
    public abstract void stampa();
}

class BoVett extends Box
{   private int back, front, quanti;

```

```

private Informazione[] vett;
public BoVett(int n)
{   super(n);
    vett = new Informazione[maxElem];
}
public synchronized void put(Informazione og)
    throws InterruptedException
{   while (quanti == maxElem) wait();
    vett[back] = og; quanti++;
    back = (back+1)%vett.length;
    Console.scriviStringa
        (Thread.currentThread().getName() +
         " ha immesso " + og.a);
    notifyAll();
}
public synchronized Informazione get()
    throws InterruptedException
{   Informazione og;
    while (quanti == 0) wait();
    og = vett[front]; quanti--;
    front = (front+1)%vett.length;
    Console.scriviStringa
        (Thread.currentThread().getName() +
         " ha prelevato " + og.a);
    notifyAll();
    return og;
}
public synchronized void stampa()
{   Console.scriviStringa
        ("Numero elementi " + quanti);
    int k = front;
    for(int i = 0; i < quanti; i++)
    {   Console.scriviInt(vett[k].a);
        k=(k+1)%vett.length;
    }
    Console.nuovaLinea();
}
}

class BoList extends Box
{   private class Elemento
    {   Informazione inf; Elemento pun;   }
    BoList(int n)

```



```

    { super(n);
    }
    private Elemento testa = null, fondo = null;
    public synchronized void put(Informazione og)
        throws InterruptedException
    { while (quanti == maxElem) wait();
      Elemento p = new Elemento();
      p.inf = og; p.pun = null; quanti++;
      if (testa == null) { testa = p; fondo = p; }
      else { fondo.pun = p; fondo = p; }
      Console.scriviStringa
        (Thread.currentThread().getName() +
          " ha immesso " + og.a);
      notifyAll();
    }
    public synchronized Informazione get()
        throws InterruptedException
    { Informazione og;
      while (quanti == 0) wait();
      og = testa.inf; testa = testa.pun; quanti--;
      if (testa == null) fondo = null;
      Console.scriviStringa
        (Thread.currentThread().getName() +
          " ha prelevato " + og.a);
      notifyAll();
      return og;
    }
    public synchronized void stampa()
    { Elemento p = testa;
      if (testa == null)
        { Console.scriviStringa("Coda vuota"); return; }
      Console.scriviStringa
        ("Numero elementi " + quanti);
      while (p != null)
        { Console.scriviInt(p.inf.a); p = p.pun; }
      Console.nuovaLinea();
    }
  }

class Inserisci extends Thread
{ private Box c;
  public Inserisci(String id, Box cc)
  { super(id); c = cc; }
}

```

```

    public void run()
    {   Informazione ob; int n = 0;
        try
        {   while (!isInterrupted())
            {   ob = new Informazione(); ob.a = n; n++;
                c.put(ob);
                // altre operazioni
                for(int i=0; i<100000000; i++);
            }
        }
        catch (InterruptedException e) {}
    }
}

class Estrai extends Thread
{   private Box c;
    public Estrai(String id, Box cc)
    {   super(id); c = cc;   }
    public void run()
    {   Informazione ob;
        try
        {   while (!isInterrupted())
            {   ob = c.get();
                // altre operazioni
                for(int i=0; i<100000000; i++);
            }
        }
        catch (InterruptedException e ) {}
    }
}

public class ProvaBox
{   public static void main(String[] args)
    {   String messaggio;
        Box bb = new BoVett(5); // oppure = new BoList
        Inserisci in1 = new Inserisci("In1", bb);
        Inserisci in2 = new Inserisci("In2", bb);
        Estrai es1 = new Estrai("Es1", bb);
        Console.scriviStringa
            ("Per partire scrivi vai, per terminare alt");
        for (;;)
        {   messaggio = Console.leggiStringa();
            if (messaggio.equals("vai")) break;
        }
    }
}

```

```
    }  
    in1.start(); in2.start(); es1.start();  
    for (;;)   
    {   messaggio = Console.leggiStringa();  
        if (messaggio.equals("alt")) break;  
    }  
    bb.stampa();  
    in1.interrupt(); in2.interrupt();  
    es1.interrupt();  
}  
}
```

## 15.9. Thread e garbage collection

Come detto nel Capitolo 8, la liberazione della memoria dinamica avviene ad opera di una routine (*Garbage Collector*) che, periodicamente, distrugge gli oggetti per i quali non esiste più alcun riferimento.

Se l'oggetto appartiene alla classe *Thread* o a una classe da essa derivata, la distruzione avviene se è soddisfatta anche la condizione che il corrispondente thread si trova nello stato “dead”.